

Inspector 4.7

User Manual

Riscure BV

Inspector 4.7: User Manual

Riscure BV

Publication date November 4, 2013

Copyright © 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013 Riscure BV

The information in this document, including but not limited to, text, graphics, images is the property of Riscure and is protected by Dutch and international copyright laws. Unauthorised use or distribution of any material of this document may violate copyright, trademark, and other laws and is subject to civil as well as criminal sanctions. The document, or any proportion of it may not, except where otherwise indicated, be reproduced, duplicated, copied, transferred, distributed, stored or otherwise exploited for any commercial use without prior written permission by Riscure. Modifications to the contents of the document are expressly prohibited.

You agree to prevent any unauthorised copying of the materials and to ensure that all employees, where applicable, of your organisation adheres to these limitations.

The documentation may contain text, graphics or images that are subject to the copyright rights of their providers.

Table of Contents

| | |
|--|------|
| Preface | xvii |
| 1. Introduction | 1 |
| 1.1. Overview | 1 |
| 1.2. Contents | 2 |
| 1.3. Glossary | 3 |
| 2. Inspector Software | 8 |
| 2.1. User interface | 8 |
| 2.2. Function modules | 11 |
| 2.3. Settings | 12 |
| 2.4. User and System Space | 18 |
| 2.5. Profiles | 18 |
| 2.6. Tools menu | 22 |
| 2.7. Installation and configuration | 23 |
| 2.8. Using the GPU computing power: CUDA | 25 |
| 2.9. License | 28 |
| 2.10. Third party software | 28 |
| 3. Cryptography | 30 |
| 3.1. Attacking cryptographic algorithms | 30 |
| 3.2. Migrating pre-4.6 Ciphers | 30 |
| 3.3. Cipher selection | 30 |
| 3.4. Direction | 31 |
| 3.5. Settings | 32 |
| 3.5.1. Key | 32 |
| 3.5.2. Candidates | 32 |
| 3.5.3. Method | 33 |
| 3.5.4. Criterion | 33 |
| 3.5.5. Report | 34 |
| 3.5.6. Generate traces | 34 |
| 3.6. Leakages | 34 |
| 3.6.1. Normal View | 35 |
| 3.6.2. Expert View | 36 |
| 3.6.3. Leakage models | 37 |
| 3.7. Key Intermediates | 38 |
| 4. Side Channel Acquisition | 40 |
| 4.1. Select the acquisition methodology | 40 |
| 4.1.1. Legacy vs new acquisition | 40 |
| 4.1.2. Migrating to the new acquisition | 40 |
| 4.2. Hardware setup tasks | 41 |
| 4.2.1. Communication devices | 41 |
| 4.2.2. Acquisition devices | 41 |
| 4.3. Using Acquisition modules | 44 |
| 4.3.1. Oscilloscope | 44 |
| 4.3.2. Device | 45 |
| 4.4. Using Acquisition2 modules | 46 |
| 4.4.1. Acquisition2 Framework | 46 |
| 4.4.2. Data Generator | 47 |
| 4.5. Modifying modules | 48 |
| 5. Fault Injection | 49 |
| 5.1. Preparing the hardware setup | 49 |
| 5.1.1. Device List | 49 |
| 5.1.2. Configuration A: ISO7816 smartcard with power glitching | 50 |

| | |
|---|-----|
| 5.1.3. Configuration B: IO Device with optical fault injection | 51 |
| 5.1.4. Configuration C: Embedded target with voltage fault injection | 52 |
| 5.1.5. Configuration D: Multi-area optical fault injection and protection | 52 |
| 5.1.6. Configuration E: IO Device with EM fault injection | 53 |
| 5.2. Select the perturbation methodology | 53 |
| 5.2.1. Legacy vs new perturbation | 53 |
| 5.2.2. Migrating to the new perturbation | 54 |
| 5.3. Electrical perturbation | 54 |
| 5.3.1. FI setup | 54 |
| 5.3.2. Triggered Perturbation | 55 |
| 5.4. Dynamic perturbation | 56 |
| 5.4.1. Conceptual overview | 56 |
| 5.4.2. Communication protocols | 57 |
| 5.4.3. Snippets and Programs | 58 |
| 5.4.4. Perturbation variables | 58 |
| 5.5. XY(Z) Perturbation (Optical/EM-FI) | 58 |
| 5.5.1. Coordinate Systems: XYZ Device vs. Chip | 59 |
| 5.5.2. Scan Area | 61 |
| 5.5.3. Microscope | 62 |
| 5.5.4. XY(Z) Devices | 64 |
| 6. Inspector Hardware Components | 67 |
| 6.1. Hardware Manager | 67 |
| 6.1.1. Overview | 67 |
| 6.1.2. Settings | 68 |
| 6.1.3. New devices | 68 |
| 6.2. Oscilloscopes | 69 |
| 6.2.1. Setup and use of the LeCroy | 69 |
| 6.2.2. Setup and use of the PicoScope | 75 |
| 6.2.3. IVI-compliant oscilloscopes | 75 |
| 6.3. Side Channel Analysis | 79 |
| 6.3.1. Power Tracer 3 | 79 |
| 6.3.2. Power Tracer 4 | 83 |
| 6.3.3. Current Probe | 87 |
| 6.3.4. EM Probe Station | 88 |
| 6.3.5. MicroPross MP300 | 92 |
| 6.3.6. RF Tracer | 96 |
| 6.4. Fault Injection | 98 |
| 6.4.1. VC Glitcher | 98 |
| 6.4.2. Diode Laser Station | 104 |
| 6.4.3. Splitter | 109 |
| 6.4.4. EM-FI Transient Probe | 111 |
| 6.5. Signal filtering | 113 |
| 6.5.1. icWaves | 113 |
| 6.5.2. CleanWave | 129 |
| 6.5.3. Hardware Filters | 133 |
| 6.6. Managing Targets | 135 |
| 6.6.1. TargetManager | 135 |
| 6.6.2. Java Card as target | 135 |
| 6.7. XMEGA | 136 |
| 6.7.1. XMEGA A3 training board | 136 |
| 7. Real-Time pattern matching | 139 |
| 7.1. Introduction | 139 |
| 7.2. Software | 140 |
| 7.3. Detecting patterns in raw signals | 140 |

| | |
|---|-----|
| 7.4. Reducing noise with the filter | 144 |
| 8. Software Development for Inspector | 147 |
| 8.1. Overview | 147 |
| 8.2. Module Wizard | 149 |
| 8.3. Developing Classic Modules | 150 |
| 8.3.1. Module development | 150 |
| 8.3.2. Module compatibility | 152 |
| 8.3.3. Module Methods | 153 |
| 8.3.4. Developing custom cryptographic ciphers | 154 |
| 8.3.5. Module development with Eclipse | 167 |
| 8.3.6. Writing perturbation modules | 174 |
| 8.3.7. Writing triggered perturbation modules | 181 |
| 8.4. Inspector OSGi framework | 184 |
| 8.4.1. OSGi plugins | 184 |
| 8.5. Developing application protocols | 186 |
| 8.6. Developing device drivers | 187 |
| 8.6.1. Driver classes | 187 |
| 8.6.2. Device categories | 188 |
| 8.6.3. Guidelines | 188 |
| 8.7. Developing measurement setups | 188 |
| 8.7.1. Measurement setup life cycle | 189 |
| 8.7.2. Advanced measurement setups | 192 |
| 8.8. Developing targets | 199 |
| 8.8.1. Target life cycle | 200 |
| 8.9. Developing target managers | 203 |
| 8.10. Developing transport protocols | 203 |
| 8.11. Developing custom user interfaces | 203 |
| 9. The Developer How-Do-I | 204 |
| 9.1. How do I implement an AGG compatible JavaBean? | 204 |
| 9.1.1. What is a JavaBean and what is a bean property? | 204 |
| 9.1.2. How do I obtain an (editable) user interface view for a JavaBean instance? | 205 |
| 9.1.3. How do I implement the MVC pattern with the AGG? | 206 |
| 9.1.4. How do I set constraints on a (single) bean property? | 209 |
| 9.1.5. How do I constrain a relationship amongst bean properties? | 212 |
| 9.1.6. How do I find all constraint violations on a JavaBean instance? | 215 |
| 9.2. How do I control the way properties are displayed by the AGG? | 216 |
| 9.2.1. How do I change the display name of a property? | 216 |
| 9.2.2. How do I add a tag or description to a property? | 216 |
| 9.2.3. How do I add a unit to a property? | 216 |
| 9.2.4. How do I control the way a number editor is being displayed? | 216 |
| 9.2.5. How do I display a range slider? | 221 |
| 9.2.6. How do I display a drop down list to allow an end-user to select a value? | 222 |
| 9.2.7. How do I display a check box? | 227 |
| 9.2.8. How do I display an arbitrary amount of primitives? | 227 |
| 9.2.9. How do I display an arbitrary amount of nested beans? | 228 |
| 9.2.10. How do I display an open file dialog? | 231 |
| 9.3. How do I show/hide certain properties in the AGG? | 232 |
| 9.3.1. How do I always hide a property? | 232 |
| 9.3.2. How do I show or hide advanced/expert properties? | 232 |
| 9.3.3. How do I show or hide properties based on a certain condition? | 233 |
| 9.4. How do I annotate properties at runtime rather than at compile-time? | 236 |
| 9.4.1. Instantiating annotations at runtime | 236 |

| | |
|--|-----|
| 9.4.2. The @Programmatic annotation | 237 |
| 9.4.3. The thin wrapper that makes @Programmatic work | 238 |
| 9.5. How do I expose properties for tuning during module execution time? | 239 |
| 9.5.1. The @Tunable annotation | 239 |
| 9.5.2. Acquire a JPanel filled with @Tunable annotated properties | 240 |
| 9.5.3. How to popup the tunable panel during module execution? | 240 |
| 9.6. How do I extend the AGG? | 240 |
| 9.7. How do I implement an acquisition2/perturbation2 compatible hardware device driver? | 240 |
| 9.7.1. How do I register and unregister a device in the HWM? | 241 |
| 9.7.2. How do I make Inspector detect and call my device driver code? | 242 |
| 9.7.3. How do I register a 'dummy' device? | 242 |
| 9.7.4. How do I register and unregister devices on the basis of PnP events?.... | 245 |
| 9.7.5. How do I allow an end-user to manually register and unregister a device in the HWM? | 251 |
| 9.7.6. How do I implement a session on a device? | 255 |
| 9.7.7. How do I implement specific specialisations of the Session interface?.... | 257 |
| 9.7.8. How do I implement the OscilloscopeSession interface? | 257 |
| 9.7.9. How do I implement the IODeviceSession interface? | 265 |
| 9.8. How do I implement an acquisition2/perturbation2 application protocol? | 267 |
| 9.8.1. The Application Protocol Wizard | 267 |
| 9.8.2. How do I allow an end user to configure the protocol for an acquisition? | 270 |
| 9.8.3. How do I implement the protocol execution? | 273 |
| 9.8.4. How do I implement the communication log according to the expectation of the acquisition2 and perturbation2 frameworks? | 275 |
| 9.8.5. How should I obtain input data for my protocol? | 276 |
| 9.8.6. How can I close resources opened by my protocol at the end of an acquisition or perturbation? | 277 |
| 9.9. How do I implement an acquisition2/perturbation2 raw protocol? | 277 |
| 9.9.1. Developing a Raw Protocol | 277 |
| 9.9.2. Embedded Boot Glitching With a Warm Reset | 278 |
| A. Third party software licenses | 281 |
| B. Release notes | 292 |
| B.1. Release notes Inspector 4.0 | 292 |
| B.2. Release notes Inspector 4.1 | 293 |
| B.3. Release notes Inspector 4.1.1 | 297 |
| B.4. Release notes Inspector 4.2 | 299 |
| B.5. Release notes Inspector 4.2.1 | 302 |
| B.6. Release notes Inspector 4.3 | 303 |
| B.7. Release notes Inspector 4.4 | 305 |
| B.8. Release notes Inspector 4.5 | 307 |
| B.9. Release notes Inspector 4.5.1 | 310 |
| B.10. Release notes Inspector 4.6 | 311 |
| C. Keyboard Shortcuts | 316 |
| D. MP300 Device driver parameters | 317 |
| E. Modules List | 319 |
| E.1. Classic Acquisition | 319 |
| E.1.1. SideChannelAcquisition | 319 |
| E.1.2. XYAcquisition | 321 |
| E.1.3. AES Acquisition | 322 |
| E.1.4. DES Acquisition | 323 |
| E.1.5. DSA Acquisition | 324 |
| E.1.6. ECC Acquisition | 324 |

| | |
|---|-----|
| E.1.7. ECDSA Acquisition | 325 |
| E.1.8. Gsm Acquisition | 325 |
| E.1.9. IOCTL Acquisition | 326 |
| E.1.10. Oscilloscope | 326 |
| E.1.11. RSA Acquisition | 328 |
| E.1.12. Sasebo Acquisition | 328 |
| E.1.13. USB Token Acquisition | 330 |
| E.2. Acquisition2 | 330 |
| E.2.1. Acquisition2 Modules | 330 |
| E.3. Align | 334 |
| E.3.1. Dynamic Alignment | 334 |
| E.3.2. Elastic Alignment | 337 |
| E.3.3. ElasticAverage | 340 |
| E.3.4. ElasticRadius | 341 |
| E.3.5. Round align | 343 |
| E.3.6. Static Align | 344 |
| E.3.7. Stretch | 346 |
| E.4. Analysis | 347 |
| E.4.1. AutoCorrelation | 347 |
| E.4.2. Correlation | 349 |
| E.4.3. CrossCorrelation | 351 |
| E.4.4. PatternExtract | 352 |
| E.4.5. PatternMatch | 354 |
| E.4.6. Spectrogram | 355 |
| E.4.7. Spectrum | 357 |
| E.4.8. UniqueData | 358 |
| E.5. Compress | 360 |
| E.5.1. PatternResample | 360 |
| E.5.2. RFResample | 363 |
| E.5.3. Resample | 363 |
| E.5.4. Sync Resample | 365 |
| E.5.5. WindowedResample | 367 |
| E.6. Crypto | 368 |
| E.6.1. AES Advanced Analysis | 368 |
| E.6.2. AES DFA | 371 |
| E.6.3. AES Known Key Analysis | 373 |
| E.6.4. AES Second Order Analysis | 374 |
| E.6.5. DES Advanced Analysis | 377 |
| E.6.6. DES DFA | 381 |
| E.6.7. DES Known Key Analysis | 384 |
| E.6.8. DES Masked Input Analysis | 387 |
| E.6.9. DES Partly Constant Analysis | 388 |
| E.6.10. DES Second Order Analysis | 390 |
| E.6.11. ECCByteMultiplyAnalysis | 393 |
| E.6.12. ECNRPartialNonceAnalysis | 395 |
| E.6.13. RsaBinExpoAnalysis | 397 |
| E.6.14. RSA CRT DFA | 400 |
| E.6.15. RsaCorrelation | 401 |
| E.6.16. RsaCrtAnalysis | 403 |
| E.6.17. RSA High Order Analysis | 405 |
| E.6.18. RsaNeighborCorrelation | 410 |
| E.6.19. SEED Analysis | 411 |
| E.7. Crypto2 | 413 |
| E.7.1. Calculator | 413 |

| | |
|--|-----|
| E.7.2. First Order Analysis | 413 |
| E.7.3. AES Chosen Input First Order Analysis | 415 |
| E.7.4. Known Key Correlation | 418 |
| E.7.5. Simulator | 419 |
| E.7.6. Template Analysis | 420 |
| E.7.7. Verify | 422 |
| E.7.8. Known Key Analysis | 422 |
| E.7.9. AES Chosen Input Known Key Analysis | 423 |
| E.7.10. Points Of Interest Selection | 425 |
| E.8. Edit | 426 |
| E.8.1. CriExport | 426 |
| E.8.2. CrilImport | 427 |
| E.8.3. DataEdit | 428 |
| E.8.4. DataSort | 429 |
| E.8.5. Export | 430 |
| E.8.6. Import | 431 |
| E.8.7. MatExport | 432 |
| E.8.8. MatImport | 435 |
| E.8.9. Recover | 436 |
| E.8.10. Reverse | 437 |
| E.8.11. SampleEdit | 437 |
| E.8.12. Select | 439 |
| E.8.13. Split | 439 |
| E.8.14. TraceMix | 440 |
| E.8.15. Transpose | 441 |
| E.8.16. Trim | 441 |
| E.9. Filter | 441 |
| E.9.1. Abs | 441 |
| E.9.2. Binary | 442 |
| E.9.3. Chain | 443 |
| E.9.4. FilterGuidance | 444 |
| E.9.5. Frequency Band Decomposition | 446 |
| E.9.6. Harmonics | 447 |
| E.9.7. InvNotchFilter | 448 |
| E.9.8. LowPass | 449 |
| E.9.9. MovingAverage | 449 |
| E.9.10. Negate | 450 |
| E.9.11. PatternPad | 451 |
| E.9.12. PeakExtract | 452 |
| E.9.13. SegmentedChain | 453 |
| E.9.14. Spectral | 454 |
| E.9.15. Statistical Correction | 455 |
| E.9.16. XtalClear | 457 |
| E.10. icWaves | 459 |
| E.10.1. Simulate | 459 |
| E.11. Perturbation | 460 |
| E.11.1. Perturbation Module | 460 |
| E.11.2. SmartCardOpticalPerturbation | 465 |
| E.11.3. SmartCardPerturbation | 467 |
| E.11.4. TriggeredOpticalPerturbation | 470 |
| E.11.5. TriggeredPerturbation | 470 |
| E.12. Perturbation2 | 471 |
| E.12.1. SC Perturbation | 471 |
| E.12.2. SC Single XYZ Perturbation | 476 |

| | |
|---|-----|
| E.12.3. SC Perturbation with Glitch Program | 481 |
| E.12.4. Perturbation Advanced Program | 483 |
| E.12.5. SC Perturbation after Reset | 485 |
| E.12.6. EP Perturbation after Reset | 485 |
| E.13. Sort | 487 |
| E.13.1. TraceSort | 488 |
| E.14. Statistics | 489 |
| E.14.1. Average | 489 |
| E.14.2. Distribution | 490 |
| E.14.3. Notify | 491 |
| E.14.4. StandardDeviation | 492 |
| E.15. Configuration | 493 |
| E.15.1. icWavesConfiguration | 493 |
| E.15.2. splitterConfiguration | 494 |
| E.15.3. Training Card configuration | 495 |
| E.16. XYZ | 497 |
| E.16.1. AveragePlot | 497 |
| E.16.2. SpectralIntensity | 498 |
| F. Hardware Drivers | 501 |
| F.1. I/O Devices | 501 |
| F.1.1. PC/SC reader | 501 |
| F.1.2. Power Tracer | 501 |
| F.1.3. Protocol Device | 502 |
| F.2. Raw In/Output devices | 503 |
| F.2.1. Serial Port | 503 |
| F.2.2. Serial card readers (e.g. SASEBO-W) | 504 |
| G. Error Codes | 506 |
| G.1. E03629 | 506 |
| H. Additional API information | 507 |
| H.1. VC Glitcher API | 507 |
| H.1.1. Instruction set | 508 |
| I. Transport Protocols | 520 |
| I.1. ISO/IEC 7816 | 520 |
| I.1.1. Introduction | 520 |
| I.1.2. Implementation details | 520 |
| I.1.3. Configuration | 520 |
| I.2. RiscureLV | 522 |
| I.2.1. Command packet | 523 |
| I.2.2. Response packet | 523 |
| I.2.3. Example flow | 523 |
| J. Application Protocols | 524 |
| J.1. EMV | 524 |
| J.2. GlobalPlatform SCP | 524 |
| J.3. MRTD | 525 |
| J.3.1. Active Authentication | 525 |
| J.3.2. Extended Access Control | 525 |
| J.4. OTA (SCP80) | 526 |
| J.5. Training card protocols | 526 |
| J.6. SIM/USIM Authentication | 527 |
| J.7. Embedded Protocol | 527 |
| J.7.1. Specification | 527 |
| J.7.2. Application protocol development | 532 |
| K. Cipher Suites | 535 |
| K.1. AES | 535 |

| | |
|--------------------------------|-----|
| K.1.1. Analysis steps | 535 |
| K.1.2. Background | 537 |
| K.2. ARIA | 537 |
| K.2.1. Analysis steps | 540 |
| K.2.2. Output | 540 |
| K.2.3. Validation | 544 |
| K.3. Camellia | 546 |
| K.3.1. Analysis steps | 547 |
| K.3.2. Output | 547 |
| K.3.3. Validation | 550 |
| K.4. DES | 551 |
| K.4.1. Analysis steps | 552 |
| K.4.2. Output | 553 |
| K.5. DSA | 554 |
| K.5.1. Analysis steps | 554 |
| K.5.2. Output | 555 |
| K.6. ECDSA | 556 |
| K.6.1. Analysis steps | 556 |
| K.7. HMAC | 556 |
| K.7.1. Analysis steps | 557 |
| K.7.2. Analysis settings | 558 |
| K.7.3. Output | 558 |
| K.7.4. Validation | 560 |
| K.8. RSA CRT | 560 |
| K.8.1. Analysis steps | 561 |
| K.8.2. Output | 562 |
| L. Trace set coding | 564 |
| Index | 567 |

List of Figures

| | |
|--|----|
| 2.1. Inspector GUI | 8 |
| 2.2. The settings dialog | 12 |
| 2.3. The Paths dialog | 13 |
| 2.4. The Other settings dialog | 14 |
| 2.5. The Defaults settings dialog | 15 |
| 2.6. The confirmation dialog for unsaved files when Inspector is exiting | 16 |
| 2.7. The Tweak settings dialog | 16 |
| 2.8. The Tweak warning dialog | 17 |
| 2.9. The Inspector "Profiles" menu and the effect of a profile on the Inspector menu bar.... | 19 |
| 2.10. The Inspector profile editor | 20 |
| 2.11. Performance comparison for Spectrum | 27 |
| 2.12. Performance comparison for long processing chain | 27 |
| 3.1. The cipher selection panel | 31 |
| 3.2. The direction panel | 31 |
| 3.3. A settings panel | 32 |
| 3.4. The (Camellia) Normal View | 35 |
| 3.5. The (Camellia) Expert View | 36 |
| 3.6. The Key Intermediates View | 38 |
| 4.1. Measurement setup with the Power Tracer | 41 |
| 4.2. Schematic of a typical RFA measurement setup | 42 |
| 4.3. Setup for EM measurements | 43 |
| 4.4. EM Probe Station with Power Tracer. | 43 |
| 4.5. Oscilloscope dialog | 44 |
| 4.6. Side Channel Acquisition dialog | 45 |
| 4.7. ScopeAcquisition architecture | 46 |
| 5.1. Configuration A | 51 |
| 5.2. Configuration B | 51 |
| 5.3. Configuration C | 52 |
| 5.4. Configuration D | 53 |
| 5.5. Perturbation module vs. perturbation program | 55 |
| 5.6. The Tango and chip coordinate systems | 60 |
| 5.7. The scan area within both reference frames | 61 |
| 5.8. Camera tab | 63 |
| 5.9. XYZ tab | 65 |
| 6.1. Hardware Manager dialog | 67 |
| 6.2. Example device settings for serial ports | 68 |
| 6.3. Add devices | 69 |
| 6.4. DHCP configuration on the oscilloscope | 70 |
| 6.5. The IP address on the oscilloscope | 71 |
| 6.6. Manual IP settings | 72 |
| 6.7. Hardware Manager: add new device | 73 |
| 6.8. Hardware Manager: device overview | 74 |
| 6.9. Acquisition2: Scope acquisition with LeCroy | 75 |
| 6.10. The Power Tracer 3 | 79 |
| 6.11. The Power Tracer 4 | 83 |
| 6.12. Displayed parameters for smart card channel | 84 |
| 6.13. Variations of the charging status symbol | 84 |
| 6.14. Displayed parameters for serial channel | 85 |
| 6.15. Formula to calculate the voltage level V_{out} on the Power signal port of Power Tracer 4 | 86 |
| 6.16. The Current Probe and its amplifier | 87 |

| | |
|---|-----|
| 6.17. Circuit diagram of current probe and connections | 88 |
| 6.18. the ElectroMetrics probe and the Inspector probe | 89 |
| 6.19. Time signal of ElectroMetrics probe (top) and Inspector probe LS (bottom) with equal magnetic field. | 89 |
| 6.20. Frequency spectra of ElectroMetrics probe (top) and Inspector probe LS (bottom) with equal magnetic field. | 90 |
| 6.21. Adding an EM Probe Station to the Hardware Manager | 90 |
| 6.22. 3D view of RF reader antenna (purple), chip circuit (orange), EM probe and shielding device (pink) | 91 |
| 6.23. 2D view of RF reader antenna (purple), chip circuit (orange), EM probe and shielding device (pink) | 91 |
| 6.24. Shielding deflects field by RF reader antenna; unshielded (left) and with probe shield (right) | 92 |
| 6.25. Magnetic flux of the chip field; unshielded (left) and with probe shield (right) | 92 |
| 6.26. The MicroPross MP300 with TCL1 resource | 93 |
| 6.27. Configuration Dialog for MP300 | 94 |
| 6.28. The RF Tracer | 96 |
| 6.29. connectors on the RF Tracer | 97 |
| 6.30. The VC Glitcher | 99 |
| 6.31. VC Glitcher block diagram | 99 |
| 6.32. New VC glitcher connectors | 100 |
| 6.33. The Diode Laser Station | 104 |
| 6.34. The Splitter | 110 |
| 6.35. The EM-FI probe tips | 112 |
| 6.36. EM-FI probe behavior when glitch pulse width is 50ns | 113 |
| 6.37. The icWaves | 114 |
| 6.38. Conceptual overview of the icWaves | 115 |
| 6.39. Sum of Absolute Differences illustrated | 115 |
| 6.40. Hardware SAD implementation | 116 |
| 6.41. Signal demodulation | 117 |
| 6.42. Mixer design | 117 |
| 6.43. Random noise uniformly distributed in 2-4 MHz | 117 |
| 6.44. Example signal after frequency translation | 118 |
| 6.45. Example signal after frequency translation and low pass filter | 118 |
| 6.46. Trigger settings | 118 |
| 6.47. LCD display | 120 |
| 6.48. icWaves filter properties | 120 |
| 6.49. icWaves configuration dialog | 121 |
| 6.50. Installation dialog | 122 |
| 6.51. Installation dialog | 123 |
| 6.52. Installation dialog | 123 |
| 6.53. icWaves firmware upgrade tool | 124 |
| 6.54. The CleanWave | 130 |
| 6.55. Connecting the CleanWave | 131 |
| 6.56. Connecting the CleanWave (continued) | 132 |
| 6.57. Hardware filter | 133 |
| 6.58. Filter characteristic of 25 MHz high-pass filter | 134 |
| 6.59. Main dialog of the Target Manager | 135 |
| 6.60. Jumper positions for application (left, default) and bootloader mode (right) | 137 |
| 7.1. Misalignment in side channel analysis | 139 |
| 7.2. Acquiring a reference signal with icWaves | 141 |
| 7.3. Acquiring a reference trace | 142 |
| 7.4. icWaves configuration module | 143 |
| 7.5. icWaves configuration window | 144 |

| | |
|--|-----|
| 7.6. Triggering on Trigger out | 144 |
| 7.7. Example power trace and average spectrum | 145 |
| 7.8. icWaves using tunable filter | 145 |
| 7.9. Filtered traces | 146 |
| 8.1. Inspector plug-in overview | 148 |
| 8.2. Cipher list in the Calculator module. | 155 |
| 8.3. Cipher and attacks panels in FirstOrderAnalysis module UI. | 156 |
| 8.4. Options panel of DES cipher in FirstOrderAnalysis module UI. | 160 |
| 8.5. DES cipher in the FirstOrderAnalysis module. | 162 |
| 8.6. The "Attack" panel for DES in FirstOrderAnalysis module. | 167 |
| 8.7. Eclipse new JRE dialog | 168 |
| 8.8. Eclipse new Java Project dialog | 169 |
| 8.9. Eclipse new Java Project dialog. Class path settings | 170 |
| 8.10. Eclipse new Run configuration dialog | 171 |
| 8.11. Eclipse new Run configuration dialog. Arguments settings | 172 |
| 8.12. Eclipse new Run configuration dialog. Classpath settings | 173 |
| 8.13. Eclipse new Run configuration dialog. Environment settings | 174 |
| 8.14. Timing diagram of a typical test run. | 181 |
| 8.15. The MultiScopeSetup implementation of MeasurementSetup selected in ScopeAcquisition | 190 |
| 8.16. The TriggeredProtocolTarget implementation of the Target interface selected in ScopeAcquisition | 200 |
| 8.17. Sequence diagram of the interaction between the framework, target and measurement setup | 202 |
| 9.1. Displaying numbers as text | 217 |
| 9.2. Example of a tool tip | 217 |
| 9.3. Displaying numbers as spinners | 218 |
| 9.4. Displaying numbers as sliders | 219 |
| 9.5. Displaying numbers in different base notations | 221 |
| 9.6. Displaying ranges as range sliders | 222 |
| 9.7. Displaying an Enum as a drop down list. | 223 |
| 9.8. Displaying a drop down list in the AGG | 225 |
| 9.9. A dynamic update of the drop down list in the AGG when compared to Figure 9.8, "Displaying a drop down list in the AGG" | 225 |
| 9.10. Displaying an Enum as a drop down list. | 226 |
| 9.11. Using the DynamicEnum interface, equivalent to Figure 9.7, "Displaying an Enum as a drop down list." | 226 |
| 9.12. An example of an AGG generated checkbox | 227 |
| 9.13. Examples of AGG generated reorderable CRUD views | 228 |
| 9.14. An example of a stacked list and a tabbed view of the same property | 230 |
| 9.15. An example of a reorderable stacked list | 231 |
| 9.16. Displaying a File and InputStream property | 232 |
| 9.17. JPanel filtered on @Tunable | 240 |
| 9.18. The Hardware Manager dialog | 242 |
| 9.19. The newly added static device in the hardware manager dialog | 243 |
| 9.20. Settings presented to the end-user to allow him to change the name and description of the static device | 245 |
| 9.21. Settings presented to the end-user to allow him to change the name and description of the static device | 245 |
| 9.22. The driver listing following a press of the "Add device..."-button in the Hardware Manager dialog | 252 |
| 9.23. The settings bean displayed to the user after making a selection | 253 |
| 9.24. Configuring a scope in the acquisition2 framework | 261 |

| | |
|--|-----|
| 9.25. Entering the descriptors in the Protocol Wizard | 268 |
| 9.26. Entering the command and response lengths in the Protocol Wizard | 268 |
| 9.27. The acquisition2 framework displaying protocol settings | 272 |
| E.1. Measurement tab of XYAcquisition module | 332 |
| E.2. Target tab of XYAcquisition module | 333 |
| E.3. AES decryption with all-random input data | 417 |
| E.4. AES decryption with partly-constant input data | 417 |
| E.5. FilterGuidance dialog with PROCESS phase settings | 444 |
| E.6. DES perturbation module user interface | 461 |
| E.7. Standard perturbation parameters | 462 |
| E.8. The effects of glitch voltages on a clock cycle | 463 |
| E.9. The effects of glitch voltages on the VCC | 464 |
| E.10. The VC Glitcher report window | 464 |
| E.11. Glitch tab of Perturbation module | 472 |
| E.12. Perturbation tab of Perturbation module | 474 |
| E.13. Target tab of Perturbation module | 476 |
| E.14. XYZ tab of Dual XYZ Perturbation module | 478 |
| E.15. Camera tab of Dual XYZ Perturbation module | 480 |
| E.16. Perturbation tab of Dual XYZ Perturbation module | 481 |
| E.17. Custom Glitch tab of the Snippet Perturbation module | 482 |
| E.18. Target tab of Perturbation Advanced Program module | 484 |
| E.19. Glitch tab of Raw Embedded Perturbation module | 486 |
| E.20. Target tab of Raw Embedded Perturbation module | 487 |
| F.1. Power Tracer properties | 502 |
| F.2. Power Tracer pulse properties | 502 |
| F.3. Protocol device dialog | 503 |
| F.4. Serial port properties | 504 |
| I.1. ISO 7816 Properties | 521 |
| I.2. ISO 7816 Specific protocol | 522 |
| J.1. EAC Chip Authentication | 526 |
| K.1. ARIA encryption and decryption processes | 538 |
| K.2. ARIA substitution layer, type 1 | 538 |
| K.3. ARIA substitution layer, type 2 | 538 |
| K.4. Initialization for ARIA key expansion | 539 |
| K.5. Camellia encryption with 128 bit keys (18 rounds) | 546 |
| K.6. Camellia's Feistel function | 547 |
| K.7. The Feistel structure of DES | 552 |
| K.8. SHA1-HMAC overview | 557 |
| L.1. Trace set example | 566 |

List of Tables

| | |
|---|-----|
| 3.1. | 38 |
| 5.1. Calculated scan positions | 61 |
| 6.1. Probe specs in comparison with calibrated Electrometric probe | 89 |
| 6.2. Trigger configuration | 93 |
| 6.3. VC glitcher connectors | 100 |
| 6.4. Voltage specifications | 102 |
| 6.5. EM-FI probe tip specifications | 112 |
| 8.1. Creating a new parameter | 176 |
| 8.2. Smart card clock frequency | 177 |
| 8.3. Methods in the Pattern class | 177 |
| 8.4. Examples of setting the glitch pattern | 178 |
| 8.5. Methods in the Util class | 179 |
| 8.6. Possible return values of the filter method | 179 |
| 8.7. | 184 |
| 8.8. Required annotations | 196 |
| 9.1. JavaBean accessor naming conventions | 204 |
| 9.2. JavaBean mutator naming conventions | 205 |
| 9.3. JavaBean backing field naming conventions | 205 |
| 9.4. BeanPanelUtils.getBeanPanel(...) argument overview | 205 |
| 9.5. An overview of annotations used by the AGG for the purpose of validation | 210 |
| 9.6. @Programmatic method naming conventions | 238 |
| 9.7. HardwareUtils.registerDevice(...) argument overview | 241 |
| 9.8. An overview of device classes and their corresponding session type. | 255 |
| 9.9. The properties that can be configured on a channel via the OscilloscopeProperties interface | 258 |
| 9.10. The properties that can be read and/or configured on a channel via the ChannelProperties interface. | 259 |
| 9.11. The properties that can be read and/or configured to trigger a measurement commencement via the TriggerProperties interface | 260 |
| 9.12. ProtocolTarget.send(...) argument overview | 273 |
| C.1. Keyboard shortcuts | 316 |
| D.1. Parameters | 317 |
| E.1. Parameter combinations | 462 |
| J.1. Algorithm field values | 529 |
| J.2. Algorithm mode field values | 530 |
| J.3. MESSAGE_TYPE field values | 530 |
| J.4. Response code values | 531 |
| J.5. Frame tags | 531 |
| J.6. Function tags | 531 |
| J.7. Parameter tags | 531 |
| K.1. ARIA round key generation. | 539 |
| L.1. Trace set objects in the trace set header | 565 |
| L.2. Sample coding | 566 |

List of Examples

| | |
|--|-----|
| 8.1. The perturbation parameter "Glitch voltage" in GlitchSetupSettings | 190 |
| 9.1. The initial bean class that does not yet implement the Model part of the MVC pattern | 206 |
| 9.2. Enabling property change notifications as part of the Model implementation of the MVC pattern | 207 |
| 9.3. The standard fire property change event pattern | 209 |
| 9.4. Firing a property change event for a property when the old and new values are unknown | 209 |
| 9.5. Firing a property change event when multiple properties changed in the containing object | 209 |
| 9.6. Bean property constraints through annotations, with a custom error message | 211 |
| 9.7. Bean property constraints through custom validation logic | 212 |
| 9.8. A JavaBean with unconstrained properties | 213 |
| 9.9. Obtaining a list of all invalid end-user input | 216 |
| 9.10. The bean displayed by the AGG in the figures below | 217 |
| 9.11. bean code demonstrating spinners | 218 |
| 9.12. Bean code demonstrating sliders | 219 |
| 9.13. Three examples of the truncated number property | 221 |
| 9.14. Bean code demonstrating range sliders | 222 |
| 9.15. Bean code demonstrating enum as drop down list | 223 |
| 9.16. ComboBoxItem code used in next example | 224 |
| 9.17. Bean code demonstrating combobox with complex objects | 224 |
| 9.18. Bean code demonstrating dynamically updated drop-down list | 226 |
| 9.19. Bean code demonstrating a check box | 227 |
| 9.20. Bean code demonstrating a reorderable arbitrary number of CRUD views | 227 |
| 9.21. The class that will be shown as a nested bean by the examples in this section | 229 |
| 9.22. Bean code demonstrating stacked list and tabbed interface of same property objects | 230 |
| 9.23. Bean code demonstrating a reorderable stacked list | 231 |
| 9.24. Bean code demonstrating buttons that open a file dialog | 232 |
| 9.25. Toggling expert properties on an AGG generated view | 233 |
| 9.26. An example of two properties making use of @Applicable | 234 |
| 9.27. Showing and hiding properties using property change support and custom logic | 235 |
| 9.28. An alternative way of using @Applicable with a read-only boolean property | 236 |
| 9.29. Implementing an annotation interface with the aid of AnnotationLiteral | 237 |
| 9.30. Usages of the @Programmatic annotation | 238 |
| 9.31. Annotate properties with @Tunable | 239 |
| 9.32. GUI panel with @Tunable properties | 240 |
| 9.33. An example of the org.osgi.framework.BundleActivator file, indicating that two activators are present in the JAR | 242 |
| 9.34. Registering a static device in Inspector's HWM | 243 |
| 9.35. Skeleton implementation of DeviceInterface.open(Object prefs) | 256 |
| 9.36. Skeleton implementation of Session | 257 |
| 9.37. The recommend pattern to follow when implementing get*Properties() methods | 262 |

Preface

Note:

This document is available in HTML and PDF format. However, standalone PDF readers cannot resolve relative URLs. Therefore, if you are viewing this document with a standalone PDF reader, links to external files distributed with this document will not work. In some cases a [PDF] link is provided with the HTML link. This link opens the .PDF file of the external document. To jump to the tutorials PDF document use this link [../tutorials/Tutorials.pdf].

1 Introduction

This introductory chapter introduces Inspector and describes its main functions. It also gives a short overview of the content of this manual document and its structure.

1.1 Overview

Side channel analysis is a method for studying the behaviour of electronic devices. Rather than using a regular I/O interface, physical phenomena like timing, power consumption and electromagnetic emissions are used. The analysis is often applied to cryptographic devices in order to investigate the leakage of secret data such as keys and PIN codes. The results are used to identify weaknesses in the implementation of hardware and software. Side channel analysis is an important instrument in the product development and evaluation cycle of smart cards and other memory devices, and contributes to achieving a high level of security.

Inspector is a side channel analysis tool for system developers and evaluators. Its main features are:

- Browsing and visual inspection of acquired samples
- Signal processing capabilities for filtering and alignment
- Signal analysis to investigate signal properties and information leakage
- Key retrieval for various algorithms
- Acquisition support for controlling oscilloscopes and external devices
- Controlling perturbation devices for fault injection.

The appropriate type of signal analysis depends on the application under investigation. For instance, each cryptographic algorithm requires a specific approach. The standard Inspector release includes several general-purpose modules for statistical analysis, filtering and trace alignment, as well as some key retrieval methods for popular crypto algorithms. Additional dedicated modules for advanced analysis are separately available or can be custom-developed by Riscure. Users are also encouraged to implement their own analysis techniques.

Inspector implements an open signal coding format that allows a user to analyze files generated with his own data acquisition equipment. Within this format, information is kept on the actual samples as well as a descriptive text and supplementary cryptographic data.

New users of Inspector are advised to read the help files. Especially the examples and exercises in the Tutorials document are helpful in gaining a good understanding of the tool's capabilities. Riscure encourages users to develop their own analysis modules, but is also prepared to assist in development. All questions and requests can be addressed to inspectorsupport@riscure.com [mailto:inspectorsupport@riscure.com]. If you have any bugs to reports please use the logging feature described in the settings section. **Frequently Asked Questions** (FAQs) are collected and presented on the website here [<http://www.riscure.com/tools/inspector/support>]. This can be a useful information source for beginners as well as more experienced users.

Users will be notified of future software updates of Inspector. Further, as an Inspector user, you are invited to try out the free online Riscure developer tools. These are also available on our website [<http://www.riscure.com>].

1.2 Contents

This manual describes Riscure's complete Inspector platform, including software and hardware components and the manual structured in the following way:

- In the second chapter *Inspector Software* the Inspector software core is discussed. It describes the user interface and the architecture of the software, also general configuration and specifications of the software are detailed.
- Chapter 4, *Side Channel Acquisition* describes how to perform Side Channel Acquisition on your TOE.
- Chapter 5, *Fault Injection* describes how to perform Fault Injection attacks on your TOE.
- Chapter 7, *Real-Time pattern matching* describes how to use the icWaves to optimize triggering.
- Chapter 6, *Inspector Hardware Components* describes the various components that are supported by Inspector and how they are used.
- Chapter 8, *Software Development for Inspector* describes how to add new functionality to Inspector by developing and adding software components.

The remainder of the manual consists of appendices that serve as a reference guide for inspector components and features, such as keyboard shortcuts, modules, drivers and protocols.

Other appendices provide information on third party licenses, release notes, and references for specific hardware.

- Appendix A, *Third party software licenses* contains the exact licenses under which the the third party software is used
- Appendix B, *Release notes* contains the release notes of the previous Inspector releases and updates.
- Appendix C, *Keyboard Shortcuts* contains a table of popular keyboard shortcuts available in Inspector.
- Appendix D, *MP300 Device driver parameters* contains a table of parameters for the Acquisition framework driver of the MP300 device.
- Appendix E, *Modules List* is a reference for all modules in Inspector. Here you can find detailed documentation on how to use each module, as well as background information for complex modules.
- Appendix F, *Hardware Drivers* describes the available hardware drivers for the Acquisition2 framework. inspector.
- Appendix H, *Additional API information* describes additional APIs included in Inspector
- Appendix I, *Transport Protocols* describes the available transport protocols for the Acquisition2 framework.
- Appendix J, *Application Protocols* describes the available application protocols for the Acquisition2 framework.

- Appendix K, *Cipher Suites* contains a listing of the Cipher Suites provided in Inspector.
- Appendix L, *Trace set coding* describes the Inspector traceset file format.

1.3 Glossary

Side channel attacks have a lot of very specific terminology. To make the terminology used in this manual clearer we include a glossary of terms here.

| | |
|----------------------|---|
| API | Application Programming Interface |
| Application protocol | Communication protocol capable of commanding the target of evaluation to perform the desired operation. Example: EMV, USIM, MRTD |
| Camera | Still picture camera capable of creating a snapshot image on demand. Example: uEye, SXVR |
| Cipher | Algorithm for performing encryption, decryption and/or signing with intermediate values and including one or more leakage models Example: AES, DES, RSA, ECC |
| Consumer | Module which takes traces as input but does not produce traces as output. The output consists of other information besides traces. Example: crypto analysis, export, spectral intensity |
| CPA | Correlation Power Analysis |
| CRI | Cryptography Research Incorporated; the data format called DPAWS of their tool can be imported and exported in Inspector. |
| DFA | Differential Fault Analysis |
| DoM | Difference of Means. Type of DPA which divides the measurements into two groups depending on whether they are above or below the mean. |
| DPA | Differential Power Analysis. Class of attacks performing differential analysis on measured leakage, including CPA and DoM. |
| DPA (Classic) | See DoM |
| Driver | Plug-in that is able to control a device. A driver can be static (i.e. the device always exists), plug-and-play (i.e. the device is created automatically when plugged in), or user created (i.e. a device exists once the user configures it). |

| | |
|-----------------------|--|
| Execution | An execution of a cipher performs the encryption/decryption operation from start to finish. |
| | Triple DES consists of 3 DES executions. |
| Execution key | Secret input of a cipher execution |
| | Example: Triple DES has 3 execution keys typically named K1, K2, and K3 |
| FI | Fault Injection |
| Filter | Module which produces a trace for every trace it consumes and manipulates the consumed trace in order to create the produced trace. |
| | Example: Abs |
| Full key | See key |
| Glitch cycle | Clock cycle in which the VC Glitcher exerts a glitch pattern. |
| Glitch fragment | Set of zero or more wait cycles followed by one or more glitch cycles. |
| Glitch pattern | A pattern which describes the glitch(es) performed. For the VC Glitcher this is a sequence of bits indicating whether a glitch should be performed during a 2 nanosecond time slot. For the embedded glitcher this is a series of timings which alternating will perform a glitch or will not. |
| Glitch sequence | Either a set of one or more glitch fragments (VC Glitcher) or a glitch pattern (Embedded Glitcher). |
| HD | Hamming Distance; the number of bits that differ between two bit strings. |
| HW | Hamming Weight; the total number of one bits (i.e. set to "true") in a bit string. |
| I/O device | Communication device capable of transmitting a variable length block of bytes and receiving a response block of bytes. Generally an underlying transport protocol governs the transmission and receiving of the blocks and may guarantee fault tolerance and sequencing among others. |
| | Example: Power Tracer, PC/SC reader |
| Input generator | Generator for input data to crypto protocols. Generated input generally has a fixed length depending on the block size of the cipher used. Generation can be random or specific depending on the requirements of the crypto analysis. |
| Intermediate value | In cryptographic context: Value of an intermediate variable given a set of sub keys and input |
| Intermediate variable | In cryptographic context: Name of the input/output of a single operation in an algorithm |

| | |
|------------------------|--|
| IP | Intellectual Property or Internet Protocol |
| Key | Unique secret number provided to cipher algorithms to perform an encryption or decryption. Not to be confused with round key or sub key. In the case of triple DES the key refers to the secret used for the entire process. |
| Key (Master) | See key |
| Key (Partial) | Representation of what information we know of a (round) key |
| Key (Sub) | Part of a (round) key that can be targeted independently from a DPA/DFA perspective. |
| Leakage model | Model of any kind of leakage (including power, EM, light, sound) based on one or more intermediate values |
| Measurement setup | Devices which measure and/or influence physical properties of the target of evaluation. Example: Oscilloscope (measure), CleanWave (influence), Laser (influence) |
| Optical perturbation | The process of (physically) disrupting the operation of a chip with the use of a laser beam. |
| Oscilloscope | Device capable of observing a physical quantity during a time period and providing this observation in the form of a numeric time series. |
| Perturbation data | List of perturbation parameters plus the verdict |
| Perturbation parameter | A physical quantity that influences the behavior of a chip, like e.g. glitch length, number of glitches, number of wait cycles. Subset of general parameters which can be varied in order to perform a successful glitch. |
| Plug-in | Program which extends the basic functionality of a platform. Example: Module, Driver |
| POI | Points of Interest; the locations analysed during template analysis. |
| Power model | See leakage model |
| Producer | Module which produces traces without the need for input traces. Example: acquisition, import, crypto simulator |
| Protocol | See application protocol |
| Pulse | A sudden increase/decrease in voltage on a signal line. Usually created to indicate the occurrence of an event. |
| Pulse generator | A device creating a pulse in response to a predefined event. |

| | |
|----------------------|--|
| Raw I/O device | Communication device which is capable of sending and receiving a stream of bytes. Unlike an I/O device a raw I/O device is unaware of the meaning of the bytes in the stream. |
| | Example: Serial port |
| Round | An iteration of a block cipher associated with exactly one round key. The name/number of a round depends on the cipher specification, although usually the first round is referred to as round 1. |
| | Example: AES-128 consists of 11 rounds, the first one referred to as the initial round, the next 9 as rounds 1 through 9 and the last round as the final round. |
| | Example: Triple DES has 3 times 16 rounds, unofficially numbered 1 through 48. |
| Round key | Key that is used at a certain round/iteration of a cipher. |
| SCA | Side Channel Analysis |
| Scope | See Oscilloscope. |
| Search space | In FI context: Space containing all possible combinations of perturbation parameters. |
| Setup | Combination of target and measurement setup. |
| Smart triggering | Starting a process based on a complex series of events. |
| SNR | Signal-to-Noise Ratio |
| Stable power | Power with minimal amount of noise. |
| Table | See XYZ table. |
| Target (acquisition) | Device in direct communication with the target of evaluation. |
| | Example: Power Tracer, Serial port |
| Target (DPA) | Intermediate variable being observed and the sub/round key being targeted. |
| | Example: For AES 'SBox out of round 2, targeting round key 1' |
| TOE | Target of Evaluation: The device under attack or analysis. |
| | Example: Smart card, embedded CPU |
| Trace | A numerical (byte, int, float) series optionally include a title and/or byte series. |
| Trace set | A collection of traces in which the length of the different components of a trace is the same for all traces in the set. A trace set can optionally contain a description, offset, x/y labels and x/y scale. |

| | |
|--------------------|--|
| Transport protocol | Communication protocol which takes a block message and transfers it to a stream device and reads back the response from the stream and returns the response block. Examples: ISO 7816, ISO 14443, HDLC. |
| Trigger | Event which starts a process. Usually used in time critical applications such as acquisitions to get an exact start time. In general a trigger event is a pulse in the voltage on a signal line. |
| Verdict | The execution class that a perturbation run belongs to, like e.g. NORMAL, SUCCESSFUL, RESET, ... |
| Video camera | Device capable of producing a continuous stream of images visualising the target of evaluation. Example: uEye |
| Wait cycle | Clock cycle in which the VC Glitcher does not playback a glitch pattern. |
| XYZ manipulator | A manipulator which moves one or more components of the measurement setup in 3 dimensional space over the target. Example: EMPS, Secondary DLS optical manipulator |
| XYZ table | A platform to which the target is attached is moved in a 3 dimensional space while the component(s) of the measurement setup above remain fixed. Example: Primary DLS table |

2 Inspector Software

Two main parts of the Inspector software are: the Inspector Core and the 'user modifiable' modules. The existing modules are described in Appendix E, *Modules List* and the development of modules is described in Section 8.3.1, "Module development". The core allows for viewing of acquired traces and the execution of the modules. This chapter describes the core of the Inspector software.

2.1 User interface

The Inspector main screen consists of a tool bar, a desktop for various windows, an output area with tabs for standard output and standard error, and a status bar at the bottom.

Figure 2.1. Inspector GUI



The tool bar shows the following buttons:

- **open** a trace set file
- **reopen** trace set file from recent history

-  **select** a trace set window
-  **save** a trace set. The user can also change properties of the trace set before the file is saved (Ctrl+S)
-  **license** certificates. The user may check the validity or update his certificates
-  **settings** of application and trace set
-  **load** and execute a function module
-  **execute** the loaded module on the selected trace set
-  **Pause** the execution of the running module
-  **abort** the current module execution
-  **open** a module source code.
-  **reopen** module source from recent history, or create a new module using the development wizard
-  **select** module source editor from recent history
-  **search** for text in module source code
-  **compile** a Java source file and load it as a function module (F9)
-  **help** files

An open trace set is plotted in a viewer which allows users to manipulate its representation. A trace set can be controlled via buttons in the tool bar, via the scrollbars, key accelerators, and pop up menus.

The tool bar allows the user to control the trace set viewer by means of the following:

- *Lines* the number of lines displayed (accelerators: Ctrl+PageUp, Ctrl+PageDown)
- *Traces* the number of traces displayed (accelerators: Shift+PageUp, Shift+PageDown)
- *Overlap* display multiple traces in one graph instead of separate graphs (applicable only if Traces > 1, accelerator: Shift+A)

The functions of the scrollbars and key accelerators are:

- The scrollbar on the left (or arrow up/down keys) is used to zoom in and out. For small steps, use the arrow keys, holding down the Ctrl key.
- The bottom scrollbar (or arrow left/right keys) is used to browse through a trace. For small steps, use the arrow keys while holding down the Ctrl key.
- The scrollbar on the right (or the page up/down keys) is used to browse through the trace set. Left clicking the mouse in a trace selects the trace and the sample pointed at. The status bar displays the value of the selected sample. Double-click to mark the sample with a cross. Accelerators for browsing up and down the trace set are the PageUp and PageDown keys. Ctrl+Home and Ctrl+End move to the first and last trace.

Dragging (moving the mouse with the left mouse button held down) is used to select part of a trace. Pressing 'Enter' will zoom in to the selection. To undo this, press the Escape or Backspace key. Dragging with the Ctrl key held down is used to shift a trace horizontally, which is helpful when comparing different traces. Use Ctrl+A to toggle the selection of the complete trace.

Right clicking the mouse button brings up pop up menus with the following options:

- *local title* Edit the local title of the trace
- *global title* Edit the global title of the trace
- *goto trace* Display the trace with a given index (Shift+T)
- *delete* Delete the selected trace (Delete) *
- *cut* Cut the selected part of the trace to the clipboard (Ctrl+X) *
- *copy* Copy the selected part of the trace to the clipboard (Ctrl+C)
- *paste* Paste the clipboard contents into the current trace (Ctrl+V) *
- *screen shot* Place an image of the trace set window in the system clipboard. (Shift+S)
- *append* Append a trace set to the current trace set.
- *move label* Move the graph label (click the mouse button while pointing at the label).
- *settings* Change the settings for displaying traces. (Shift+P)

* (only available if the current trace set contains a single trace)

The functions append; screen shot; goto trace and global title are also available under the Edit menu option, when a trace set is selected.

The status bar displays the properties of the trace set:

- The progress of a function or the insertion point position of a module edit window.
- The number of available traces, number of displayed traces, index of the selected trace (selection is performed by clicking one of the displayed traces).
- The number of available samples, number of displayed samples, index of the selected sample (selection is performed by dragging the mouse over the graph).
- The value of the selected sample, or the number of samples a trace is shifted with respect to other traces.
- The supplementary data (byte array) stored along with the trace. This value can be copied to the clipboard by right-clicking on it.

The keyboard shortcuts are listed in the [Keyboard Shortcuts](#) appendix.

2.2 Function modules

The type of signal analysis that you want to apply depends on the application under investigation. For instance, each cryptographic algorithm requires a specific approach. Inspector therefore implements an open analysis interface (i.e. API) that permits you to switch easily between various function modules and even implement your own analysis techniques.

1. Select the trace set on which a function is to be applied. The trace set is automatically selected after opening.

2.

Load the function module. A module is selected via the library module button . This invokes a nested drop down menu. After a successful load, the name of the function module is displayed in the title bar of Inspector. When the module is loaded it is automatically invoked. You are asked to select the number of traces and samples to analyze (note: default is the entire trace set). You may also be prompted for other options specific for the type

of analysis. The dialog parameters can be saved (button ) and reloaded (button

) for convenience.

3. When the analysis is complete, the results are displayed in a new trace set window. Note: the trace set produced is a temporary file (name is preceded with '~') which is deleted when the window is closed. Deletion is prevented by explicitly saving the file.

4.

The module can be invoked again by pressing the run button .

Many function modules have their own help files; you can read these by loading the respective

module and clicking the Help button  on the input data form at execution.



Every function module has a reset button . With this button the default values of the module's parameters can be reloaded. This should be used with caution as it resets the parameters set by the user. Note that not all parameters in a module have default values; only the parameters that have default values are reset by this button.

Inspector comes with a lot of the standard and optional modules for editing, statistics, alignment, filtering, analysis, acquisition, xy-scanning and crypto. These are all described in Appendix E, *Modules List*. The *edit* group includes functions to copy and paste samples and traces, but also to manipulate (crypto) data stored along with the traces.

To practice using a function module, please refer to the second exercise in the tutorial document.

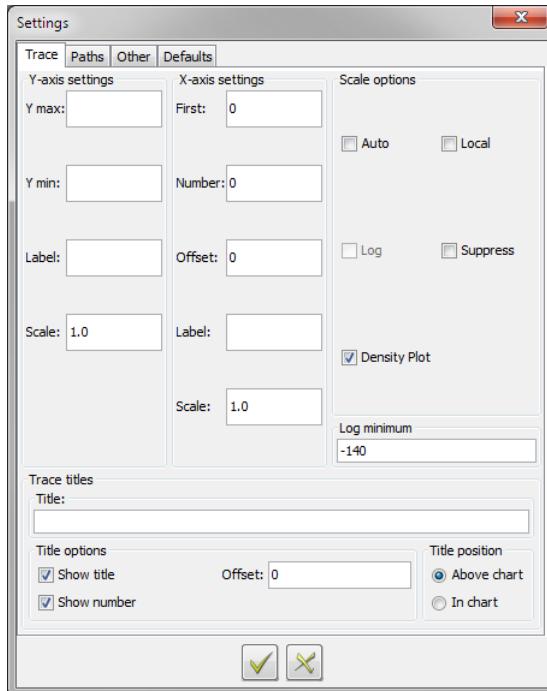
Users that want to develop their own modules should read the Section 8.3.1, “Module development” section.

2.3 Settings

The settings dialog which is available from the "File" menu allows you to view the current global settings and trace specific settings. There are also a number of settings that can be changed. All settings are stored as soon as you click "OK" in the dialog box. The Settings Dialog is split in three categories visible under different tabs; Trace settings, Paths settings and Other settings.

Trace Settings

Figure 2.2. The settings dialog



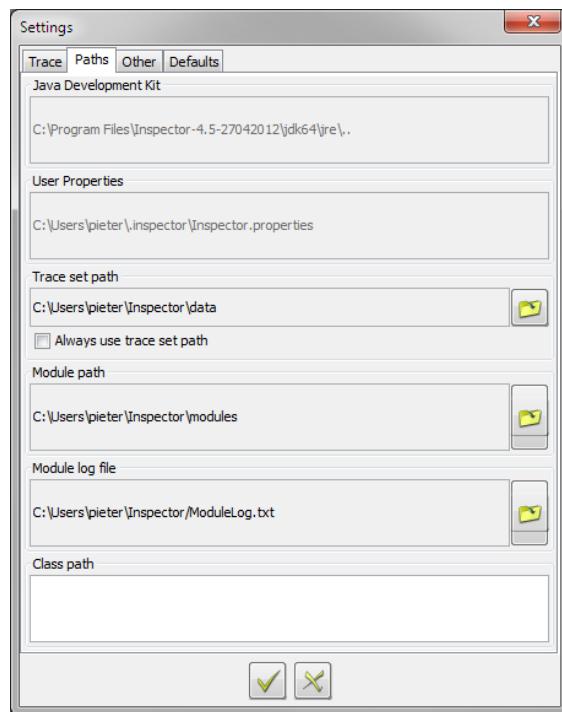
The following Trace Settings are noteworthy:

| | |
|--------------|--|
| Auto | switches automatic vertical trace scaling on or off. |
| Log | tells Inspector to display a trace in Logarithmic scale |
| Local | tells Inspector to use independent individual vertical scales when multiple traces are displayed. This is convenient when two traces have a large difference in amplitude, then both traces are scaled individually. |
| Suppress | tells Inspector to hide scale values and labels, and just display the plain values (Ctrl-u). This is used to switch the X-axis scale from time to samples. |
| Density Plot | tells inspector to use the density plot (slower, but more detail) rather than the min-max plot |

This trace settings dialog can also be called, by right-clicking on the trace and selecting "settings".

Path Settings

Figure 2.3. The Paths dialog



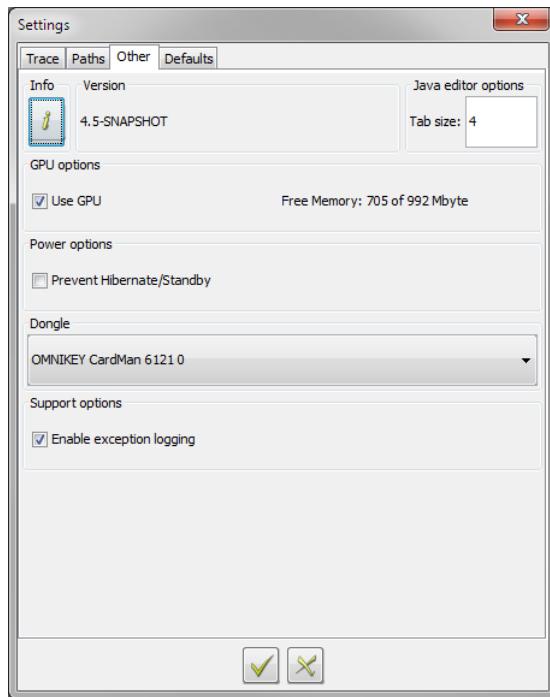
The following Paths Settings are noteworthy:

| | |
|----------------------|---|
| Java Development Kit | You can see which Java Development Kit is used by Inspector. You cannot change this from here. If you want to run with a different development kit, you need to start Inspector from the command line with that specific JDK. |
| User Properties | You can see where the user properties (all settings) are stored. You cannot change this setting as it is determined by the way Inspector is started, see section below. |

| | |
|----------------|---|
| Trace set path | You can change the trace set path, where all your data is stored. This can be changed by browsing to a new existing directory. When the option <i>Always use trace set path</i> is checked Inspector will also store temporary trace files in the given directory, instead of the directory of the original trace set file. |
| Module path | You can change the module path, where all the sources and classes of your modules are stored |
| Log path | You can change the log path, where the log files are stored |
| Class path | You can setup an extra class path to pick up user specific classes and jar files |

Other Settings

Figure 2.4. The Other settings dialog



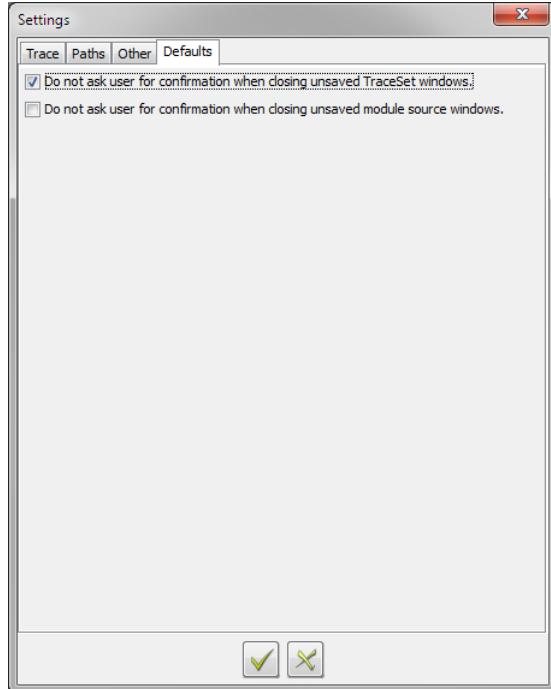
The following Other Settings are noteworthy:

| | |
|---------------------|---|
| Version | show the version number of Inspector |
| Java Editor Options | You can change the tab size used in the editor |
| GPU Options | When a GPU is present in the system, this check box enables its use by Inspector |
| Power Options | You can prevent your computer from Hibernation and Standby while running inspector |
| Dongle | You can select the actual port on which the dongle is found and checked. This is to prevent searching for the dongle on a port that you may want to read/write data to. |

| | |
|-----------------|--|
| Support options | With the Enable exception logging checked, warnings and errors are logged to an exception log. This is particularly useful when reporting bugs. The file is called support-<date>-<pid>.log and is located in the root of the Inspector installation folder. |
|-----------------|--|

Default settings

Figure 2.5. The Defaults settings dialog



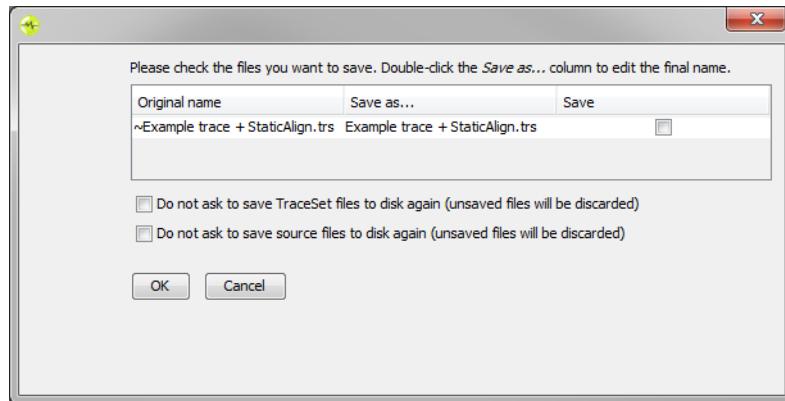
The following settings controlling default behaviour of Inspector in certain situations are available:

- | | |
|---|---|
| Do not ask user for confirmation when closing unsaved TraceSet windows | If selected, Inspector will ignore (and delete) an unsaved trace file when the TraceSet window (or the entire application) is closed. On the other hand, if this option is <i>not</i> selected, Inspector will ask the user whenever an unsaved TraceSet (i.e. a temporary TraceSet) is being closed. If this happens at exit time, all unsaved TraceSets will be shown as a list within a dialog. The dialog shows the original name, the final name the file will be saved as and a checkbox allowing the user to save/discard individual files. The destination filename can be edited by double-clicking on it. |
| Do not ask user for confirmation when closing unsaved module source windows | If selected, Inspector will discard an unsaved module source file when the source window (or the entire application) is closed. Otherwise, if this option is <i>not</i> selected, Inspector will ask the user whenever an unsaved source code window is being closed. In the same way as for unsaved TraceSets, unsaved module source files will be |

shown as a list consolidated with unsaved TraceSets at exit time.

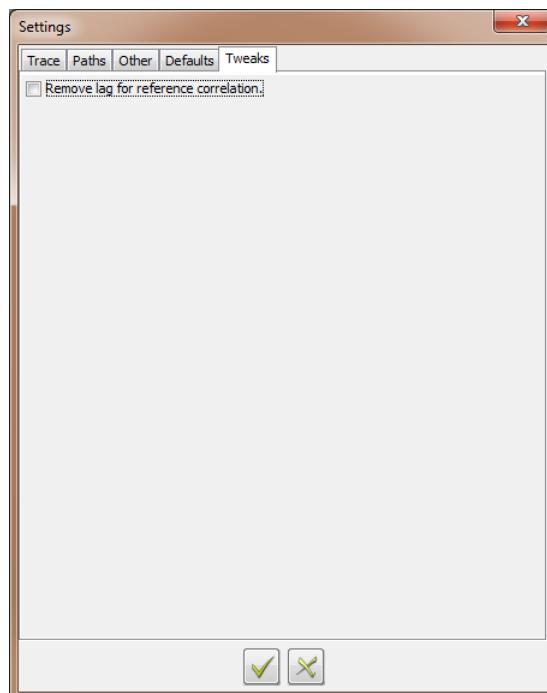
When the user exits Inspector the software will check whether there are unsaved module sources or unsaved TraceSets. Depending on the values of the two 'confirmation' options (described above) a dialog will be presented, if necessary. See Figure 2.6, "The confirmation dialog for unsaved files when Inspector is exiting" for an example.

Figure 2.6. The confirmation dialog for unsaved files when Inspector is exiting



Tweak settings

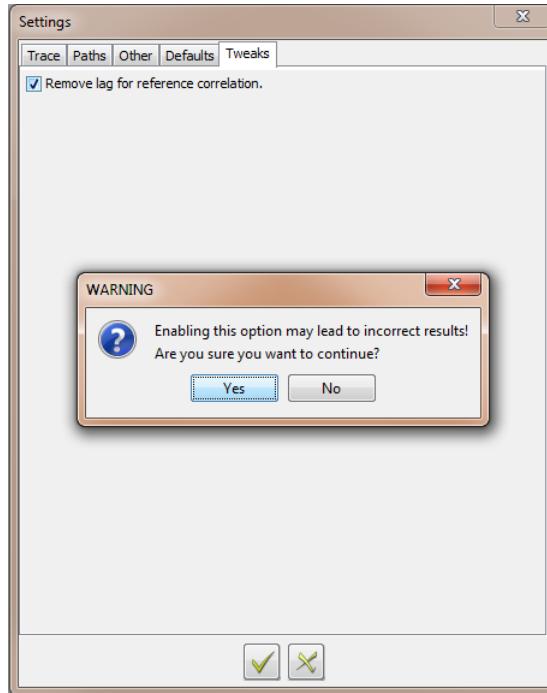
Figure 2.7. The Tweak settings dialog



Selecting *Remove lag for reference correlation*, shown in Figure 2.7, "The Tweak settings dialog", will speed up all calculations that use reference correlation (e.g. static alignment), but

it has a high probability of yielding incorrect results. Riscure strongly discourages using this functionality. To emphasise the dangers involved, selecting this option will prompt a dialog as shown in Figure 2.8, "The Tweak warning dialog".

Figure 2.8. The Tweak warning dialog



Where is all this information stored ?

Inspector keeps a set of properties files around to store the settings of the system, the user and the particular invocation of Inspector. The properties files all have the same format, but are different from previous versions of Inspector.

The system properties file is located where you installed inspector which is normally in:

- C:\Program Files\Inspector-vx.x\Inspector.properties

This file contains the defaults for all properties. You can edit it by hand only and setup installation-wide properties.

The user properties file is always located in:

- %USERPROFILE%\.inspector\Inspector.properties (note the leading dot in the inspector directory).

Any properties in this file will override the one in the system file. Again you can only edit this file by hand and setup user-wide properties.

The invocation file is normally located in the directory from which you start up Inspector. If you start Inspector from the default Inspector icon, or from the Start Menu, this file will be set to:

- %USERPROFILE%\Inspector\Inspector.properties

The properties in this file will override all others. If you change a property in Inspector, it will be written to this file. If you set a property back to the default value (the one set in one of the user/system properties file) it will remove it from this properties file.

If you need to run multiple instances of Inspector, you have two options. Either you want the settings to be the same in both instances, in which case you can just start both Inspectors from the menu or the icon. Settings changed in one Inspector will propagate to the other Inspector.

If you want different settings for each Inspector you need to start Inspector from different directories. To do this you need to create shortcuts to the inspector.exe and set the "Starts In" setting in the shortcuts properties to different directories. This will create different Inspector.properties files in different locations.

2.4 User and System Space

The inspector program, its modules and demo datasets are installed in System space, normally under:

- C:\Program Files\Inspector-v4.7\

User data and modules are stored in User space which is normally located in:

- %USERPROFILE%\Inspector
- %USERPROFILE%\Inspector\data
- %USERPROFILE%\Inspector\modules

It is populated with user modules and your datasets. It is an area that will NOT be touched or updated by the installation of future inspector versions. This allows you to upgrade inspector while you do not have to copy your modules and datasets in again.

If you open a dataset from the system area, or open a source module from the system area and try to save/compile it, it will suggest you to save it in your user area. This makes sense if you wish to modify data or modules.

System modules are located in the normal "Run" menu as it has always been. User modules are located under the "Run"->"User" sub menu, and can be run from there. All modules are available also in the Chain module. Intermediate files (filters and other data) are now all stored in the user area.

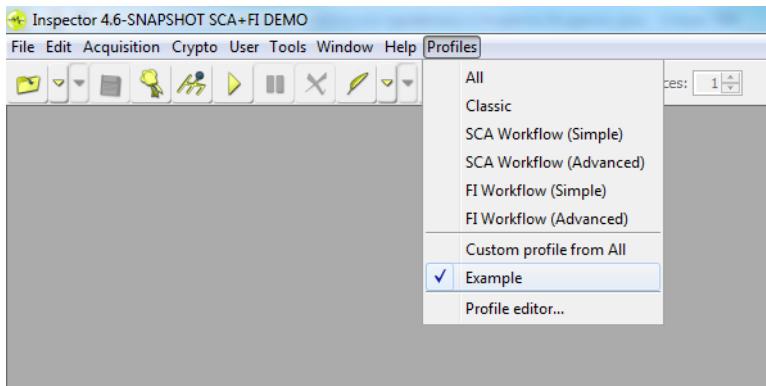
Modules in the System area and the User area may have the same name. Inspector (and Chain) know which module to load, based on if you select it from the System or User area. Your module may use other classes from either System or User area. In fact most Acquisition modules use Chain.

User area has preference over System area. If you load a System module, only the System area is searched for the module and any extra classes. If you load a module from the User area, then for any extra classes the User area is searched first, followed by the System area.

2.5 Profiles

Inspector uses *profiles* to allow dynamic altering of the modules that appear in the Inspector menu bar, as well as where and under what name these modules appear. The active profile can be changed from the "Profiles" menu in the Inspector toolbar. The menu items defined in the selected profile will appear between the "Edit" and "Tools" menus.

Figure 2.9. The Inspector "Profiles" menu and the effect of a profile on the Inspector menu bar

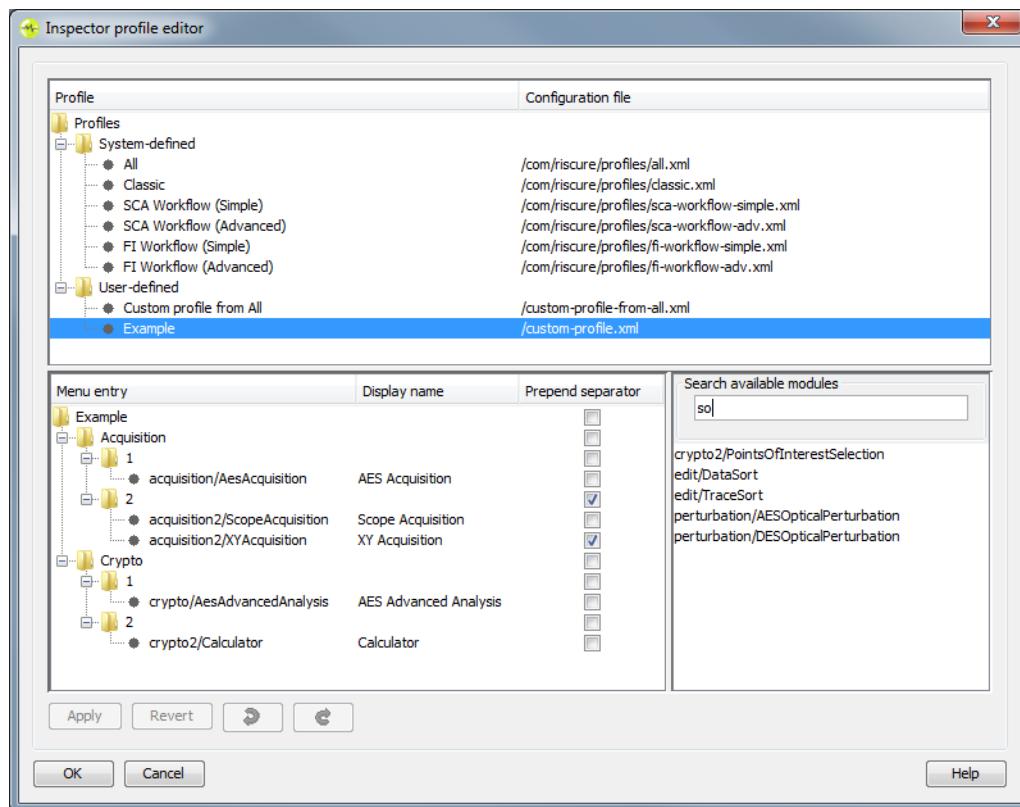


There are two categories of profiles, namely system-defined profiles and user-defined profiles. The "Profiles" menu will first list all system-defined profiles, followed by a separator line, followed by the user-defined profiles. The screenshot shown above shows six system-defined profiles and two user-defined profiles, with the "Example" profile being the active one.

Inspector profile editor

Since Inspector provides a profile editor which can accessed from the "Profiles" menu in the Inspector menu bar. The profile editor, shown below, consists of three sections.

- The upper half (*profiles section*) which displays the profiles recognized by Inspector
- The lower left quarter (*menu section*) which displays the contents of the profile selected in the profiles section. This section is used to alter the selected profile.
- The lower right quarter (*modules section*) which displays the Inspector system modules that are not already present in the current selected profile.

Figure 2.10. The Inspector profile editor

A user can perform the following actions in the profile editor

- *Create a new profile*: This functionality becomes available by right-clicking the "Profiles" or "User-defined" node in the profiles section
- *Duplicate an existing profile*: This can be done by right-clicking any of the profiles in the profiles section and selecting the appropriate option.
- *Select a profile for inspection and/or editing* : This is done by simply left-clicking on any of the profiles in the profiles section. It should be noted that system-defined profiles cannot be edited.
- *Remove a user profile* : This action can be performed after right-clicking on any of the user profiles. This action cannot be undone.
- *Rename a profile, (sub) menu or module display name*: The rename functionality becomes visible after right-clicking the element that needs to be renamed. This action can be undone.
- *Prepend a (sub) menu or module entry with a separator*: Check the checkbox in the "Prepend separator" column to insert a separator before the associated sub menu or module entry. This action can be undone.
- *Insert a new (sub) menu into the current profile*: This functionality appears after right-clicking either the root or any of the (sub) menus on the menu section. This action can be undone.
- *Remove (sub) menus and module entries*: By right-clicking any of the menus and entries on the menu section, the remove functionality becomes available. If multiple items are selected and right-clicked upon, they can be removed at the same time. This action can be undone.

- *Search the available modules*: Typing any text in the "Search available modules" text field will filter the list of available modules shown below it.
- *Insert new modules into the current profile*: Modules are inserted into a (sub) menu by dragging one or more of them from the modules section onto the sub menu in which the modules should be inserted. This action can be undone.
- *Apply, revert, undo and redo actions*: The four buttons directly under the menus section are intended for applying current changes, reverting current changes, undoing the latest change, and redoing the latest undone change respectively. These buttons are enabled whenever it becomes possible to perform the corresponding action.

Profile location and format

System-defined profiles are embedded in the Inspector program and cannot be edited by the user. User-defined profiles reside in the user space under:

- %USERPROFILE%\Inspector\.profiles

User-defined profiles are stored in an XML format, of which the schema and the XML used to generate the menu in the screenshot above are given below.

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema attributeFormDefault="unqualified" elementFormDefault="qualified" version="1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="root">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="schemaVersion" type="xsd:string" minOccurs="1" maxOccurs="1" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="MenuList">
    <xsd:sequence>
      <xsd:element name="menu" type="Menu" minOccurs="1" maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Menu">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="menus" type="MenuList" minOccurs="0" maxOccurs="1" />
      <xsd:element name="entries" type="EntryList" minOccurs="0" maxOccurs="1" />
    </xsd:sequence>
    <xsd:attribute name="prependSeparator" type="xsd:boolean" default="false" />
  </xsd:complexType>

  <xsd:complexType name="EntryList">
    <xsd:sequence>
```

```
<xsd:element name="entry" type="Entry" minOccurs="1" maxOccurs="unbounded" />
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Entry">
<xsd:attribute name="module" type="xsd:string" use="required" />
<xsd:attribute name="displayName" type="xsd:string" use="required" />
<xsd:attribute name="prependSeparator" type="xsd:boolean" default="false" />
</xsd:complexType>

</xsd:schema>

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<root>
<schemaVersion>1.0</schemaVersion>
<profileName>Example</profileName>
<menus>
<menu prependSeparator="false">
<name>Acquisition</name>
<menus>
<menu>
<name>1</name>
<entries>
<entry displayName="AES Acquisition" module="acquisition/AesAcquisition"/>
</entries>
</menu>
<menu prependSeparator="true">
<name>2</name>
<entries>
<entry displayName="Scope Acquisition" module="acquisition2/ScopeAcquisition"/>
<entry displayName="XY Acquisition" module="acquisition2/XYAcquisition" prependSeparator="true" />
</entries>
</menu>
</menus>
</menu>
<menu>
<name>Crypto</name>
<menus>
<menu>
<name>1</name>
<entries>
<entry displayName="AES Advanced Analysis" module="crypto/AesAdvancedAnalysis"/>
</entries>
</menu>
<menu prependSeparator="false">
<name>2</name>
<entries>
<entry displayName="Calculator" module="crypto2/Calculator"/>
</entries>
</menu>
</menus>
</menu>
</root>
```

2.6 Tools menu

The Tools menu entry contains several shortcuts to configure the hardware (icWaves, VC Glitcher, DLS camera) and Inspector.

The following items are present:

- Hardware Manager: Opens the hardware manager interface. See Section 6.1, “Hardware Manager”.
- Target Manager: Opens the target manager. See Section 6.6.1, “TargetManager”.
- Open camera view: Shows the DLS camera image. For more information, refer to the SmartCardOpticalPerturbation module help file.
- Camera settings: Opens the settings panel of the DLS camera. For more information, refer to the SmartCardOpticalPerturbation module help file.
- icWaves configuration: Opens the configuration panel for the icWaves. For more information, refer to the icWavesConfiguration module help file.
- Splitter configuration: Opens the configuration panel for the Splitter. For more information, refer to the splitterConfiguration module help file.
- Perturbation history: Opens a view showing all the perturbations stored in the database. Each report can then be opened. For more information on the database reports, refer to the VC Glitcher reports help file.
- View Module log: Inspector logs all the parameters, plus the input/output file names if present, for each module that is run. This shortcut opens the log for viewing.

2.7 Installation and configuration

The Inspector software is delivered pre-installed on a PC system to guarantee a working setup for our customers. Of course it is always possible that a new installation is required. For example, a second PC system can be used for running Inspector and of course a new version release of Inspector needs installing. The required hardware and the installation process of the software is described in this paragraph.

Note



Administrator privileges are required to run Inspector.

Hardware

The bare minimum system requirements are:

- Microsoft Windows XP 32 bit or Microsoft Windows 7 64 bit (other operating systems are not supported)
note: a 64 bit driver is not available for the RFTracer, which means it can only be used in the 32 bit version of Inspector
- SUN Java Runtime Environment (JRE) version 1.4 or higher

- 2 GB RAM
- 160 GB hard disc space (dependent on the size of the trace sets)

We ship a system with the following specs:

- XEON 2GHz processor
- 4 GB RAM memory
- 450GB SAS hard disk
- +9800 GT video card with GPU

For working comfortably we advise the following specs:

- 6GB memory
- CUDA supported GPU with > 1GB memory
- HW RAID-0 or -5 on fast disk

64 bit support

The installer provides an option to install a 64-bit version next to a 32-bit version. When checked, an additional start menu item is created to run Inspector in 64-bit mode. Note that 64-bit mode will only work on a 64 bit OS and can only use drivers which provide a 64-bit API (currently this excludes the RFTracer).

Installation

Inspector comes as an installer on the installation CD. This installer creates the Inspector file structure and start-up items during installation. Typically Inspector comes in two flavours:

- Inspector SCA
- Inspector FI

These different flavours use the same software core, but different modules are installed. As a result, both versions can also be combined to be used as one, but the license used must be valid for both versions.

Before starting the installation, make sure to login as a user that has administrative rights. Do not connect any hardware until Inspector is installed.

- Run the installation executable "Inspector-Installer.....exe" from the installation CD. It might take some time, but after a while an installation dialog shows up. Continue installing by pressing next and agreeing with the license agreement,

For Inspector FI, PostgreSQL will be installed. After installation close the PostgreSQL dialog to finish the installation of Inspector.

The release notes for the current version of Inspector are included on the installation CD. For the release notes of the previous versions refer to the Previous release notes appendix.

Related software

Inspector is typically used with different types of hardware. On the supplied PC system the drivers are pre-installed. However with a new installation the Windows operating system asks for a driver, which can be found in Inspector's "drivers" folder. Further information on installing a driver for a specific hardware component can be found in the Hardware section.

Background

The Inspector file structure is as follows:

- Inspector.exe: the Inspector executable
- Inspector.jar: the Java binaries
- Inspector.properties: the Inspector preferences, including configuration and license information, and a history of recently visited files
- Inspector.ico and Uninstall.ico: the icon files
- doc: a folder containing Inspector documentation, such as this manual and document with tutorials
- data: a folder in which users can store trace set files
- jdk: a folder containing the Java JDK
- lib: a folder containing native libraries
- drivers: a folder containing supporting packages and drivers
- modules: a folder in which function modules are stored

The Inspector.exe file is the main executable. It initializes the Java virtual machine in a configuration which is fine-tuned for Inspector, and which works optimally if sufficient memory is allocated for it. The default amount of memory allocated by Inspector is around 1 GByte for the 32bit version. The 64 bit version uses 2/3 of the memory available on the system.

2.8 Using the GPU computing power: CUDA

As of version 4.1, Inspector supports the execution of analysis tasks on graphical processing units supported by the CUDA [http://www.nvidia.com/object/cuda_home.html] technology. CUDA is an architecture developed by NVIDIA for allowing programmers to use their GPUs for general computing. The list of supported graphic cards can be found on NVIDIA's web site: CUDA-Enabled GPU products [http://www.nvidia.com/object/cuda_learn_products.html]. These cards are recommended if they have a memory of at least 1 GB or larger. Cards with less memory are likely to fail while analyzing traces from the GPU. As explained later, Inspector will fall back to using the CPU implementation when such an error occurs.

Installation of CUDA

In order to enable CUDA in Inspector, the CUDA architecture needs to be installed on your machine. To simplify this task, the installers for the CUDA general drivers and for the CUDA

toolkit for Windows XP are shipped with Inspector. The toolkit is located in the drivers folder of the Inspector installation.

After installing CUDA, you will need to upgrade your video drivers. Otherwise, the screen will become garbled when you use Inspector. The latest driver for the specific card should be downloaded from the manufacturer (e.g. NVIDIA [<http://www.nvidia.co.uk/Download/index.aspx?lang=en-uk>])

Inspector use of CUDA

After installing the CUDA drivers and the CUDA toolkit, you need to enable GPU usage within Inspector. To that end, you can go to the Settings window, and find the *Use GPU* check box in the *Other* tab. If your system does not support CUDA, the check box will be disabled. Otherwise, checking this option will instruct Inspector to start using CUDA.

After enabling CUDA, when an operation supported by CUDA is requested, Inspector will profile both the standard implementation and the CUDA implementation. This means that Inspector will run the first few traces through the processing using both implementations and compare their results. Then, it will decide to use the faster implementation.

Once the profiling has been performed for the current settings, it is stored by Inspector so that subsequent calls with the same relevant parameters will reuse the profiling results.

In the event that a given operation fails when being run in the GPU, Inspector will fall back to the standard implementation. This means that using GPU cards with little memory could work for some trace lengths but would probably fail for longer traces. In that case, Inspector will show an error informing that the CUDA execution failed, but will continue the processing on the CPU.

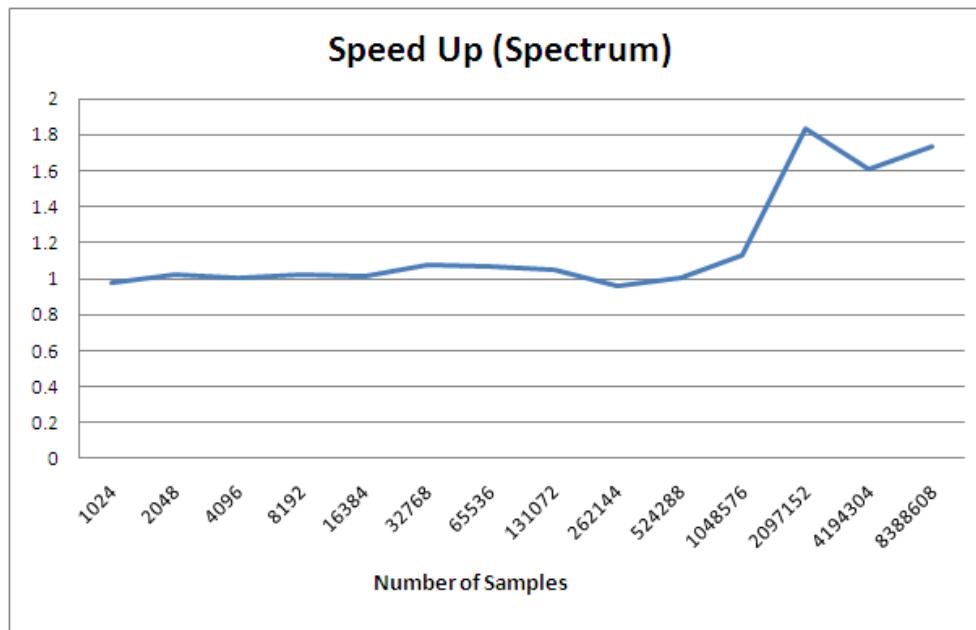
Currently, Inspector only implements FFT related functions in the GPU. Therefore, any Spectrum analysis and any filtering making use of FFT can benefit from the speedup provided by CUDA. In general, the impact will be greater when performing several filtering operations in a Chain.

Expected results

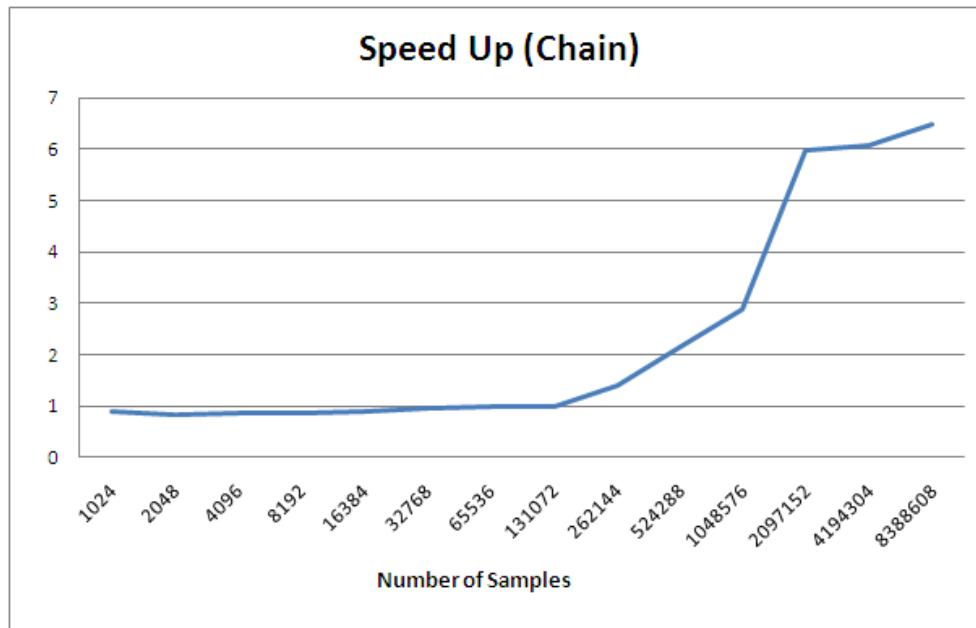
To give an indication of what to expect from the GPU implementation, we have profiled it with different input sizes and different kinds of processing. We have used a workstation with the following configuration:

| | |
|---------------|--|
| CPU | Intel Xeon 5160 @ 3 GHz |
| RAM memory | 2GB SDRAM DDR2 @ 667 MHz |
| GPU | NVIDIA Quadro NVS 290 for graphics and NVIDIA Tesla C870 for CUDA processing |

First, we have tested the performance for a Spectrum operation over a trace set with random traces. The following figure compares the performance with GPU enabled and disabled for different input size. You can see that for small input traces the performance is almost identical, which is expected because then Inspector decides to use the CPU after the profiling phase. Then, speed ups of up to 1.8-2 are obtained thanks to the GPU.

Figure 2.11. Performance comparison for Spectrum

Next, we have applied the same processing with a filter chain. The chain includes the following modules: XtalClear, Spectral and Harmonics. This is a typical chain for RF-EMA acquisitions. The resulting speed up figures are shown below:

Figure 2.12. Performance comparison for long processing chain

From these figures you can see that you can expect a speed up that could go up to 7x depending on your settings. For short traces the overhead of moving the data between the GPU and the CPU is bigger than the performance increase, and therefore the profiling stage chooses the CPU for the execution. On the other hand, for larger traces or chained operations, the GPU is able to provide an interesting performance gain to the analyst.

2.9 License

The license certificate for using Inspector is present in the USB dongles provided after purchase. Without a dongle Inspector will start in DEMO mode. In this mode Inspector will allow loading and browsing of trace sets, but will not allow execution of modules or saving of trace sets. In most cases a permanent certificate is issued with the purchase of Inspector. In specific cases a temporary certificate can be issued.

The validity of the license certificate can be verified in the license dialogue, available through the "Show license" button. Certificates can be updated remotely via email and this license dialogue. Note that when you received a license dongle before July 2009, there is a date validity check in license checker. This will render the license invalid when the date of the PC is set back. If a certificate is rendered invalid and needs to be updated please send an email to inspectorsupport@riscure.com [mailto:inspectorsupport@riscure.com]

2.10 Third party software

Inspector uses the following third party software:

- fplll-3.0 [<http://perso.ens-lyon.fr/damien.stehle/>], which LLL-reduces euclidean lattices, licensed under LGPL v2.1
- Java Native Access (JNA) [<https://jna.dev.java.net/>] version 3.2.5, which enables JAVA programs easy access to native shared libraries (DLLs) , licensed under LGPL 3.1
- Bouncy Castle Cryptography [<http://www.bouncycastle.org/java.html>] version 1.45, specifically the JCE implementation is used and part of the elliptic curve library is used for ECC in new cipher framework, licensed under an adaptation of the MIT X11 license.
- ASM [<http://asm.ow2.org>]version 3.1. This is an all purpose Java byte code manipulation and analysis framework and is used by Inspector to load modules. It is released under its own license.
- RXTX [<http://users.frii.com/jarvi/rxtx/intro.html>] version 2.2. This software is licensed under LGPL v2.1 and is used for serial port communication.
- FFTW [<http://www.fftw.org/>] version 3.1, is a library for Fourier transform and it is licensed under GPL version 2.
- FastDTW [<http://code.google.com/p/fastdtw/>] version 1.1 by Stan Salvador is an approximate Dynamic Time Warping (DTW) algorithm which is used in the ElasticAlign module. This code does not carry any restrictions.

Some pieces of software are released under the GNU GPL [<http://www.gnu.org/copyleft/gpl.html>]<http://www.gnu.org/copyleft/gpl.html> or LGPL license [<http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html>]. This entitles Riscure to use it, while committing to the license. In adherence to this license, the source of this software, including its license is included on the Inspector installation CD, in the "3rd party software" folder. The sources of the software can also be downloaded from the Internet by clicking on the names in the list above.

The third party software is used without modification. The fplll executable, present in <inspector folder>\lib\Win32, is called from the ECCPartialNonce java module. This executable can be modified and replaced by the user if he wishes to do so. This is also true for the FFTW library.

The exact licenses for the included software and their disclaimers can be viewed in the Appendix A, *Third party software licenses* attached to this document.

3 Cryptography

This chapter describes Cryptography tools and terms as used in Inspector, as well as the functionality that is shared between multiple modules in the crypto 2 framework. For explanations of, and attacks on, specific ciphers, please refer to Appendix K, *Cipher Suites*.

As of Inspector 4.6 the whole Crypto framework has been reworked. As such, Cipher code developed prior to Inspector 4.6 is no longer compatible. Tips on migrating old code to the new framework is discussed in Section 3.2, “Migrating pre-4.6 Ciphers”.

Section 3.3, “Cipher selection” describes the element used to select the appropriate cipher, Section 3.4, “Direction” allows the user to set the direction of the attack. In Section 3.5, “Settings”, various settings from the crypto framework are discussed. Section 3.6, “Leakages” details the new Leakages scheme introduced in Inspector 4.6, whereas Section 3.7, “Key Intermediates” shows the new interface to specify and visualise cipher keys. Additionally, every section specifies the modules to which it applies.

3.1 Attacking cryptographic algorithms

An analyst can mount an attack on an implementation of a cryptographic algorithm by first collecting a large set of traces including samples and crypto data (the data which is input or output for the algorithm). It is important that the crypto data be variable in order to establish sufficient variation in the intermediate data to observe correlation. The trace set may need preprocessing to reduce noise and establish alignment. The analyst should select part of the target trace to perform the analysis on. Although it is possible to use the entire trace, this would waste memory and processing resources.

In the legacy Crypto framework, implementing a new algorithm required writing a separate module. In the new Crypto 2 framework, ciphers and leakages have been extracted from the module level, making them into reusable components. In practice this means that once a cipher and its leakages have been created, they can be used for all Crypto 2 functionality, such as Known Key Analysis or Simulator.

3.2 Migrating pre-4.6 Ciphers

The easiest way to migrate ciphers that were created before Inspector 4.6 is to create a new Cipher using the New Module Wizard from the File menu. Once the new Cipher has been created, copy the encrypt and the decrypt methods from the old Cipher (including optional helper methods) and have these replace the encrypt and the decrypt methods of the new Cipher.

Keep in mind that the user no longer needs to define a Leakages object for a cipher. Leakages are now defined through the user interface and saved as XML files. All the information needed for the creation of Leakages through the UI is gathered from the Intermediates object.

3.3 Cipher selection

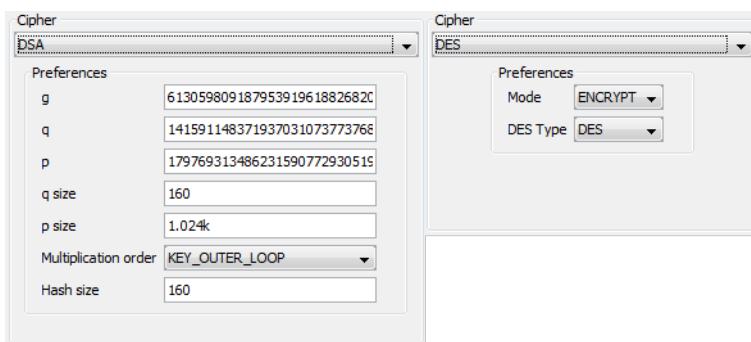
Applicable to:

- Section E.7.1, “Calculator”
- Section E.7.2, “First Order Analysis”

- Section E.7.4, “Known Key Correlation”
- Section E.7.5, “Simulator”
- Section E.7.6, “Template Analysis”
- Section E.7.7, “Verify”
- Section E.7.8, “Known Key Analysis”
- Section E.7.10, “Points Of Interest Selection”

This subsection shows the functionality of the cipher selection panel. For explanations of, and attacks on, specific ciphers, please refer to Appendix K, *Cipher Suites*.

Figure 3.1. The cipher selection panel



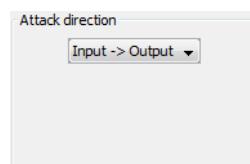
The cipher selection panel, shown in Figure 3.1, “The cipher selection panel”, is used to select the cipher used for the operation of the module. Selecting a cipher will show a settings panel if the cipher requires additional settings. If the module features a Leakage selection panel, as described in Section 3.6, “Leakages”, selecting or changing a cipher will also show the attacks that are appropriate for the current cipher.

3.4 Direction

Applicable to:

- Section E.7.2, “First Order Analysis”
- Section E.7.6, “Template Analysis”
- Section E.7.8, “Known Key Analysis”
- Section E.7.10, “Points Of Interest Selection”

Figure 3.2. The direction panel

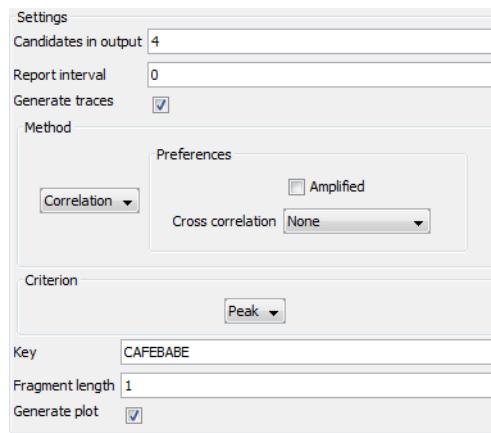


The direction panel is used to specify the direction of the attack, either *Input -> Output* or *Output -> Input*, as shown in Figure 3.2, “The direction panel”. If the module features a Leakage selection panel, as described in Section 3.6, “Leakages”, changing the direction will change the available attacks.

3.5 Settings

This section details several settings that are used by multiple modules in the crypto 2 framework. An example of a settings panel, in this case of the Known Key Analysis module, is shown in Figure 3.3, “A settings panel”

Figure 3.3. A settings panel



3.5.1 Key

Applicable to:

- Section E.7.1, “Calculator”
- Section E.7.4, “Known Key Correlation”
- Section E.7.5, “Simulator”
- Section E.7.6, “Template Analysis”
- Section E.7.7, “Verify”
- Section E.7.8, “Known Key Analysis”
- Section E.7.10, “Points Of Interest Selection”

This setting is used to specify the key that is used to run the cipher.

3.5.2 Candidates

Applicable to:

- Section E.7.2, “First Order Analysis”

- Section E.7.6, “Template Analysis”
- Section E.7.8, “Known Key Analysis”

This setting specifies the number of best key candidates that are considered when performing an attack. Increasing this number could increase the likelihood that a correct key is found, but will also increase the time it takes to run the algorithm.

3.5.3 Method

Applicable to:

- Section E.7.2, “First Order Analysis”
- Section E.7.8, “Known Key Analysis”

This setting specifies one of two methods to compute the result:

- **Differential**: means a classical differential trace where all traces corresponding to a data bit set to 1 are collected in one group while all other traces are collected in the second group. The result is given by subtraction of the average trace of both groups. This method is relatively fast, but also sensitive to noise.
- **Correlation**: means a correlation trace where result traces are computed as a sequence of correlation coefficients of sample columns with data columns. This method is slightly more resource consuming, but produces better results in the presence of noise.

Choosing the Correlation method opens up several other options, which further refine the behavior of this method.

- **Amplified**: means an amplified correlation trace where the mathematical computation of correlation is amplified by a modified standard deviation.
- **Cross correlation**:
 - **None**: No Cross Correlation applied.
 - **Cross correlation**: cross-correlation. Here we take into account that incorrect candidates may also produce significant peaks, and incorporate the strength of various ghost peaks.
 - **Euclidian Similarity**: like Cross Correlation, but uses Euclidean Similarity as a measure instead of correlation.

3.5.4 Criterion

Applicable to:

- Section E.7.2, “First Order Analysis”
- Section E.7.8, “Known Key Analysis”

This setting specifies the way of selecting a good correlation.

- **Peak**: Just search for the highest peak in the result trace.

- **Sum:** Add all values in the selection that exceed a noise threshold given by $2/vn$, where n is the number of traces.

3.5.5 Report

Applicable to:

- Section E.7.2, “First Order Analysis”
- Section E.7.8, “Known Key Analysis”

This setting specifies the number of processed traces after which the algorithm will print an intermediate result. The system will then continue to generate intermediate results every time that that number of traces are processed, until the process finishes. Setting this to 0 will not generate any intermediate results.

3.5.6 Generate traces

Applicable to:

- Section E.7.2, “First Order Analysis”
- Section E.7.6, “Template Analysis”
- Section E.7.8, “Known Key Analysis”

This setting specifies whether the module should generate output traces. Typically the user will want to save the correlation traces that the module outputs. However, disabling this option will reduce the runtime, while still printing key/correlation pairs on the log window.

3.6 Leakages

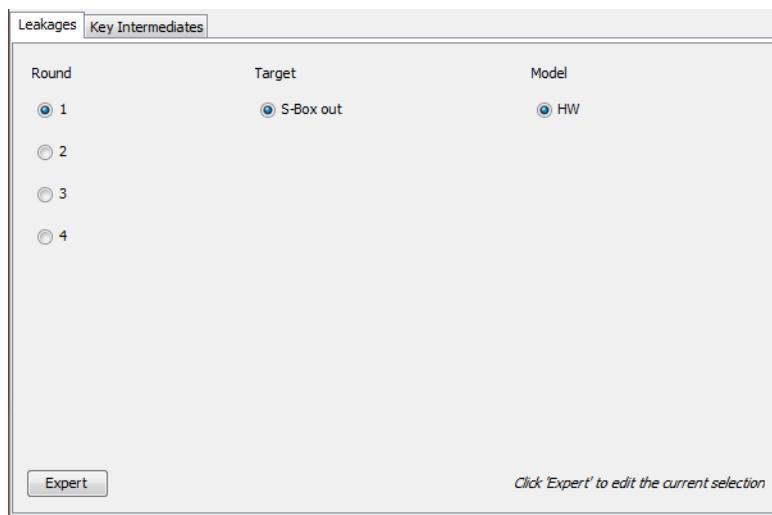
Applicable to:

- Section E.7.2, “First Order Analysis”
- Section E.7.4, “Known Key Correlation”
- Section E.7.5, “Simulator”
- Section E.7.6, “Template Analysis”
- Section E.7.8, “Known Key Analysis”
- Section E.7.10, “Points Of Interest Selection”

This subsection explains the leakage selection panel. Leakage selection has been split into two distinct parts, the Normal View and the Expert View. In general, the Normal View should be enough for the average user. The Expert View should only be used to define custom attacks, when the desired attack is not available in the Normal View.

3.6.1 Normal View

Figure 3.4. The (Camellia) Normal View



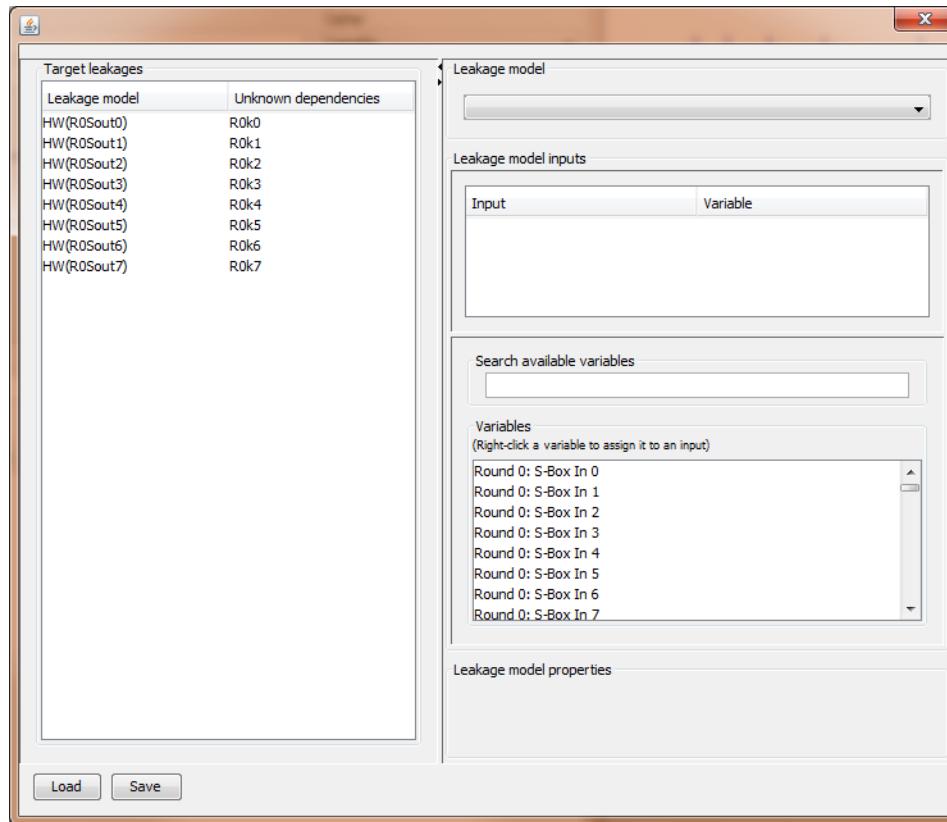
The Normal View displays common attacks on the selected cipher. For explanations of cipher-specific terms, such as S-Box, and how to attack specific ciphers, please refer to Appendix K, *Cipher Suites*.

In the Normal View, attacks are represented by several columns of radio buttons as shown in Figure 3.4, “The (Camellia) Normal View”. In order to fully define an attack, a single selection is required in every column. Please note that some combinations of choices are incompatible. This is displayed by incompatible choices turning red. When an incompatible (red) option is selected, the incompatible other selections are automatically deselected and, in turn, will turn red themselves.

In order to customise a (partial) attack that was selected in the Normal View, press the button labeled "Expert". This will automatically display the selected Normal View attack in the Expert View panel, ready to be customised.

3.6.2 Expert View

Figure 3.5. The (Camellia) Expert View



The Expert View, shown in Figure 3.5, “The (Camellia) Expert View”, shows all known leakages for the current cipher, and allows for the user to select them independently. In order to successfully create an attack from scratch, please follow this workflow:

1. Right click in the panel marked "Target leakages" and select "add new leakage model"
2. Select the created leakage model
3. From the top right panel (marked "Leakage model") select the desired model. For explanations of the listed models, see Section 3.6.3, “Leakage models”.
4. From the Variables panel select the (number of) desired leakage(s) required by the leakage model, and the leakage model properties (if any).
5. Repeat this process until the entire attack has been defined. Note that leakage models can be copied for quicker attack building.
6. When the attack is done, make sure to save the configuration to a file by clicking "Save" for future reference.
7. Close the Expert View pop-out to return to the previous view.

There are several things to keep in mind when creating custom attacks:

- Generally speaking, only leakage models with a single dependency can be resolved.

- Leakage models without any dependencies have a known value and cannot be attacked.
- All leakages have been split up into bytes. To attack a full key, please add all the parts of the related leakage to the attack.

3.6.3 Leakage models

In order to make full use of the power of the Expert View, the user must understand the leakage models that are presented in this view. This section will explain all leakage models used in the Expert View. The models are presented with a name, followed by their abbreviation.

Bit: Bit[index](a)

Number of inputs: 1

Maximum output value: 1

The Bit model takes a single input byte and outputs only the single bit specified in the properties, counting from the most significant bit (i.e. bit index '0' is the most significant bit, bit '7' is the least significant bit).

Hamming Distance: HD(a; b)

Number of inputs: 2

Maximum output value: $\max(\text{length}(a), \text{length}(b))$

The Hamming Distance model takes two inputs and outputs the number of bits that are different. Formally: $\text{HD}(a,b) = \text{HW}(a \text{ xor } b)$. Also see Section 3.6.3, "Hamming Weight: HW(a)".

Hamming Weight: HW(a)

Number of inputs: 1

Maximum output value: $\text{length}(a)$

The Hamming Weight model takes a single input and outputs the number bits that are logic '1'.

Identity: Identity(a)

Number of inputs: 1

Maximum output value: a

The Identity model takes a single input and outputs the value of the input

Identity XOR: (a XOR b)

Number of inputs: 2

Maximum output value: $\max(a, b)$

The Identity model takes two inputs and outputs the value of the bitwise XOR of the inputs. Also see Section 3.6.3, "Identity: Identity(a)".

Switching Distance: SD($d=x.xxx$; a ; b)

Number of inputs: 2

Maximum output value: $\max(\text{length}(a), \text{length}(b))$

The Switching Distance model takes two inputs and compares them bitwise. For every 0->1 transition it adds 1 to the output, for every 1->0 transition it adds $1-d$. This is also shown in the following table:

Table 3.1.

| x | y | out |
|---|---|-------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | $1-d$ |
| 1 | 1 | 0 |

XOR: Bit[index](a XOR b)

Number of inputs: 2

Maximum output value: 1

The XOR model takes two inputs and outputs the value of the specified bit after performing the bitwise XOR of the inputs. Also see Section 3.6.3, “Bit: Bit[index](a)”.

3.7 Key Intermediates

Figure 3.6. The Key Intermediates View

| Key | Value |
|----------|-----------|
| Round 0 | <not set> |
| Round 1 | <not set> |
| Round 2 | <not set> |
| Round 3 | <not set> |
| Round 4 | <not set> |
| Round 5 | <not set> |
| Round 6 | <not set> |
| Round 7 | <not set> |
| Round 8 | <not set> |
| Round 9 | <not set> |
| Round 10 | <not set> |
| Round 11 | <not set> |
| Round 12 | <not set> |
| Round 13 | <not set> |

Applicable to:

- Section E.7.2, “First Order Analysis”
- Section E.7.4, “Known Key Correlation”

- Section E.7.5, "Simulator"
- Section E.7.6, "Template Analysis"
- Section E.7.8, "Known Key Analysis"
- Section E.7.10, "Points Of Interest Selection"

The Key Intermediates panel, shown in Figure 3.6, "The Key Intermediates View", allows the user to view and set all keys related to the current cipher. If the user has not performed an attack on the current cipher, all values will display <not set>.

To manually set a key, click in the value field for the appropriate round- or sub key and set the value and press "enter" or click on another field in the table.

To unset the key, remove the complete value from the box and press "enter" or click on another field in the table. To unset ALL values in the table, click "Clear all values".

To partially set a round key, select the appropriate round from the drop down box. This will cause the view to switch to the Sub keys table. Please note that if a round key is partially set, the round key table will still show <not set>, since there is no way to determine its complete value. Also note that changing the value in either table will have immediate effect on the values in the other table, i.e. changing the round key will set and/or change all the values in the sub keys panel and changing a single sub key will change the value of the round key.

4 Side Channel Acquisition

This chapter describes the many details of performing Side Channel Analysis (SCA) with Inspector. It begins with describing the process of an SCA with the Power Tracer. It continues describing performing an SCA with the RF Tracer and ways of processing the acquired traces for getting better analysis results. More advanced subjects like EM acquisition, module development and controlling embedded devices with Inspector are also described in this chapter. It concludes with a section on developing drivers and the coding format of the trace sets.

4.1 Select the acquisition methodology

Inspector supports two methods for acquiring side channel information: legacy acquisition and new acquisition.

The legacy acquisition framework was the only method available up to Inspector 4.4 in which the new acquisition framework was introduced. In Inspector 4.6 feature parity between the new and the legacy framework was achieved. For new projects the new acquisition framework should be used. The legacy framework will be supported for some time, however no new features will be included.

4.1.1 Legacy vs new acquisition

Legacy acquisition is built around a monolithic concept. All acquisition functionality is included in a single module. This concept makes it easy to change or extend every single aspect of acquisition. However changing a single aspect, for example the communication protocol with the target, requires knowledge of the entire framework. Furthermore in order to change a single aspect the same change has to be made to multiple modules. For example when changing the communication protocol both the acquisition with and without XY table have to be modified.

The new acquisition framework has a modular design. Each aspect of the acquisition is captured in a single component. In order to achieve the desired functionality one can simply combine the appropriate components. Inspector includes predefined modules for all common acquisition scenarios. The modules are built in such a way that the communication protocol is configurable. Hence in order to use a new communication protocol it only has to be written once and can then be used in every acquisition scenario including perturbation. While making changes to the communication protocol is considerably easier compared to legacy acquisition changing the entire flow of acquisition is harder.

4.1.2 Migrating to the new acquisition

Anyone familiar with subclassing ‘SideChannelAcquisition’ and implementing ‘initAcquisitionProcess’ and ‘runAcquisitionProcess’ will find very familiar functions when subclassing ‘BasicProtocol’ and implementing ‘init’ and ‘run’. The largest difference when migrating is that the protocol is no longer arming the measurement setup itself. Instead the protocol indicates which ‘phase’ it is in. When the phase that is selected by the user in the user interface is reached the measurement setup is armed.

Communication protocols for all Riscure provided training targets are included as examples. In addition to the basic communication these protocols also provide examples on how to set the

verdict so that the protocol can be used in perturbation, how to handle errors gracefully using ‘onError’, and how to implement a user interface for easy protocol configuration.

4.2 Hardware setup tasks

Inspector requires communications devices and acquisition devices. Explain how to knot these two together here - it is the same for both methodologies.

4.2.1 Communication devices

A communication device is a device by which Inspector communicates with the target. The PowerTracer, MP300 TCL1/TCL2, RFTracer and VCGlitcher are communication devices for smart card targets. The serial port or a network interface can be used for driving embedded targets.

Most communication devices provide a trigger to the measurement setup to indicate the start of the measurement.

4.2.2 Acquisition devices

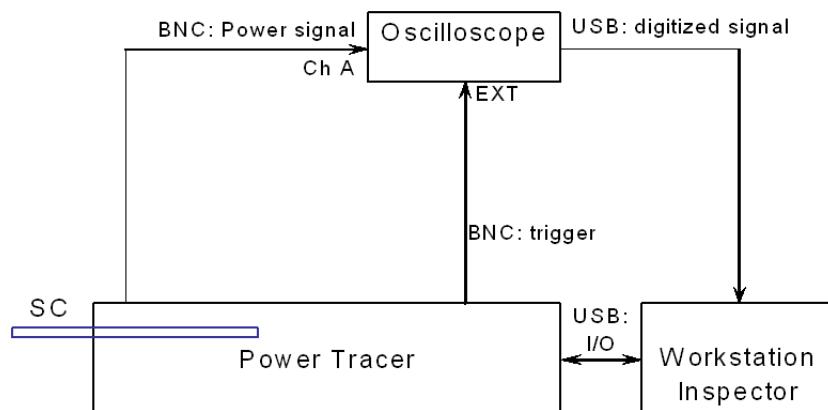
There are several hardware tools available for doing Side Channel Analysis with Inspector. The main tool is the Power Tracer, but also available are the MP300 and RF Tracer for doing SCA on contactless cards and the EM Probe Station for measuring the EM emission instead of power consumption. The hardware specifications and the exact way of connecting the device are detailed in Chapter 6, *Inspector Hardware Components*.

4.2.2.1 Contact cards (Power Tracer)

Figure 4.1, “Measurement setup with the Power Tracer” shows a schematic of a general setup for power measurements with the Power Tracer. The smart card (SC) is contained by the Power Tracer. The Power Tracer is controlled by Inspector through the USB interface and provides the following I/O functionality:

- I/O communication with the smart card
- Trigger signal to the oscilloscope
- Power signal to the oscilloscope

Figure 4.1. Measurement setup with the Power Tracer



The exact way to connect the Power Tracer is described in the Hardware chapter.

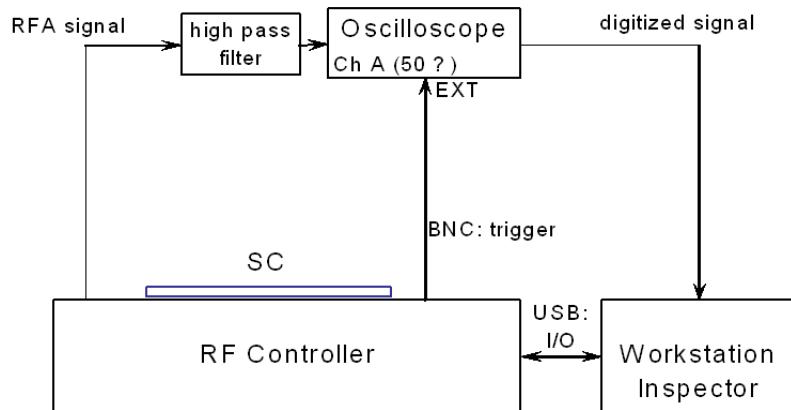
4.2.2.2 Contactless cards (MP300 with TCL1/TCL2)

The smart card (SC) is placed on the external antenna of the MP300. The MP300 provides:

- I/O communication to the smart card
- Trigger signal to the oscilloscope
- RFA signal to the oscilloscope

The 25 MHz high pass filter attenuates the strong 13.56 MHz component in the RFA signal. Figure 4.2, "Schematic of a typical RFA measurement setup" shows the schematic of the hardware setup for RFA measurements.

Figure 4.2. Schematic of a typical RFA measurement setup

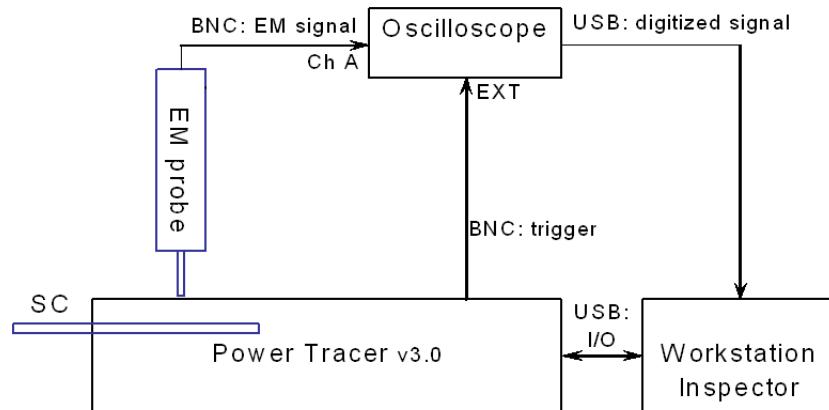


4.2.2.3 EM Probe Station

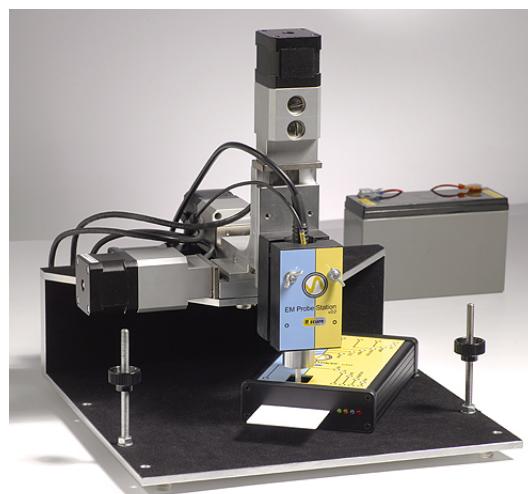
The smart card (SC) is contained by the Power Tracer as with the normal Power Analysis. The Power Tracer provides:

- I/O communication to the smart card
- Opening for the sensor tip of the EM probe
- Shielding of the smart card and sensor tip against exterior EM noise
- Electrical isolation between the smart card and the digital section of the measurement system to minimize disturbance

The EM emanation of the smart card is measured by the EM probe. The EM probe provides the EM signal to the oscilloscope. Figure 4.3, "Setup for EM measurements" shows the basic setup for EM measurements.

Figure 4.3. Setup for EM measurements

The *EM Probe Station* consists of two EM Probes (EM sensor and an amplifier), XYZ stepping motors and a controller for the motors. The two probes have different sensitivity. The High Sensitivity (HS) is intended for test objects with weak EM emanations like modern smart cards or other device with low power consumption and/or metal layers that act as magnetic shields. The Low Sensitivity (LS) probe is intended for test objects in the presence of a strong EM noise field (like contactless cards in the RF field) or test objects that emit a strong EM field (objects with a high power consumption). The HS and LS probes are available from version 4 of the EM probe. The probe output is connected to one of the input channels of a digital oscilloscope.

Figure 4.4. EM Probe Station with Power Tracer.

The opening in the Power Tracer should be positioned directly underneath the sensor tip of the EM probe, see Figure 4.4, “EM Probe Station with Power Tracer.”. After inserting the smart card in the Power Tracer, the EM probe can be lowered. The sensor tip will touch the smart card lightly. By adjusting the X and Y position of the Power Tracer relative to the sensor tip, the optimum position for EM measurements can be reached. Be aware that shifting the Power Tracer beyond the limit positions may damage the sensor tip. The limit positions can be found in a safe way while the sensor tip is lifted above the opening of the Power Tracer. Inspector controls each measurement in the same manner as power measurements

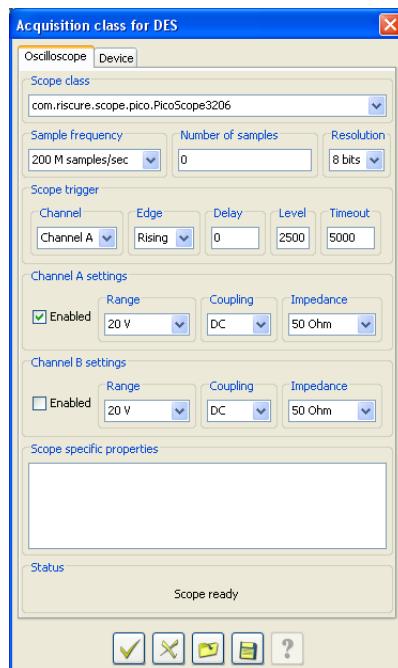
4.3 Using Acquisition modules

Inspector is an ideal platform for controlling your oscilloscope and managing large signal acquisition sessions. Inspector comes with multiple modules that implement the acquisition of signals.

4.3.1 Oscilloscope

For collecting single traces an analyst would use the Oscilloscope module. This module has a dialog where different oscilloscopes can be selected and all relevant oscilloscope parameters can be controlled. The dialog of this module used in an acquisition class is shown in Figure 4.5, “Oscilloscope dialog”. Read the documentation of the Oscilloscope module for more details. The actual oscilloscope is represented by an implementation of the Scope [..javadoc/com/riscure/scope/Scope.html] interface. The Oscilloscope module can switch between various implementations of the Scope interface. An analyst can add support for new oscilloscopes by defining a class that implements the Scope interface.

Figure 4.5. Oscilloscope dialog

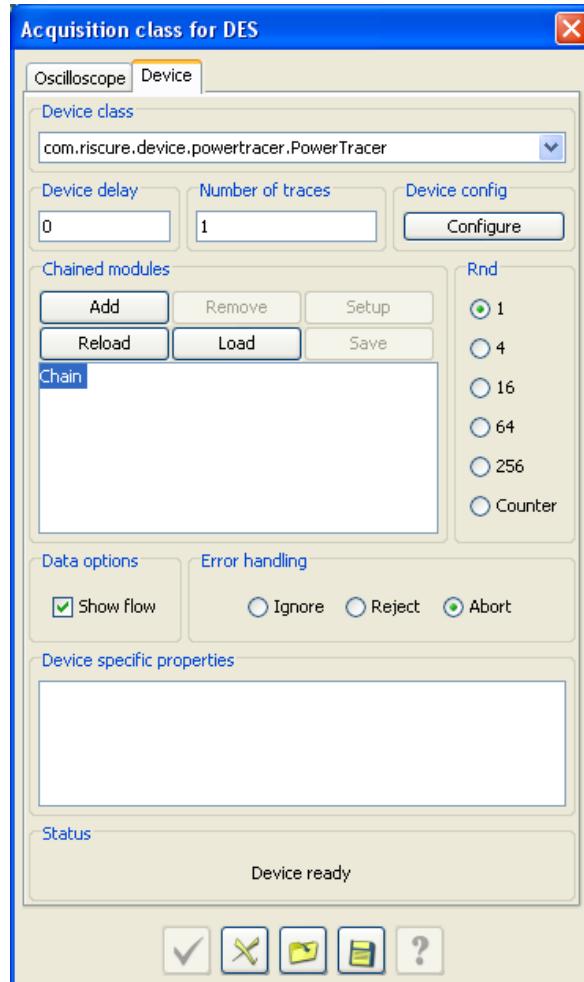


For collecting multiple traces an analyst would use a sub class of the SideChannelAcquisition module. This module extends the Oscilloscope module to run a session where a device (e.g. smart card) is repeatedly requested to perform an operation which is subsequently recorded with the oscilloscope. This module can control the external device through the so-called Device [..javadoc/com/riscure/device/Device.html] interface. This interface allows the module to start the operation to be monitored and trigger the oscilloscope at the appropriate time. An analyst may build his own Device implementation to control their favorite device. The actual application protocol on the external device under test is application dependent, and should be implemented in a sub class. Analysts should therefore not run the SideChannelAnalysis class directly, but rather implement the appropriate application behavior in a sub class which can then be executed. Riscure provides a module called TestApplet as an example sub class. Read the documentation of the SideChannelAcquisition module for more details.

4.3.2 Device

The dialog of the SideChannelAcquisition module in the DES acquisition class is show in Figure 4.6, “Side Channel Acquisition dialog”. This dialog is set to use the Power Tracer as device interface.

Figure 4.6. Side Channel Acquisition dialog



Users that want to build classes that implement the Scope and Device interfaces can follow the instruction in the Drivers section.

Inspector controls each measurement in the following manner:

- Inspector generates a randomized clear message text
- Inspector transfers the message text to the smart card via the Power Tracer
- The Power Tracer generates a trigger signal to start the oscilloscope measurement
- Inspector retrieves the cipher text from the smart card through the Power Tracer

This sequence is repeated as often as required by the security analyst. Every time this sequence is performed a trace is produced which is stored by Inspector. This results in a complete set of traces where every trace is linked to input and output data from the smart card.

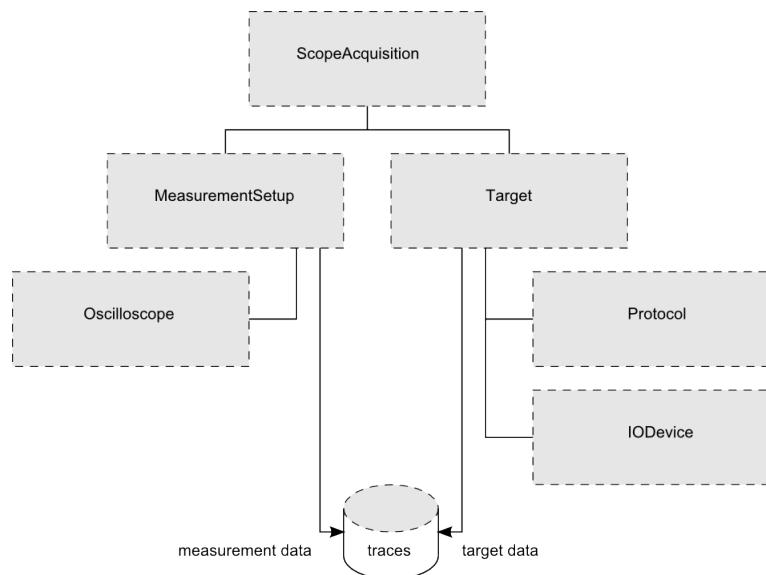
4.4 Using Acquisition2 modules

Due to the increasing number of supported acquisition devices and targets, we designed a framework for implementing various acquisition tasks in a structured and modular way. Based on this framework we can perform acquisitions on smart cards or embedded devices, possibly combined with an XY table, with many choosable protocol options, etc.

4.4.1 Acquisition2 Framework

The best way of explaining the new framework is by using ScopeAcquisition module as an example, see Figure 4.7, “ScopeAcquisition architecture”.

Figure 4.7. ScopeAcquisition architecture



The ScopeAcquisition module can be selected from the **Run** menu, submenu **acquisition2**. This module uses a preconfigured **MeasurementSetup** of one oscilloscope, and a Target consisting of an IODevice and a protocol.

The figure above shows the ScopeAcquisition module which has two components:

- A MeasurementSetup, containing devices capable of collecting measurement data, for example the trace of an oscilloscope or the X,Y coördinates of an XYTable. The MeasurementSetup provides the measurement data to the trace set.

For ScopeAcquisition, the MeasurementSetup only contains an oscilloscope.

- A Target, which consists of an IODevice to which the physical target is connected, and the protocol to use in target communications. The Target provides protocol data to the traceset. More information about the protocols can be found in Section 5.4.2, “Communication protocols”.

Each of the items in the figure will have configurable parameters specific to the selected oscilloscope, protocol or IODevice.

When the acquisition is started, the acquisition module will run the target while starting the measurement setup. The target provides the relevant protocol data to the traceset, which is stored along with the measurement data provided by the MeasurementSetup.

The traceset can be processed as usual.



Note

Typically, a user would use the Power Tracer device for acquisition. It is possible to use the VC Glitcher device instead, though this is not advised. See Note on acquisition with VC Glitcher.

4.4.2 Data Generator

The protocol is responsible for providing input data to the target. In acquisition2, protocols can use a *Data generator* to produce such data. The protocols for the training cards support this option. The user can also include an (automatic) data generator in his custom protocols.

The data generator has five different "flavors". For each type, the "Data generator settings" panel will have a slightly distinct layout.

- *Fixed data*: For each measurement, the same data string (or byte array) is sent to the target. The Automatic Gui Generator will show a "value" field on the Data generator settings panel. The user can enter any hexadecimal string here. If the string is shorter than the required data length it will be prepended with zeros. If the string is longer than the required data length it will be trimmed at the end (right-hand side).
- *Fixed list*: Inspector will cycle through a list of data values. In the settings panel the user can specify his "fixed list" as a series of space/comma-separated hex strings. Mind that each value has to be prefixed with 0x.
- *Random data*: Each measurement will contain random data. All data bytes will have a value drawn from a uniform random distribution. On the settings panel the user can enter a "seed" value to initialize the PRNG. A seed value of zero will lead to a completely random sequence, i.e. the seed will be based on the system clock. A non-zero seed value will enable the user to repeat the same experiment, since the sequence of random numbers will be reproducible.
- *Random list*: Sometimes it might be useful to send n random data strings, but repeat those same input strings multiple times. When the user enters "Cycle length = n" the data generator will produce random strings (based on the given seed value), but it will reset the seed to its initial value after every n measurements. Just like with the random data generator, a seed value of zero will lead to an unpredictable sequence. However, that random sequence will still be repeated.
- *Step data*: The user can specify the "initial value" for the first measurement; in the subsequent runs the value will be incremented with the "step size" value. The (initial) data string is padded with zeros on the left, if necessary. If the string is longer than the required length it will be trimmed at the start (left-hand side). Note that this behavior is different from the Fixed data generator.

Any protocol that implements the **DataConsumer** interface will automatically receive a data generator. The protocols created with the "New module wizard" are extensions of the **BasicProtocol** class, which indeed implements the **DataConsumer**. This means that newly

constructed protocols will contain this functionality by default. See Section 9.8.5, “How should I obtain input data for my protocol?” for more information about the **DataConsumer** interface.

4.5 Modifying modules

If no module is implementing the acquisition for the chosen target, a module needs to be developed. This is described in Section 8.3, “Developing Classic Modules” for Acquisition modules and in Section 8.4, “Inspector OSGi framework” for Acquisition2 modules.

5 Fault Injection

Fault Injection (FI) is the process of stressing a target in such a way that it no longer functions according to the specifications of the target.

Inspector supports various methods of introducing perturbations on a target. Either directly by the supply voltage or interfaces on the target, or indirectly by means of laser.

This chapter describes the steps involved to perform fault injection attacks with Inspector.

5.1 Preparing the hardware setup

This section describes the most common hardware configurations for performing a Fault Injection attack with Inspector. Each configuration is a starting point for building a working and optimized measurement setup for your target. Adding active or passive filters may be required to improve signal analysis.

5.1.1 Device List

The hardware setup for Fault Injection is highly flexible and can consist of the following devices:

- an IO device for communications with the target,
- a VC Glitcher for providing the perturbation signals,
- an oscilloscope for collecting traces,
- optionally a Diode Laser Station for optical perturbation,
- optionally an EM-FI Transient Probe for electromagnetic perturbation,
- optionally a Power Glitcher for voltage perturbation on embedded targets,
- optionally a current probe to measure power consumption by the target
- optionally an icWaves pattern matching device to avoid damage to the target
- optionally a Splitter for multi-area optical perturbation

IODevice

Inspector needs an IO device to send and receive data with the application running on your target. For smart card targets, the IO device is a card reader device such as the VCGlitcher, Power Tracer or MP300. For any other target it is usually a serial interface.

VC Glitcher

Perturbation signals are generated by the VC Glitcher. When the VCGlitcher is also used as the IODevice, the signals are directly applied to the target. In other setups the signals of the VCGlitcher are used to control a Diode Laser Station or a Power Amplifier.

Oscilloscope

An oscilloscope is used to observe the effects of a perturbation attempt on the target. Setup and use of the oscilloscope in a perturbation setup is similar to setup and use in Side Channel Acquisition.

Diode Laser Station

A diode laser station is used to perform optical perturbation attacks. A VC Glitcher is used to drive the lasers. The positioning of the laser beam is controlled by Inspector.

EM-FI Transient Probe

An EM glitch generator in combination with an EM-FI probe can be used to perform EM perturbation attacks. This glitch generator can be attached to the EMPS in the same way as an EM acquisition probe. EM glitching resembles laser glitching in its operation.

Glitch Amplifier

The Glitch Amplifier is a unity gain amplifier used to amplify the perturbation signal from the VCGlitcher so it can be used as a power supply to the target.

icWaves

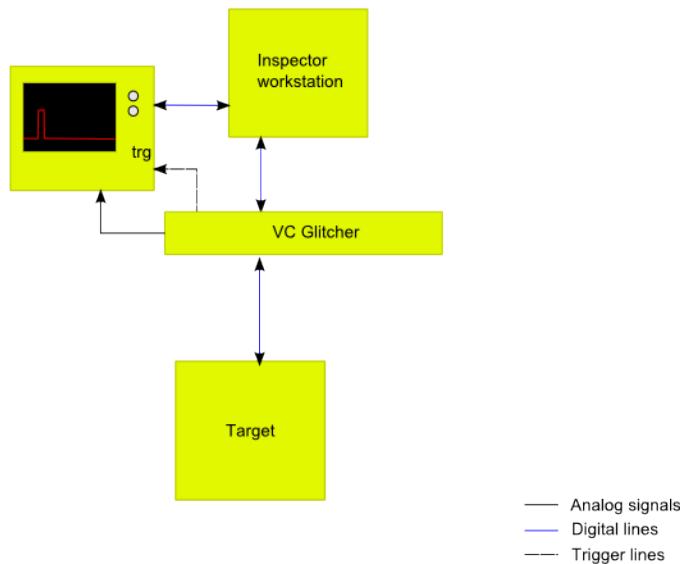
The icWaves is used to prevent a target from executing self-destruct countermeasures. It does this by observing power consumption (or any other side channel signal) and send a trigger when the signal matches a pattern. The trigger can be used to power-off the target. Pattern matching and the use of the icWaves is described in detail in Chapter 7, *Real-Time pattern matching*

Splitter

The Splitter is used to route the perturbation pulse coming from the VC Glitcher to one of the lasers connected to the Splitter.

5.1.2 Configuration A: ISO7816 smartcard with power glitching

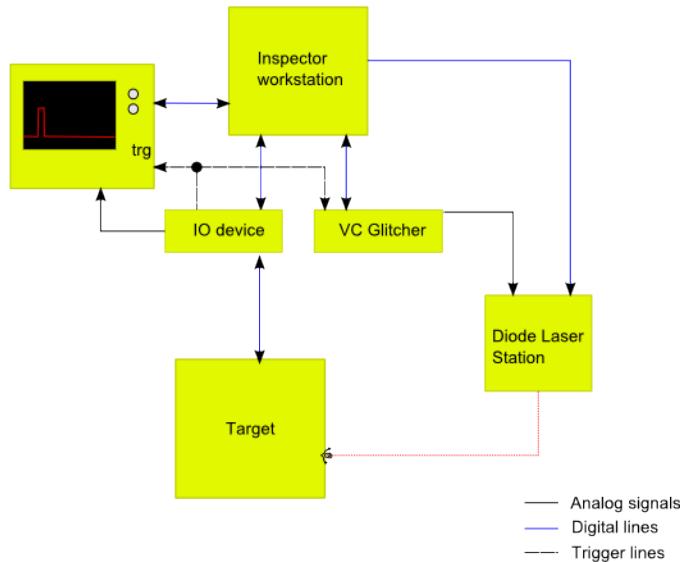
The following configuration shows how to use a VCGlitcher to perform voltage glitching on a ISO7816 target.

Figure 5.1. Configuration A

5.1.3 Configuration B: IO Device with optical fault injection

The following configuration shows how to use a VCGlitcher to perform optical fault injection on a target on which the IO is not handled by the VCGlitcher. The VCGlitcher receives a trigger from the IO device when a perturbation sequence is to start.

The IO device provides the required side channel information to the oscilloscope. An example of using this configuration could be an MP300 TCL1 or TCL2 resource as the IO Device and a contactless smartcard as the test target.

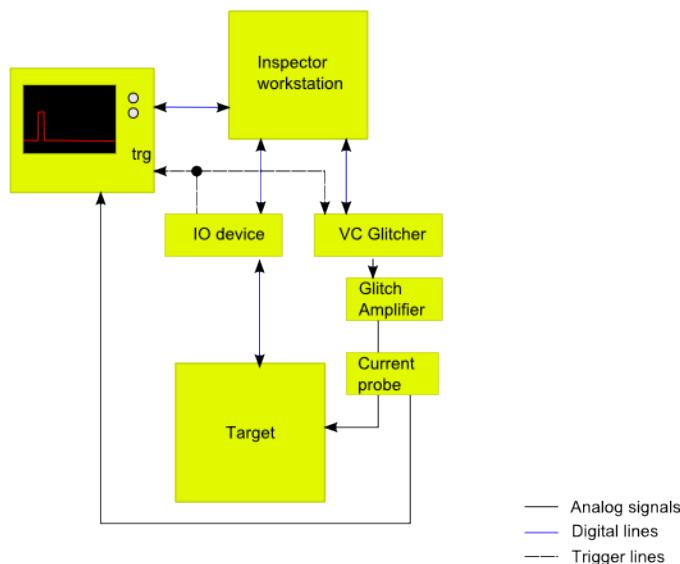
Figure 5.2. Configuration B

5.1.4 Configuration C: Embedded target with voltage fault injection

This configuration shows how to use a VCGlitcher with a Glitch Amplifier perform voltage perturbations. Voltage glitching can be performed on a clock or supply line. When the target is a SoC type target, power consumption of the target may exceed the maximum output ratings of the VC Glitcher. The Glitch Amplifier has higher output ratings that allow devices to be powered by its outputs.

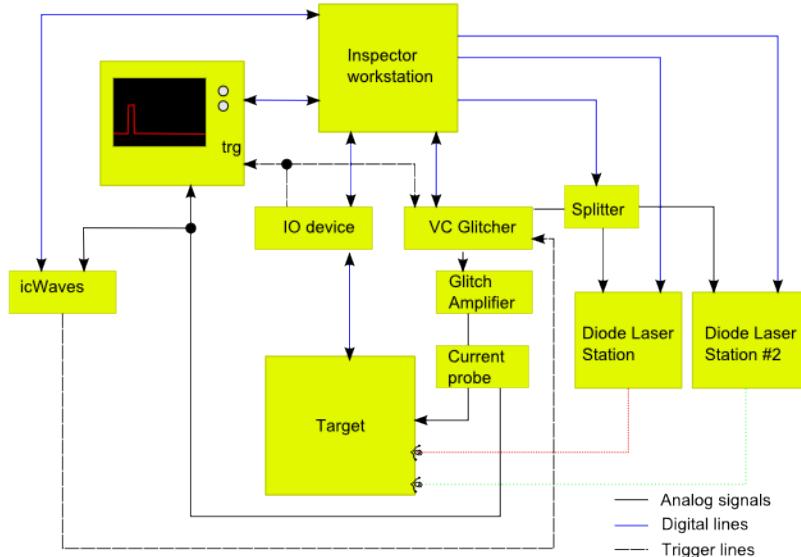
A consideration in this configuration is to only drive the power supply lines that feed the target's internal components under test, i.e. the crypto accelerators.

Figure 5.3. Configuration C



5.1.5 Configuration D: Multi-area optical fault injection and protection

This configuration shows the VC Glitcher connected to the Splitter so it can provide trigger pulses for multiple lasers. The positioning of each laser is controlled by Inspector. The configuration also introduces the icWaves device, which provides a trigger to the VC Glitcher. When the VC Glitcher receives this trigger, it turns off the power supply to the target, preventing it from executing a self-destruct sequence.

Figure 5.4. Configuration D

5.1.6 Configuration E: IO Device with EM fault injection

This configuration can be likened to Section 5.1.3, “Configuration B: IO Device with optical fault injection” and allows EM fault injection on a target with which the IO is not handled by the VCGlitcher.

The setup is similar to Figure 5.2, “Configuration B” except for the fact that the VCGlitcher output is connected to the EM-FI glitch source and the Inspector workstation is connected to the EM Probe Station that holds the EM-FI Transient Probe.

5.2 Select the perturbation methodology

Inspector provides two frameworks for perturbing targets: legacy perturbation and new perturbation.

The legacy perturbation framework was the only method available up to Inspector 4.5 in which the new perturbation framework was introduced. In Inspector 4.6 feature parity between the new and the legacy framework was achieved. For new projects the new perturbation framework should be used. The legacy framework will be supported for some time, however no new features will be included.

5.2.1 Legacy vs new perturbation

Legacy perturbation focuses primarily on smart cards and assumes the communication with the target handled by the VC Glitcher inside a perturbation program. These limitations make it hard to use for targets other than smart cards and make it very hard to glitch a crypto protocol.

The new perturbation framework is intended to inject faults in any embedded target including smart cards. Unlike the legacy framework communication is handled on the PC rather than inside the VC Glitcher. This means any I/O device can be used to communicate with the target, besides the VC Glitcher this includes the serial port, PowerTracer, and MP300. Because

the new perturbation framework is built on the new acquisition framework it uses the same modularity. This allows many different scenarios to be supported, including dual optical for multi area glitching. In most cases it is no longer necessary to write a perturbation program, the communication is handled by the Protocol in Java and glitches can be configured using the user interface. In case more control is desired this is also available through Program and Advanced modules.

5.2.2 Migrating to the new perturbation

The communication protocol no longer has to be implemented in the perturbation program. The first step in migrating is to implement the communication protocol in Java in the new acquisition framework so that it is possible to perform acquisition on the target. Once acquisition is possible the step to perturbation is very simple. For the vast majority of the cases the default glitch primitives provided in new perturbation should be sufficient. In that case there is no need to write a perturbation program.

The only difference between a protocol used for acquisition and perturbation is that setting the verdict is optional for acquisition but required for perturbation. The verdict indicates whether the attempt returned a normal result (no fault, green), a successful result (fault successfully injected, red), or was inconclusive (error, reset, mute, random data, yellow).

The protocols included for Riscure training cards 2 and 3 demonstrate calculating the expected result based on the cipher key provided by the user and setting the verdict automatically. Training card 6 demonstrates how the use the protocol configuration to configure whether a glitch in the first check or in the second check should be considered successful.

5.3 Electrical perturbation

Inspector is capable of introducing electrical perturbations using the VCGlitcher device. For smart card targets, the VCGlitcher can assert perturbations to the power supply line or to the clock line using the internal card reader interface. For other targets, the VCGlitcher can be used for generating the pattern and timing of the perturbation.

5.3.1 FI setup

The software that controls the VC Glitcher is split in two parts:

- Perturbation module
- Perturbation program

The perturbation module is the software that runs in Inspector. This module looks similar to Inspector acquisition modules. The module is responsible for:

- Defining the perturbation parameters (e.g. the duration of a glitch) that will be tested in a test run.
- Defining the APDUs that will be sent to the smart card. If required, part of the APDUs can be randomized.
- Setting the glitch pattern.
- Setting voltages

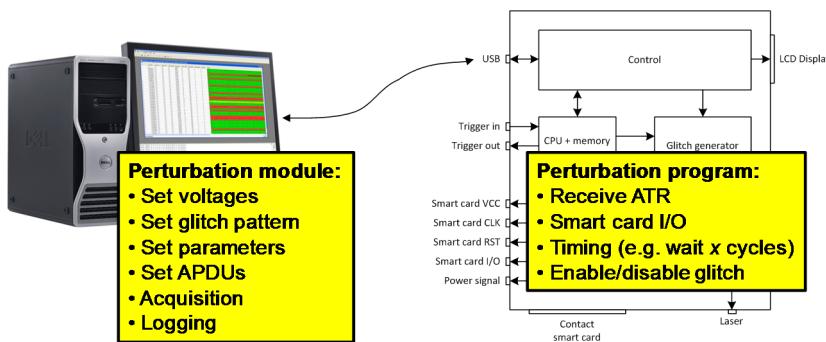
- Starting the perturbation program
- Storing results in a database

All perturbation modules also contain a perturbation program. This program is compiled by Inspector FI and loaded in the VC Glitcher. The program itself is executed on the CPU inside the VC Glitcher. It is responsible for:

- Communication with the smart card
- Timing
- Enabling/disabling the glitch pattern

When the program is finished, control is given back to the module. The module will then set parameters for the next iteration and restart the perturbation program.

Figure 5.5. Perturbation module vs. perturbation program



5.3.2 Triggered Perturbation

The Triggered Perturbation modules allow you to use the VC Glitcher with another IO device (for example, a contactless card reader or embedded target). In this setup, the VC Glitcher is triggered by the IODevice to generate the actual glitch pulse.

Example modules are provided to familiarize yourself with the capabilities of Triggered Perturbation modules.

The user interfaces of the example modules are similar to those of the Perturbation modules, please refer to Perturbation Modules for more information. There are some differences which are discussed in this section.

- The Device tab

Similar to SCA acquisition you select the IO device that is used to communicate with the target. The *TriggeredPerturbation* modules can use any device, as long as the device can send a trigger to the VC Glitcher

- Reset Modes

The Triggered Perturbation modules translate the reset modes to the following behaviour:

- no reset - No reset is performed in between perturbation attempts.

- warm reset - The perturbation module calls the reset method of the IO device (this may be similar to a cold reset, depending on how the IO device implements the reset).
- cold reset - The perturbation module calls the power down and power up methods of the IO device.
- Parameters

You can use the RF field strength as a glitch parameter when you combine the Triggered Perturbation modules with the MP300 contactless card tester. Contactless cards derive their power supply from the RF field. The target may choose to disable countermeasures in low power conditions, or otherwise be influenced by varying field strengths.

Optical perturbation

The Triggered Perturbation modules also available for the Diode Laser Station. The example modules extend the module "TriggeredOpticalPerturbation", which adds controls for the DLS components. The example modules use a similar user interface as the example modules for SmartCardOpticalPerturbation

VC Glitcher glitch modes

In triggered perturbation modules, the VCGlitcher has no control over which parameter is glitched, i.e. "Clock" or "Vcc" glitching is not applicable. The Perturbation tab will therefore not show the "CLK" glitch mode.

5.4 Dynamic perturbation

The Perturbation2 modules are technically complete rewrites of the "Classic" Perturbation modules. The code is not interchangeable. However, the concepts remain largely the same. In this section we will provide more information about Voltage and Clock glitching in the new *Dynamic Perturbation* framework. Laser glitching will be dealt with in Section 5.5, "XY(Z) Perturbation (Optical/EM-FI)".

5.4.1 Conceptual overview

Module categories

In the "Perturbation => Voltage/Clock" menu there are three module categories. For each module in these categories we will briefly discuss their applicability, but more details on the specific modules can be found in Section E.12, "Perturbation2". Note that many perturbation modules overlap in functionality, so not all of Inspector's system modules are described in the appendix.

- **Smart Card modules**

The smart card modules can be considered the most basic modules, since there are a limited number of standard protocols. The module "SC Perturbation" performs a glitch attack after sending the APDU command to the card. The goal is to influence the process triggered by the command, for instance a cryptographic operation. When the response of the card does not match the expected output (either a wrong crypto result or an abnormal status word) it is called a successful perturbation attempt.

In contrast, the module "SC Perturbation after reset" aims at the initialization phase of the target. The smart card will always send an Answer-To-Reset (ATR) message. When a voltage/clock glitch is successful it can cause the target to release more data than the ATR message. These two modules work with the standard behavior of the VC Glitcher, in combination with the GUI.

When more detailed control is required, the module "SC Perturbation with Glitch Program" can be useful. The user can write his own glitch program, where he can specify for example multiple glitches based on the actual timing of the device. Lastly, the module "SC Perturbation after RST with Glitch Program" is a combination of the previous two.

- **Embedded modules**

There are two system modules shipped with Inspector that support embedded targets. The "EP Perturbation" module is the embedded variant of the "SC Perturbation" module. It performs a glitch at a certain moment after sending the command to the target device.

The module named "EP Perturbation after Reset" performs an attack on the start up stage of the target. After the reset line is asserted an embedded device will typically send a boot sequence message to indicate its status. This module enables the user to perform a glitch attack on this initialization stage.

- **Advanced modules**

Just like in the smart card category the user has the option to specify his own glitch program for the VC Glitcher.

5.4.2 Communication protocols

Inspector has a number of built-in protocols. For example, for each training card that is supplied with Inspector a protocol is present. On the Target tab of the perturbation modules the protocol can be selected. The user can define dedicated protocols for his own target card/device.

Communicating with a device (or smart card) typically goes through a number of stages. As of Inspector 4.6 these stages are called *phases*. For instance, after a power up event the ToE will generally send out an ATR or boot string. This can be referred to as the "Power Up Phase". Then a command can be sent to the target (the "command phase"), after which the glitch should be performed. The user can indicate that he is interested only in the command phase, by selecting "trigger phase = command phase" in the module.

Note



The term "trigger phase" can be confusing for users with a background in electrical engineering. A better name would be *arm* phase. Typically a trigger is generated (either by Inspector or by the target itself) at each event, like a power up, a reset, or a command. Only when the specified phase is entered does Inspector arm all "armable" devices.

A *command* is a byte array that is sent to the card. Note that byte data types in java are always signed, so a cast might be necessary when the sign-bit is 1. For example, the statement

```
byte[] cmd = {(byte)0xa0, 0x01}
```

needs a cast for the first byte value. A device can return a response message. All the sent and received data is stored in the trace set, as well as in the 'data' column of the perturbation log.

The successfulness of a perturbation attempt is often determined by the response of the target. Especially for smart cards the typical "90 00" status words at the end of the response are an indication of a "normal" attempt. If the response differs from the expected data we call it a "successful" attempt. Such a *verdict* is indicated in the perturbation log with colors. When the user defines his own custom protocols (described below), he can program his own verdict determination algorithm.

The "New module wizard" option in the File menu can be used to generate a custom protocol class. We refer the reader to Section 9.8, "How do I implement an acquisition2/perturbation2 application protocol?", for more information about creating new protocols.

5.4.3 Sniplets and Programs

The default behavior of the VC Glitcher is rather basic. For smart cards the user can specify a number of wait cycles during which no glitch occurs, and a number of glitch cycles during which a glitch pattern is exercised. The glitch pattern is a number of <offset, length> pairs, expressed in 2ns units. However, for embedded targets the glitches cannot be synchronized with the clock cycles. This means that no wait cycles and glich cycles can be defined. Instead, a number of <offset, length> pairs is assigned, with the first offset value relative to the trigger event and subsequent offset values relative to the previous glitch end.

This behavior is very powerful, but may not be good enough for advanced users. It is possible to write a custom perturbation program that enables more detailed behavior. Such a program is actually a java class. There is a second type of perturbation program, called a perturbation *snippet*. The difference between the two is that multiple snippets can be used in a single perturbation run.

A custom perturbation program can be created with the "New Module Wizard" (under the file menu). On the first screen of the wizard the user is asked for the company name and the program name. It is important to specify a name that respects the Java coding standard, since it will be used as a class name (so start with a capital letter and include no spaces). The generated code will contain working VC Glitcher instructions that the user can modify. More information can be found in Section E.12.3, "SC Perturbation with Glitch Program" and Section E.12.4, "Perturbation Advanced Program".

5.4.4 Perturbation variables

During a perturbation run multiple parameters can be varied over a range, selected randomly (between a range), or kept constant. Such parameters include the glitch voltage, number of wait cycles, et cetera. See Section E.11.1, "Perturbation Module" for a more detailed description of the basic perturbation variables. (Note that the referred section covers the classic (legacy) framework, though the information is still useful.)

The user can also define custom variables in his own perturbation programs or snippets. These variables will always be interpreted as integers. They will automatically appear on the Perturbation tab.

5.5 XY(Z) Perturbation (Optical/EM-FI)

Apart from voltage and clock glitching, the operation of a chip can also be perturbed externally. One method of doing this is Optical FI, in which a laser beam is fired at a die. Another method is EM-FI, in which an EM signal is used to disturb the operation. Both methods are collectively

called XYZ Perturbation. In this section we will give some details on how Inspector makes both methods of XYZ perturbation available to the user.

The concepts of position calibration and the spanning of a scan grid are the same for both the Optical and EM-FI use cases. These concepts will be discussed in Section 5.5.1, “Coordinate Systems: XYZ Device vs. Chip” and Section 5.5.2, “Scan Area” respectively. A quick guide through the camera imaging options follows, which is only relevant for the Optical Perturbation scenario. Finally, in Section 5.5.4, “XY(Z) Devices” the use of the XY(Z) devices, such as the Diode Laser Station, Fiber Laser manipulator and EMPS is covered.

5.5.1 Coordinate Systems: XYZ Device vs. Chip

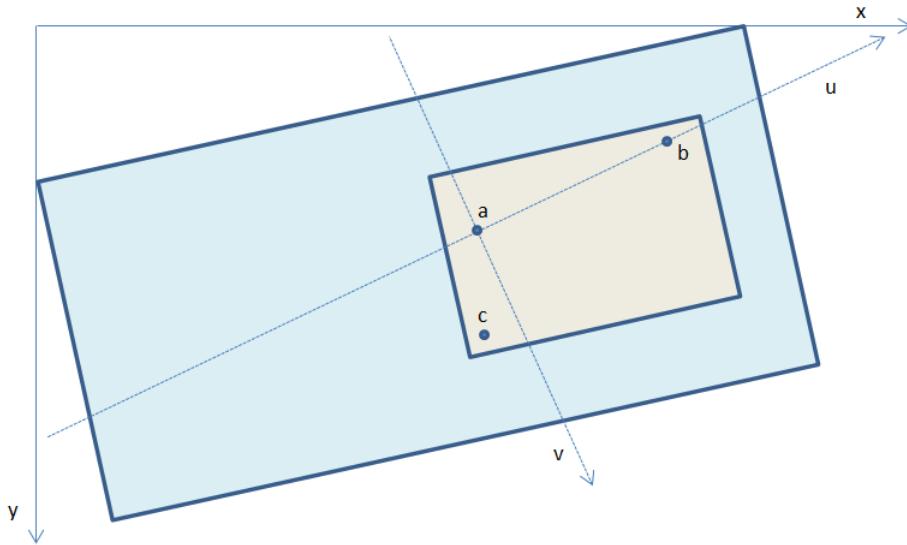
The effect of the glitch pulse (laser beam or EM pulse) is primarily dependent upon intensity, timing, and location. We will now focus on the location, and more specifically, on the coordinate system that is used. A major issue with these kinds of experiments is reproducability: Finding the right spot for the glitch pulese is a tedious job, finding it *again* (after re-inserting the card) is doubly so. To attack this problem we relate all positional measurement data to the reference frame of the chip. This means that we have to apply a transformation from the coordinates of the physical setup (like the DLS) to the reference frame of the chip. This transformation is semi-automatic: the user has to calibrate the coordinate system at the start of each session.

But first let us look at the two different coordinate systems. First we will look at the 'absolute' coordinate system used (internally) by the XY(Z) devices. Inspector has a generic position implementation where the 2D setups (commonly referred to as "XY stage", of which the Diode Laser Station is an example) internally make use of 3D coordinates with $z=0$. This third dimension is therefore transparent for the user if the hardware doesn't support it. We will explain the concepts in terms of the XY stage, when possible. Only when the third dimension is relevant will we address it. The third dimension is supported by the Fiber Laser manipulator and the EMPS.

The glitch source setups allow both relative and absolute movement. Relative movement means relative to the current position. Absolute movement means moving to a location in the coordinate grid. The origin of this coordinate grid is typically the position of the motors when the device was switched on.¹ The direction of the axes is defined by the orientation of the XY(Z) device to which the glitch source is attached. The unit of movement can be set to either μm (micrometer) or minimal motor step. The default unit is μm , but this is only relevant when the user has specs of the chip lay-out. In this manual we will generally refer to position data as unitless values.

The second coordinate system of the chip we'll look at is in fact defined by the user. By identifying three (visually notable) reference points on the chip the origin and axes can be set. See Figure 5.6, “The Tango and chip coordinate systems” for an example of a target card on the table. We see the (horizontal) x-axis and the (vertical) y-axis of the XY stage. Note that the origin is defined by the arbitrary position of the motors when the device was switched on—the origin is therefore different in each session. The target card is placed on the table at a random angle. Since we regard the setup as an XY stage we assume that the card lies flat on the table. This is a safe assumption when using a fixed laser like the DLS: the height of the target doesn't matter for the laser source. In case of the fiber laser the third dimension *is* important, because the fiber has to be placed directly above the target. Small height differences can result in either missing the intended target location (when the fiber is too high), or damaging the chip surface (when the fiber is too close). We will return later to the XYZ setup.

¹Though this might be device specific. In this manual we assume the user has the Tango table that Riscure provides as standard equipment. However, we try to describe the use of (and interaction with) the devices as general as possible.

Figure 5.6. The Tango and chip coordinate systems

The user has to identify three reference points on the target. In Figure 5.6 those reference points are marked as "a", "b", and "c". In this example all three reference points are on the chip, but they could also be on the edges of the card, provided that the range of the table/manipulator motors covers those positions.² The reference points now define the reference frame of the chip:

- The horizontal axis (referred to as vector "u") runs through points "a" and "b"
- The origin $O = (0, 0, 0)$ is point "a"
- The vertical axis (referred to as vector "v") is perpendicular to vector "u" and starts at point "a"
- (The upward w-axis intersects point "a" and is perpendicular to both other vectors)

Note that for the XY stage the reference point "c" is of little importance (only when you flip the chip will it result in a mirrored coordinate system). However, for the XYZ stage the third reference point is crucial: the v-axis should lie on the target surface, i.e. the plane that intersects the three (3 dimensional!) reference points. Using the notation " \times " for crossproduct and "(ab)" for the vector starting at point "a" and ending at point "b", we can formalize the three axes as:

$$v = (ab), w = (ac) \times (ab), \text{ and } v = u \times w$$

The new $uv(w)$ coordinate system is *normalized* with regard to the chip. This means that all position data will now be expressed relative to the reference points on the target—*independent* of the orientation of the target on the table. The major advantage of this is that the scan area of a previous experiment can be easily reconstructed in a new session. This normalized reference frame is transparent for the user, except that the controls still work in the directions of the device. This means that moving the joystick to the right will (probably) not result in just an increase of the u -coordinate, but also a small deviation in the v -coordinate (and the w -coordinate). We recommend that the user inserts the card in a standard way and assigns the reference points "a" and "b" on a horizontal line. This will avoid confusion later on.

²A possible use case for reference points on the edges of the card is as follows. Sometimes the user might want to fire at an interesting location of the target *from the back*. In order to find the exact location again, the reference points should be identifiable from both sides of the target.

5.5.2 Scan Area

The laser can be programmed to fire at a fixed location or at a series of locations. The fixed location is rather straightforward (but remember that the coordinates are in the normalized reference frame of the target). In this section we will focus on the *scan area*. This area is defined by three corner points. Additionally, the user must specify the number of horizontal and vertical stops. Together, the corner points and the number of stops define the series of laser locations.

The scan area is marked by three corner points: North-West (NW), North-East (NE), and South-East (SE). There is no technical limitation on the relative orientation of these points, but we advise the user to maintain an intuitive grid layout with "east" on the right hand side of "west" and "north" above "south" (relative to the camera view). Compared to Perturbation1 the scan area does not have to be "in line" with the coordinate axes (i.e. only horizontal and vertical). Neither does it have to be of rectangular shape. This means that the user can specify an arbitrarily rotated parallelogram.

An example is shown in Figure 5.7(left). The scan area is marked with the three points "NW", "NE", and "SE". Clearly, the quadrangle is not straight—neither in the reference frame of the table, nor in the normalized frame of the target. The corner points are expressed in the normalized coordinate system. We also see that the number of horizontal stops is 5 and the number of vertical stops is 4. This means that there will be a series of 20 laser locations, one at each intersection of the scan area grid.

On the right hand side of Figure 5.7 the scan area is shown in the reference frame of the card. The horizontal axis is called "u"; the vertical axis is referred to as "v". Suppose that the coordinates of the corners are as follows: NW = (20, 5), NE = (40, 17), SE = (37, 32). Mind that these coordinates are in the normalized coordinate system of the card and have no relation to the orientation of the table. The generated laser positions will then be interpolated (see Table 5.1, "Calculated scan positions").

Figure 5.7. The scan area within both reference frames

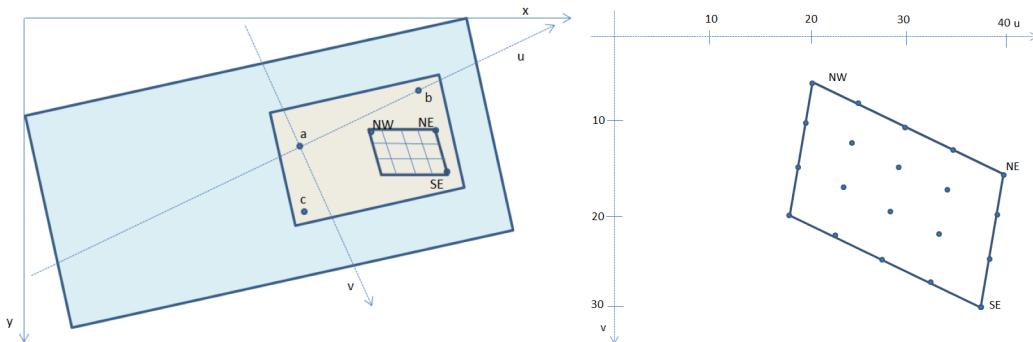


Table 5.1. Calculated scan positions

| | | | | |
|---------|---------|---------|---------|---------|
| (20,5) | (25,8) | (30,11) | (35,14) | (40,17) |
| (19,10) | (24,13) | (29,16) | (34,19) | (39,22) |
| (18,15) | (23,18) | (28,21) | (33,24) | (38,27) |
| (17,20) | (22,23) | (27,26) | (32,29) | (37,32) |

XYZ-stage

In the above example we used the XY-stage, ignoring the z-axis. How does an XYZ-stage influence the process? When the user identifies the reference points "a", "b", and "c" with the fiber laser manipulator the z-coordinates are also recorded. The normalized u- and v-axes per definition lie in the plane defined by the reference points. The card is always assumed to be flat, so every point on the chip will have $w=0$. This means that the picture on the right hand side of Figure 5.7 still remains valid: the corner points and thus all scan positions will have $w=0$.

5.5.3 Microscope

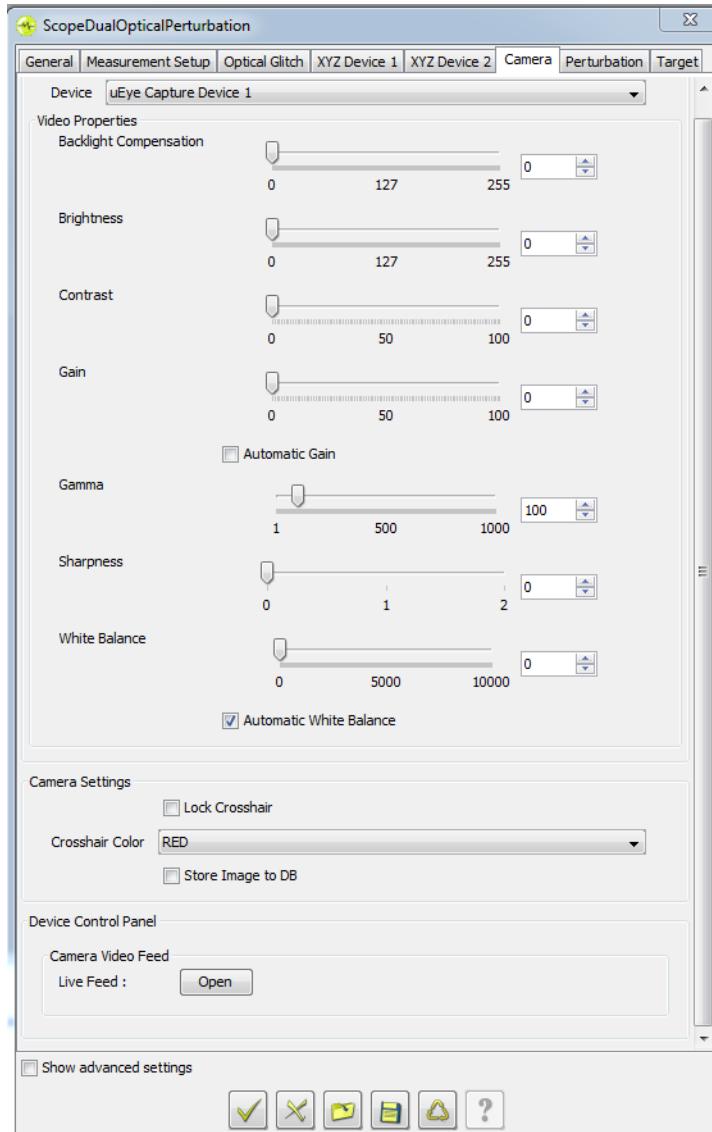
Note



This section is only relevant to the Optical FI use case.

The exact location of the laser shots requires a high precision. It is therefore necessary to use a microscope. The DLS, which is supplied by Riscure, comes with a camera, which relays live images to Inspector. This provides a continuous visual feedback for the user and also allows for the storage of an image when the user assigns reference points and every time the laser fires.

The DLS comes with three different objectives (5x, 20x, and 50x magnification), which influences both the laser spot size and the image size. See Section 6.4.2, “Diode Laser Station” for a more detailed description of the DLS.

Figure 5.8. Camera tab

Inspector supports any USB camera that is plugged in. However, it is important to install the appropriate device driver first. The driver for the UEye camera (which is part of the standard DLS setup) can be found in the drivers\Win32 folder. Due to the generic nature of Inspector's device management, any USB camera that is connected will be selectable in the camera option list.

There are a number of camera settings which the user can change. Inspector should automatically detect which options are available for camera which is connected; Unsupported options are not shown. The UEye camera supports user control for backlight compensation, brightness, contrast, (automatic) gain, gamma, sharpness, and (automatic) white balance.

The live view can be enabled by pushing the "Camera view" button. The camera images will show up in a separate window, which will always be on top. The camera window is resizable. When the "Camera view" button is pressed again the camera window will disappear.

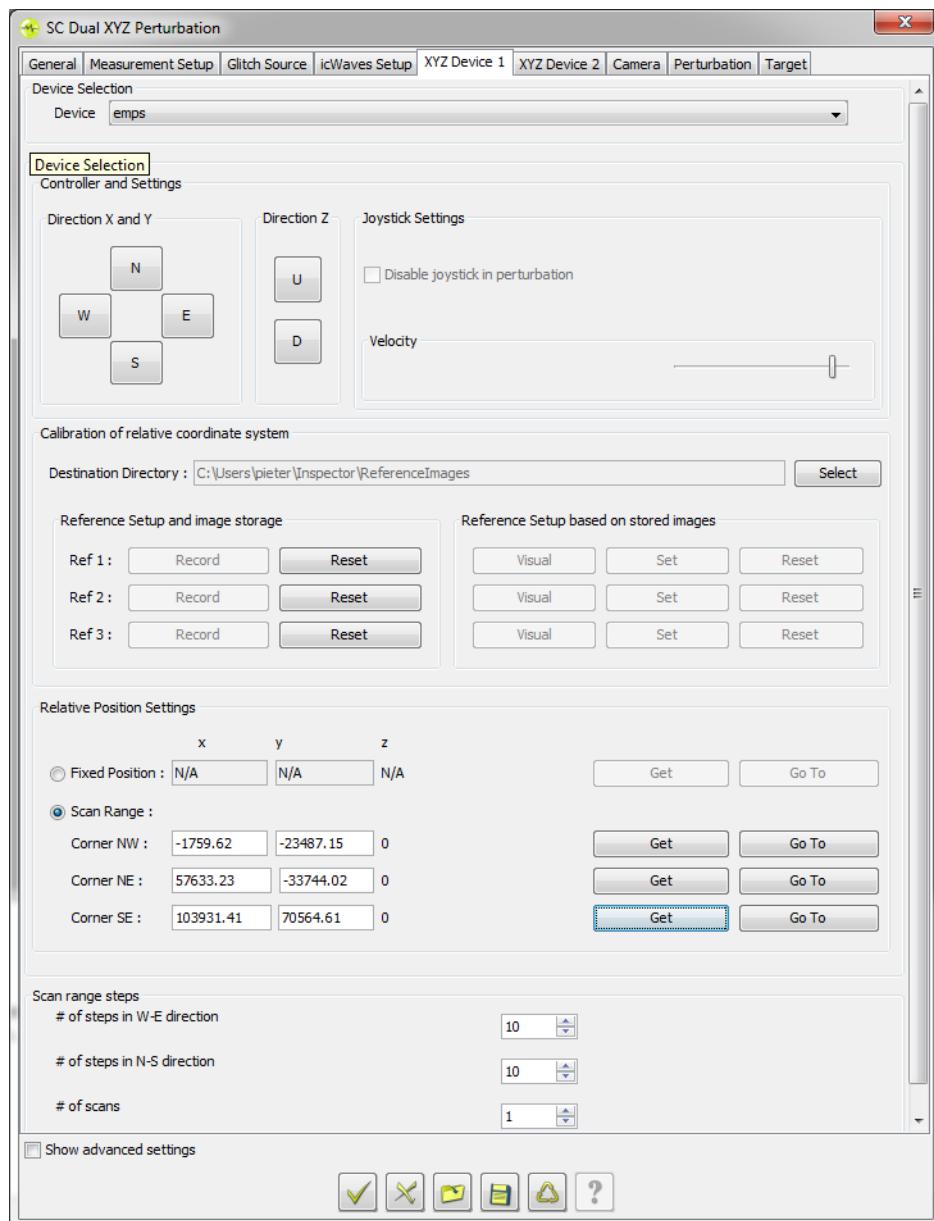
The camera view has a cross hair image overlayed on the camera view. The cross hair indicates the laser position. Calibrating the cross hair is done by test-firing the laser and clicking on the position of the laser spot. To avoid accidentally changing the location of the cross hair (by unintentionally clicking on the image) the user can lock the cross hair by use of a checkbox.

The user can select a number of colors for the cross hair, which can increase the visibility depending on the background of the chip.

5.5.4 XY(Z) Devices

This tab represents the interface with the XY(Z) devices. From here, the user can control the movement of the table (or manipulator), assign reference points on the chip, store and view images of those reference points, calibrate the coordinate system, and define a scan area. In this section we will walk through all the subpanels on this tab, thereby following the general workflow.

Figure 5.9. XYZ tab



First the user has to select an XY(Z) device from the "Device Selection" option list. The options in the list are automatically generated according to the XY(Z) devices currently connected with Inspector. Sometimes the device will be set as the previously used one. If the device does not show up in the option list then please check whether the XY(Z) device is power on and connected to PC properly. At least all Tango XYZ devices should be recognized by Inspector in a Plug and Play fashion. Other XY(Z) devices can be linked to Inspector from the Inspector hardware manager.

After the connection has been made, the "Device Control" panel is enabled. An XY(Z) device can be steered with either the physical controls (typically a joystick) or with this software control panel. The two buttons for Up and Down are disabled if the selected device does not support those directions. There is also an option to adjust the movement speed to all directions. We strongly recommend to keep "disable joystick in perturbation" checked as the

default situation. The reason is that the user should not be allowed to interfere with the controls during an experiment.

In the "Calibration panel" the user can set a "Destination Directory" to store reference position images. The default directory is {user}\Inspector\ReferenceImages\. If there is more than one XYZ Device tab in the module, the user has to select the same "Destination Directory" in both tabs. In the "Reference Setup" panel the "Record" button will first set the current XY(Z) Device position as a reference point. Additionally, Inspector will store the current image to the destination directory for calibration in the future. In the "Calibration Functions" panel the "Visual" button will open a recorded reference position image. The purpose of the "Set" button is to tell Inspector that the current XY(Z) device location is the reference position.

The "Relative Position Settings Panel" will be enabled when:

1. The user just set up and recorded three reference positions;
2. The user just finished a calibration(the "Set" button for Ref3 is clicked);
3. Inspector has knowledge of the previous setup by examining the Destination directory and there is no calibration or reference setup needed. (Used for situations where no changes have been made to the physical setup).

Please refer to Section 5.5.1, "Coordinate Systems: XYZ Device vs. Chip" for more background on the reference position set up and calibration details.

Once the "Relative Position Settings" panel is enabled, the user can start to setup a relative scan position or area. In Measurement Settings the user can set the number of measurement stops that need to be made in the directions W to E, and N to S.

Note



The relative plane created during calibration is two dimensional. A consequence of this is that the "Get" buttons in the "Relative Position Settings" panel project the position of the XY(Z) device onto the z=0 plane of the relative coordinate system.

6 Inspector Hardware Components

This chapter contains the finer details of all the hardware components that can be supplied and controlled with Inspector. It describes the function of the individual components and if applicable, software details and the specifications regarding that device. The components have been categorized by their primary use function and are referenced only once. For example: the primary function of the Power Tracer is to obtain the power profile of the target, while it also provides target control.

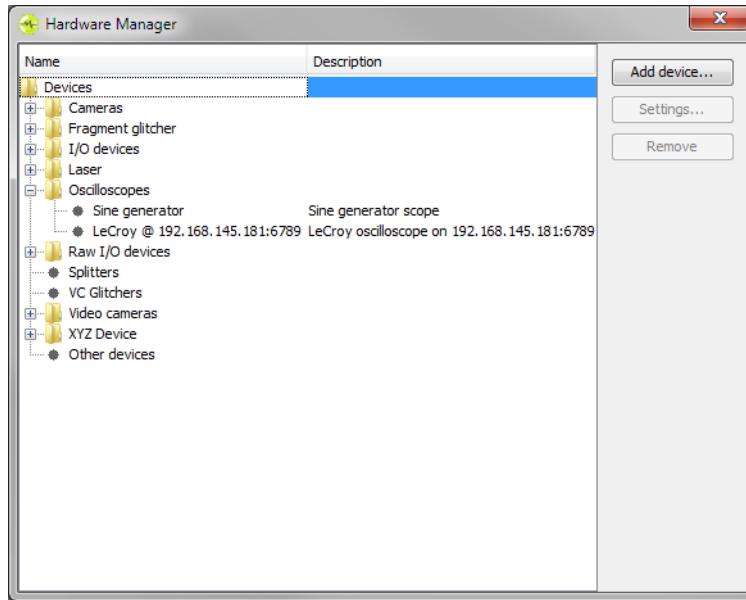
6.1 Hardware Manager

All of the hardware components used by Inspector in an Acquisition2 measurement setup are controlled using the Hardware Manager. While most of the devices are automatically detected and configured, some may need manual configuration or modification of the default parameters. For example: the baud rate of a serial interface, or the IP address of an oscilloscope. This is done using the Hardware Manager dialog.

6.1.1 Overview

The Hardware Manager can be started by selecting the option from the Tools menu. The hardware manager dialog will be shown:

Figure 6.1. Hardware Manager dialog

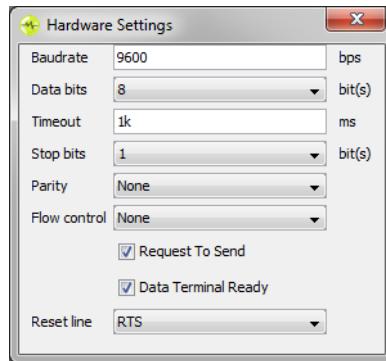


The dialog shows the different categories of hardware recognized by Inspector. The devices detected by Inspector are listed under the appropriate category. The list also shows a device description containing the principal setup information for the device (such as port number, IP address, etc.). Devices for which the category is not known are displayed as "Other devices". This typically includes new device classes created by users. See Section 8.6, "Developing device drivers" for further reading on the development of custom device drivers.

6.1.2 Settings

Device settings can be changed by selecting the device in the tree overview and clicking the Settings button. A device specific settings dialog will appear allowing settings to be changed.

Figure 6.2. Example device settings for serial ports

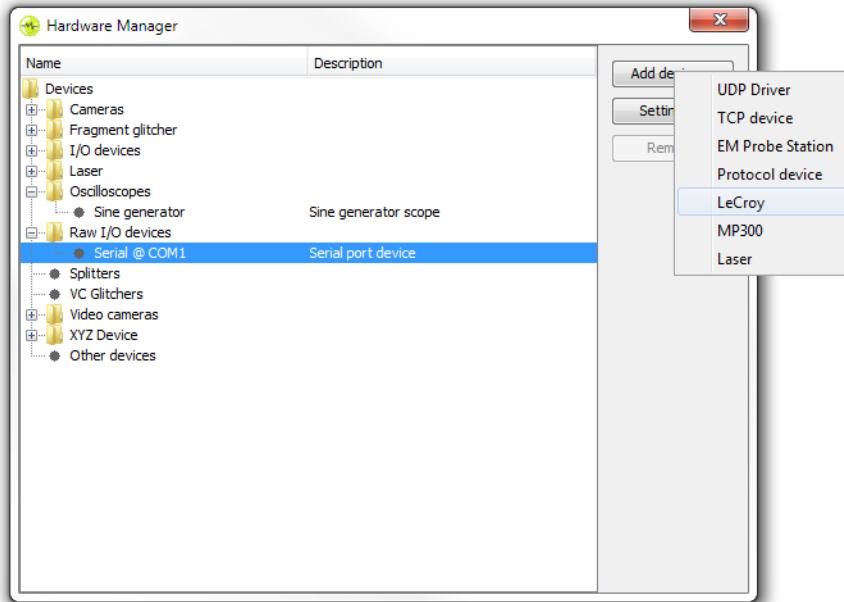


Not all devices have settings which can be changed through Inspector. For such devices the Settings button will remain disabled when selecting the device.

6.1.3 New devices

Devices are added automatically to the hardware manager when they are detected as connected. Likewise devices which are disconnected will be removed from the overview. However not all devices can be automatically detected. E.g. devices which do not support Plug-and-play, devices which require configuration before being usable, and network attached devices will have to be added manually.

Devices can be added manually by clicking the Add device button. A list of supported devices will then be shown. Clicking the appropriate device will pop up a driver specific configuration dialog showing the parameters necessary to set up the device. For network attached devices this would typically include IP address and port number.

Figure 6.3. Add devices

6.2 Oscilloscopes

One of the main features of the Inspector platform is controlling your oscilloscope and managing large signal acquisition sessions. Of course the function of the oscilloscope is the acquirement of side channel signals from your device under test. It is therefore an essential part of the equipment. Currently the two main oscilloscopes that are supplied with Inspector are:

- LeCroy WaveRunner 104Xi
- PicoScope 5203

The PicoScope is a USB oscilloscope of a good basic quality, that can be easily installed by installing a driver. The LeCroy oscilloscope is the superior quality network oscilloscope. A server is installed on this oscilloscope before delivery to enable communication with the Inspector software. The configuration of this scope is detailed later in this section.

In addition to these oscilloscopes, as of Inspector 4.2 we provide experimental support for IVI (Interchangeable Virtual Instruments) compliant oscilloscopes. This driver can be used to interface with a wide range of oscilloscopes, as shown in the driver registry [http://www.ivifoundation.org/registered_drivers/driver_registry.aspx] from the IVI foundation [<http://www.ivifoundation.org>].

6.2.1 Setup and use of the LeCroy

The exact way of connecting and using the oscilloscope is dependent on the application and the other side channel devices that are used in the setup. This section describes the setup of the LeCroy oscilloscope

The setup of a LeCroy oscilloscope for use with Inspector consists of the following steps:

- Network configuration
- Configuring the oscilloscope

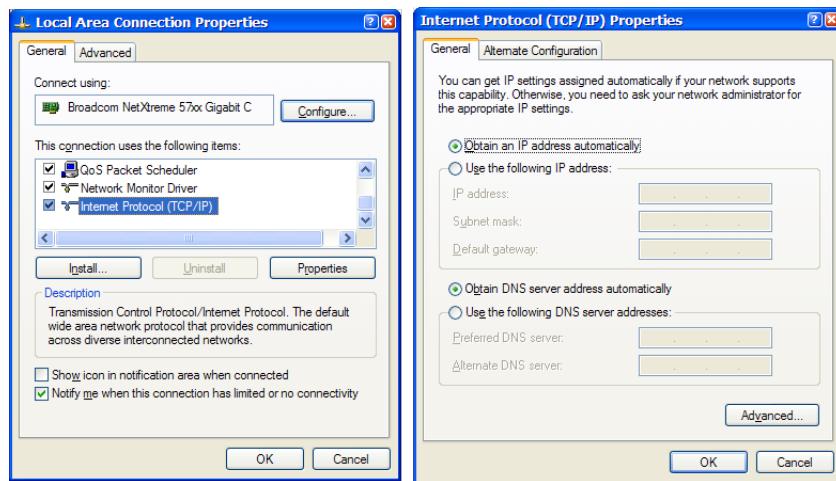
In order to use the LeCroy oscilloscope in Inspector, the Inspector workstation must be able to communicate with the oscilloscope over the network.

Office use

For office use, it is likely that both devices obtain an IP address from a DHCP server. Make sure that both devices will obtain an IP address from the DHCP server. As LeCroy oscilloscopes run Windows XP, the configuration is similar:

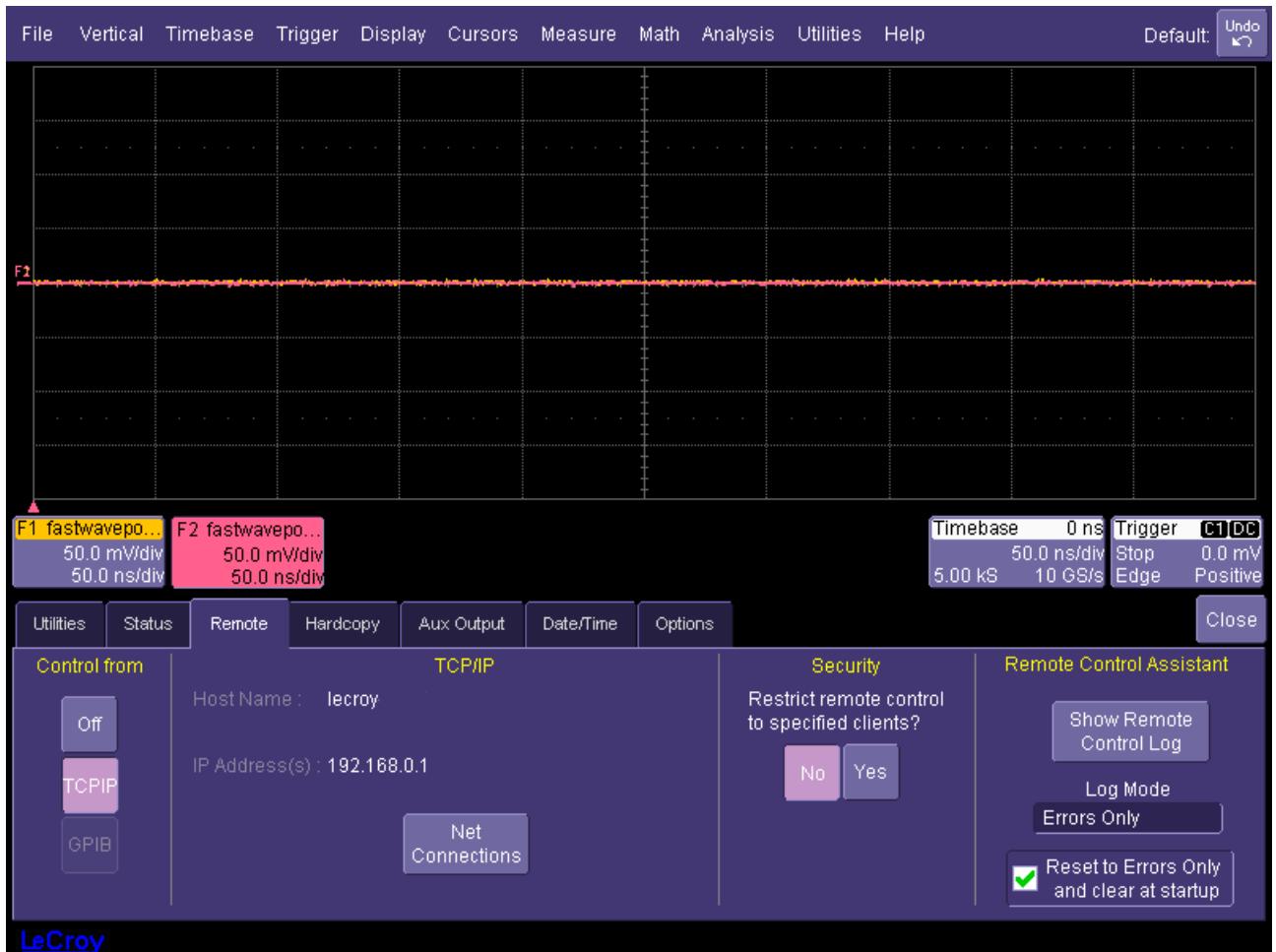
- Open the *Windows Explorer*
- Right click My Network Places and click *Properties*
- Right click the network interface (e.g. Local Area Connection) and click *Properties*
- Click *Obtain an IP address automatically* and *Obtain server address automatically*, as shown in the figure below.

Figure 6.4. DHCP configuration on the oscilloscope



The IP address obtained by the oscilloscope can be viewed using the oscilloscope's software:

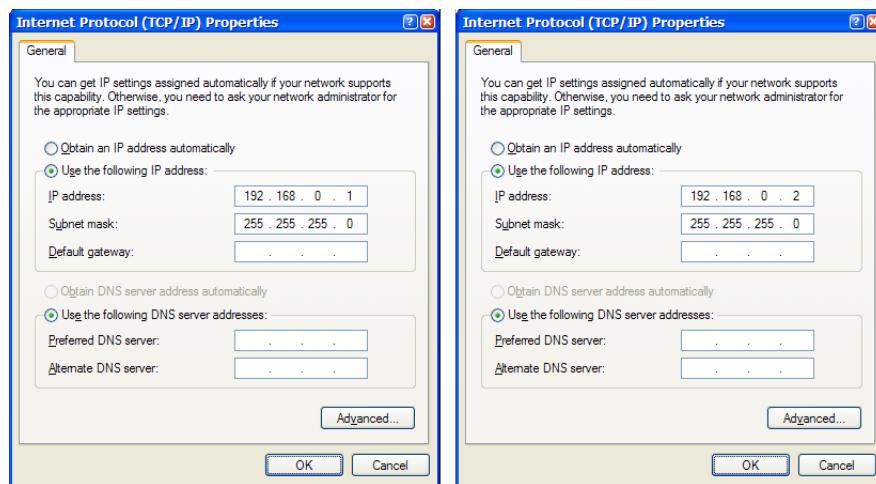
- Open the Utilities menu
- Click the Remote tab
- The current IP address is shown in the panel

Figure 6.5. The IP address on the oscilloscope

Mobile use

For mobile use of the setup or in locations where no DHCP server is available, it is likely that the workstation and the oscilloscope are connected using an Ethernet crossover cable or a simple switch. In this case, assign an IP address to both devices manually. As LeCroy oscilloscopes run Windows XP, the configuration is similar:

- Open the Windows Explorer
- Right click My Network Places and click Properties
- Right click the network interface (e.g. Local Area Connection) and click Properties
- On the oscilloscope, set the IP address to 192.168.0.1. On the Inspector workstation, set the IP address to 192.168.0.2. The subnet mask should be set to 255.255.255.0 on both devices. The figure below shows the configuration of both devices.

Figure 6.6. Manual IP settings

Inspector connects to an oscilloscope using the host name **LeCroy**. The host name needs to be resolved into an IP address and the Inspector workstation does this by querying a local hosts file or a DNS server. In an office network, this name should be configured in your DNS server (ask your network administrator to do this). Alternatively, the host name can be added to the 'hosts' file of the workstation. This is also needed for mobile use. The following steps show how to add a host name to the hosts file:

- Click Start -> Run..
- Enter **notepad c:\windows\system32\drivers\etc\hosts** and click the OK button
- Add the following line to the end of the file. Change the IP address if necessary.

192.168.0.1 lecroy

- Save and close the file
- In order to test if the network is setup properly, try to ping the oscilloscope from the Inspector workstation:
 - Click Start -> Run..
 - Enter **cmd** and click the OK button
 - Enter the command **ping lecroy**. The output should be like this:

```
C:\Users\Inspector>ping 192.168.0.1
Pinging lecroy [192.168.0.1] with 32 bytes of data:
```

```
Reply from 192.168.0.1: bytes=32 time<1ms TTL=64
```

```
Ping statistics for 192.168.0.1:
Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Configuring the LeCroy

First of all, make sure that the optional LeCroy XDEV customization package is installed on the oscilloscope. This package is required in order to use the server program, as it uses the FastWavePort feature of this package. More information on this package is available at: <http://www.lecroy.com/tm/Options/Software/XDEV/default.asp>

Inspector must be able to communicate with the LeCroy oscilloscope. Therefore, Riscure developed a small server program that must be executed on the oscilloscope. This installer for this program (i.e. *inspector_server_lecroy.exe*) is located on the Inspector installation CD.

After installation, the server program must be started by double clicking the *Inspector Server* icon on the Desktop. This must be done before using one of the acquisition modules in Inspector.

A command window containing the following information will be shown.

```
Inspector server for LeCroy oscilloscopes 1.2
Copyright 2007, Riscure BV
Created instance of Lecroy.XStreamDSO
Successfully retrieved dispatch pointers to required objects!
Changing settings:
* Setting Smart memory setting to "Fixed sample rate"
* Enabling vertical variable scale on channel 2 (Inspector channel A)
* Enabling vertical variable scale on channel 3 (Inspector channel B)
* Setting horizontal origin to 0
Successfully initialized FastWavePort system
Allocating 24MB memory for each channel... ok!
Oscilloscope initialized!
FastWavePort thread on channel 2 (Inspector channel A)
started!
FastWavePort thread on channel 3 (Inspector channel B)
started!
```

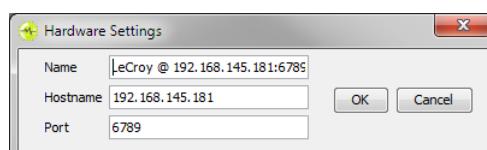
Finally, the oscilloscope must be configured to send traces to the server program. This can be done by double clicking the *Configure for Inspector* icon on the Desktop. This step is only needed if the scope settings are changed manually on the oscilloscope.

Inspector can now use the LeCroy oscilloscope just as all other oscilloscopes. **Remember to use channels 2 and 3 of the LeCroy, as these channels provide a higher sample rate.**

Using the LeCroy oscilloscope in Acquisition2

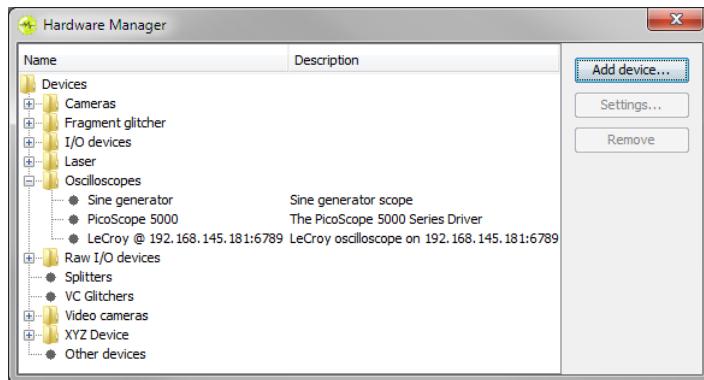
The LeCroy is defined via the Hardware Manager in the Tools menu. Select 'Add device' and select 'LeCroy'. You need to define the IP address of the oscilloscope, the port and name this LeCroy (e.g. by its IP address and port number). Please note that you should match the IP address of the LeCroy to the PC by configuring your Local Area Network - they should be in the same subnet.

Figure 6.7. Hardware Manager: add new device



After completion, you will notice that the LeCroy is shown in the oscilloscope overview of the Hardware Manager.

Figure 6.8. Hardware Manager: device overview



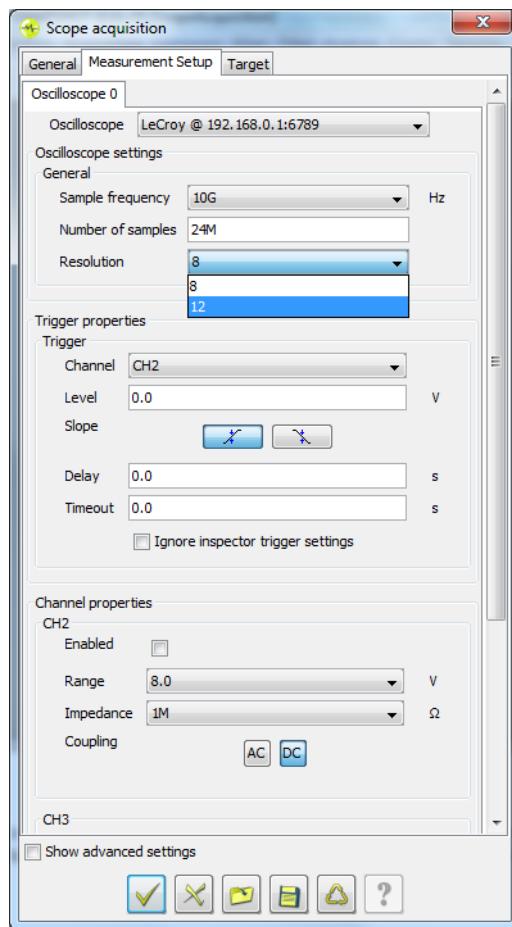
When you start the ScopeAcquisition module and select the LeCroy device on the Measurement Setup tab, you will see a checkbox labelled 'Ignore inspector trigger settings'. If you enable this option Inspector will not push the dialog settings to the scope; these are subsequently ignored. This is useful when you apply trigger settings to the LeCroy device yourself (outside Inspector). This checkbox can be seen in Figure 6.9, "Acquisition2: Scope acquisition with LeCroy".

In this figure it is also possible to see the 'Resolution' property under the "General" settings. This setting can be changed to match the specific characteristics of the LeCroy oscilloscope in use. Depending on whether an 8 or 12 bit resolution LeCroy is used, the value of the resolution setting should be set to 8 or 12 respectively. It is possible to set resolution to 8 for a 12-bits LeCroy. The impact of this will be that samples will be represented using 1 byte instead of 2, resulting in smaller trace sets at the cost of a less accurate representation of the actual values measured.

Note



Inspector will show a dropdown list of common 'Sample frequency' values. However, the software cannot verify whether the oscilloscope device actually supports the chosen value. For example a 12-bits LeCroy will not support the higher sampling rates. An invalid value will result in an error. Please verify that the selected sample frequency is supported by the oscilloscope.

Figure 6.9. Acquisition2: Scope acquisition with LeCroy

6.2.2 Setup and use of the PicoScope

The setup of the PicoScope can be performed by installing the appropriate driver. The driver can be found in the Inspector program folder "hardware\PicoScope\driver". It can also be downloaded from the website of Pico Technology [<http://www.picotech.com>] together with the application software. Note that the latest driver for the PicoScope requires the .NET framework.

6.2.3 IVI-compliant oscilloscopes

Experimental support for IVI-compliant oscilloscopes has been added to Inspector 4.2. The current support is based on using *driver specific properties* in acquisition modules in order to configure Inspector to use the appropriate instrument and measurement channels.

This section of the manual describes the software requirements that need to be met in order to use IVI-compliant oscilloscopes with Inspector, as well as the steps required to perform acquisitions using the IviScope drivers.

Software requirements

In order to use IVI compliant instruments on a computer, several software packages need to be installed first. As a first step, the user needs to install the IVI shared components [<http://>

www.ivifoundation.org/shared_components/Default.aspx] provided by the IVI Foundation [<http://www.ivifoundation.org>].

Once these components have been installed, the user needs to install a *compliance package* providing the base functionality for the required IVI modules. The current version of Inspector has been tested using the *IVI Compliance Package* from National Instruments available here [<http://joule.ni.com/nidu/cds/view/p/id/1420/lang/en>].

In addition to this, an instrument specific driver needs to be used. To find out which driver your oscilloscope needs, check with the device manufacturer or look up your device in the driver registry [http://www.ivifoundation.org/registered_drivers/driver_registry.aspx] from the IVI foundation. Beware that some of these drivers also have other requirements, so you might need to install other software packages as well.

The current implementation has been tested with LeCroy WaveRunner 104X oscilloscopes. We used a beta version of the LeCroyScope drivers obtained from LeCroy's support helpdesk. The version tested was LeCroyScope v3.2.4, and it is available through the 3rd party directory in the Inspector installation CD. However, for LeCroy oscilloscopes we recommend to continue using Inspector's drivers instead of the IVI interface, as they are stable and thoroughly tested.

The following versions for the different components have been tested, listed in order of installation:

- Ivi Shared Components 2.1.1
- NI VISA 5.0.3
- NI Ivi Compliance Package 4.1.0
- LeCroyScope drivers 3.2.4

Note that there is a limitation with respect to sample frequency settings in the current IVI interface of Inspector. According to the specifications, IVI scopes will find the closest possible sample frequency and number of samples to those requested by the user. Currently, Inspector assumes that the sample frequency provided in the acquisition dialog is actually the one the scope will use. Therefore, it is possible that the sample frequency recorded by Inspector is different from the actual sample frequency.

In those cases, it is possible to fix the sample frequency of the resulting trace set by going to the *settings* dialog and changing the X axis' scale field to the inverse of the sample frequency ($1/fs$).

Additionally, in the event of a trigger time out, the current IVI implementation of Inspector does not provide an output trace. Instead, Inspector will reject the trace and wait for the next trace. When this happens, *Forcing acquisition termination* is printed to the Log panel.

Note that for LeCroy scopes, the Inspector LeCroy server doesn't have to be running in order to be able to use the IVI interface properly. Also, you should manually enable the "Time base > Real Time Memory > Set Max Memory" option in the oscilloscope instead of "Set Max Frequency".

Instrument specification

As explained above, the current support for IVI oscilloscopes uses scope specific properties to set the different configuration options required. The first thing that the user needs to define is how to connect to the oscilloscope. To that end, IVI defines the concept of *driver sessions*.

A driver session is an association of a *hardware asset* and a *software module*. The former defines the instrument to be used and its address (IO Resource in IVI parlance). The latter defines the device-specific driver that implements the communication with the device and is typically provided by the oscilloscope vendor.

In order to define a driver session to be used with Inspector, we provide two alternatives:

1. Using an already existent *driver session* from the IVI configuration store.
2. Entering the corresponding *IO Resource* and *software module* for use with Inspector

For the first case, one simply defines the *ivi_name* property in the scope specific properties box of the acquisition module. Then, Inspector will look up the IVI configuration store and try to find a driver session with the provided logical name. This option is provided as a convenience option for users that have already configured their oscilloscopes with IVI using other software platforms.

In case the user has never used the IVI oscilloscope before on the Inspector workstation, the second option is preferred. In this case, the user has to provide Inspector with the following two properties:

- *ivi_resource*: Provides the I/O resource of the device. For instance, it could be a network address or a physical port.
- *ivi_driver*: The name of the instrument specific driver. For LeCroy oscilloscopes of the WaveRunner series this name is *lscope*.

Configuration options

Current acquisition modules allow to perform acquisitions using two different channels, and triggering using any of those two channels or an external trigger channel. However, IVI oscilloscopes support a different number of channels and name them in an oscilloscope-specific manner. Therefore, before using them with Inspector, the user needs to create a mapping between oscilloscope channels and the channels seen by Inspector's modules.

Additionally, several options can be defined. The following list shows the different options available in the current version of the IVI drivers, with the required settings emphasized using bold text:

- ***channelA***: Defines the name of the channel to be used as channel A by Inspector.
- ***channelB***: Defines the name of the channel to be used as channel B by Inspector.
- ***channelEXT***: Defines the name of the channel to be used as the external trigger channel by Inspector.
- *probe_att_a*: The attenuation of the probe used for channel A. For auto-sense probes, this value needs to be set to -1. The default value is 1 (i.e. no attenuation).
- *probe_att_b*: The attenuation of the probe used for channel B. For auto-sense probes, this value needs to be set to -1. The default value is 1 (i.e. no attenuation).
- *reset_scope*: Whether the scope settings will be reset during initialization or not. Defaults to true.

- *configure_trigger*: Instructs the driver to avoid configuring the trigger settings when set to false. Together with *reset_scope=false*; allows the use of oscilloscope-specific advanced triggering features. Defaults to true.

It must be noted that the IviScope specifications do not provide a way to select the input impedance of the external trigger channel. Therefore, the selected input impedance will be device-dependant and will depend on the actions performed during initialization. After reset, some oscilloscopes will set the input impedance to 50 Ohm and some will set it to other values (e.g. 1M Ohm).

If the user needs to use the external trigger channel with an input impedance different than the default, it is recommended to use the options *reset_scope=false*; *configure_trigger=false*; and configure the trigger settings manually on the oscilloscope.

Configuration examples

This section provides some configuration examples to illustrate how to use all these settings with Inspector. As a first example, we show a configuration that uses an already present driver session and configures the basic channel settings:

```
ivi_name=lecroy;
channelA=C1;
channelB=C2;
channelEXT=VAL_EXTERNAL;
```

In this case, we are using a driver session defined as lecroy. In addition, we indicate that channel A,B and external correspond to the oscilloscope's channels C1,C2 and VAL_EXTERNAL respectively. The latter is the default name for external trigger channels defined by the IviScope specifications.

In the following example, we define the same settings but now we use the driver name and the oscilloscope's IP address instead of using a driver session already present in the configuration store. Additionally, we do not reset the oscilloscope during initialization. This might be used to acquire measurements from virtual channels such as mathematical functions applied to the signals of physical channels.

```
ivi_driver=lcscope;
ivi_resource=VICP::192.168.1.2;
channelA=C1;
channelB=C2;
channelEXT=VAL_EXTERNAL;
reset_scope=false;
```

Finally, we provide an example which configures the oscilloscope to use an auto-sense probe in channel C1 (as channel A) and does not modify the trigger settings. By doing this, we can use trigger modes other than the standard edge trigger.

```
ivi_driver=lcscope;
ivi_resource=VICP::192.168.1.2;
channelA=C1;
channelB=C2;
channelEXT=VAL_EXTERNAL;
reset_scope=false;
configure_trigger=false;
probe_att_a=-1;
```

6.3 Side Channel Analysis

This section describes the hardware components that are used for target communications while measuring side channel information.

6.3.1 Power Tracer 3

Functional overview

Riscure has developed a dedicated smart card reader called Power Tracer. The Power Tracer has two features different from standard smart card readers:

- It can measure the power consumption of the inserted smart card.
- It can generate a signal to trigger an oscilloscope

Figure 6.10. The Power Tracer 3



- No card inserted: Off
- Card inserted, note powered: Blinking
- Card inserted and powered: On
- **Blue LED**
 - No acquisition in progress: Off
 - Acquisition in progress: On
- **Red LED**
 - Error free operation: Off
 - Error detected: On

6.3.1.2 Specifications Power Tracer 3

The Power Tracer is designed for power and EM analysis on smart cards. The low-noise EM measurement is achieved by:

- Electrical isolation between smart card and digital control
- EM shielded by closed aluminum housing. By moving the EM probe down towards the smart card the Power Tracer shields the EM probe tip

This version is shipped from 2008, with Inspector version 2.4.0. This version has many improvements, but is backward compatible with the 1.x version. Most notably, the external appearance has changed. The volume controls for gain and offset disappeared from the device, and are now implemented in configurable hardware. Furthermore two additional LEDs have been added for a more intuitive representation of internal states.

This version is shipped from 2008, with Inspector version 2.4.0. This version has many improvements, but is backward compatible with the 1.x version. Most notably, the external appearance has changed. The volume controls for gain and offset disappeared from the device, and are now implemented in configurable hardware. Furthermore two additional LEDs have been added for a more intuitive representation of internal states:

From a reliability point of view this new power tracer is more robust as the power supply to the smart card is stabilized and no longer dependent on the smart card's current consumption. With the insertion of a new card the analyst no longer has to tune the offset and gain to find a setting for which the card operates normally. The quality of the power signal of the new Power Tracer has improved resulting in a higher bandwidth and lower noise.

An overview of the main improvements:

- Various **Signal/Noise** ratio improvements:
 - Electrical isolation: the smart card is electrically separated from the control circuitry
 - Real current measurement instead of measuring resistor voltage drop
 - Shut down of external power supply during¹ acquisition

¹Dc/Dc will shut down after the oscilloscope receives a trigger. In order to keep a clean signal, using device delay instead of oscilloscope trigger delay is recommended.

- High bandwidth amplification (0 .. 1 GHz) delivers a stronger signal that can be sampled easier.
- Switching between **internal** and **external** clock
- Software **configurable gain** (100% .. 200%). This allows the analyst to tune the signal strength to the oscilloscope input range.
- Software **configurable signal offset** (-2V .. +2V). This allows the analyst to tune a part of the signal to both upper and lower bounds of the oscilloscope input range.
- Software **configurable card supply voltage** (1.8V .. 6.0V). This allows the analyst to operate at various power levels to experimentally establish an optimal signal/noise ratio of the leaked information.
- RS232 output to monitor or send I/O
- 'Clock in' input to run the smart card on other clock frequencies than 4 MHz

Known Issues

Issue: Sometimes a connection with the Power Tracer cannot be established under Windows 7 when using a USB hub.

Workaround: Unplug the Power Tracer from the USB hub and then plug the Power Tracer directly into the computer.

6.3.1.3 Software interface Power Tracer 3

The controls for gain and offset serve to tune the amplitude and DC offset of the output signal. By tuning this the analyst can optimize the resolution of the oscilloscope by fitting the scope window to the signal part under investigation.

WARNING: the offset and gain controls can be set to values outside the defined operating range of smart cards, which will cause a card to refuse communication. It is best to tune these controls with a few repeated single trace acquisitions, where the gain should start minimal (control fully turned left), and offset should start at 0 (control set in middle position), and the oscilloscope should initially be set to a range of 1 Volt.

The Power Tracer comes with a driver that implements the T=0 and T=1 transmission protocols. A number of properties can be changed using the '*setDeviceProperty*' method in a sub class of SideChannelAcquisition, or set in the '*Device specific properties*' box of the acquisition dialogue:

| Category | Property | Explanation | Allowed values |
|----------|----------------------------------|---|--|
| Logging | | to set the amount of information dumped to the out window while running | |
| | logging | Switch logging on or off for all power tracer loggers | true, false |
| | com.riscure.iso7816.layer1.level | ISO 7816 physical layer | SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, ALL, OFF |

Inspector Hardware Components

| Category | Property | Explanation | Allowed values |
|----------|---------------------------------------|--|--|
| | com.riscure.iso7816.layer2.level | ISO 7816 transport level | SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, ALL, OFF |
| | com.riscure.device.tracer.rs232.level | Serial port | SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, ALL, OFF |
| | com.riscure.device.tracer.level | Power Tracer | SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, ALL, OFF |
| General | | | |
| | tracer.commport | Serial port | COM1, COM2, ... |
| | tracer.stopbits | Number of stop bits in between characters sent (default 2) | <integer> |
| | tracer.enable.reset.feature | enable/disable automatic shutdown after delay | true, false |
| | tracer.power.shutdown.delay | time delay in ms before power down | <integer> |
| | tracer.dcdcshutdownduration | The time in ms that the Power Tracer will run on internal capacitors while the acquisition is ongoing | <integer> |
| | tracer.baudrate | Communication speed between Power Tracer and smart card | <integer> |
| | tracer.fd | One byte value for the FI and DI parameters in the ATR that defines the communication speed | <integer> |
| | tracer.pps | Enable/disable automatic PPS | true, false |
| | tracer.external.clock.enabled | Enable/disable the external clock | true, false |
| | tracer.external.clock.frequency | Configure the frequency of the external clock | <integer> |
| | tracer.gain | Configure the amplification (values 0..100 represent gain 100%..200%) of the power consumption measurement | <integer> |
| | tracer.offset | Configure the offset (values 0..400 represent offset -2V..+2V) for the power consumption signal | <integer> |
| | tracer.voltage | Configure the Vcc voltage (values 0..84 represent voltage 1.8V .. 6V) applied to the smart card | <integer> |
| Protocol | | Protocol parameters | |
| | protocol | force protocol type | T0, T1, raw; |
| | protocol.t1.nad | T=1 NAD | <integer> |
| | protocol.t1.ifsd | T=1 IFS | <integer> |
| | protocol.t1.seqicc | T=1 sequence number for the ICC | <integer> |
| | protocol.t1.seqifd | T=1 sequence number for the IFD | <integer> |

| Category | Property | Explanation | Allowed values |
|----------|----------------------------------|---|----------------|
| | protocol.t0.handle.apdu.response | Handle GET_RESPONSE and reissue requests in T=0 automatically | true, false |
| | protocol.raw.timeout | Set timeout in ms when waiting for a command response in transparent mode | <integer> |

A few examples of setting properties in subclasses of the SideChannelAnalysis class:

- `setDeviceProperty("tracer.pps", "true"); // enable PPS`
- `setDeviceProperty("tracer.fd", "0x01"); // set Fi to 372 and Di to 1`
- `setDeviceProperty("tracer.voltage", "30"); // set voltage to 3.3 Volt`

And some examples of setting properties in the 'Additional Parameters' field of the acquisition dialogue:

- `logging=true; // switch on logging`
- `tracer.external.clock=true; // enable external clock`
- `tracer.external.clock.frequency=3570000; // Configure the frequency of the external clock to 3.57 MHz and make sure the data communication speed matches the clock`

6.3.2 Power Tracer 4

Functional overview

Riscure has developed a dedicated smart card reader called Power Tracer. The Power Tracer has two features different from standard smart card readers:

- It can measure the power consumption of the inserted smart card.
- It can generate a signal to trigger an oscilloscope.

Figure 6.11. The Power Tracer 4



6.3.2.1 Setup and use Power Tracer 4

The Power Tracer is connected with three cables. Two of them (the BNC cables) need to be connected to an oscilloscope, they are the trigger and signal lines. It is easy to mix-up the cables, but the signal cable should be connected to the oscilloscope's signal input, while

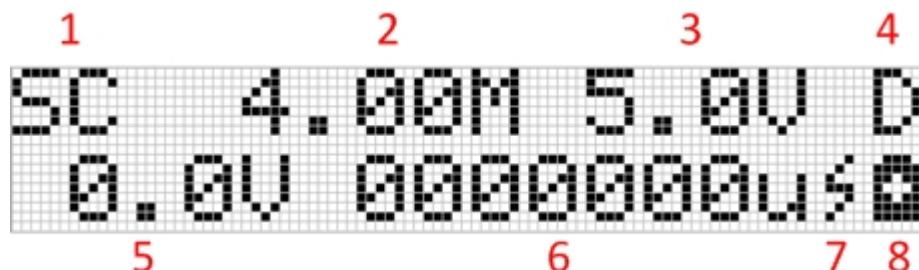
the trigger cable should be connected to the oscilloscope's external trigger input. The third cable (USB) shall be connected to the PC running Inspector. At first connection to any USB connector your operating system will ask for a driver. The driver files can be found in the **drivers** folder. The **drivers** folder is located in the Inspector installation folder.

The Power Tracer has a smart card slot which is suitable for ISO7816 type contact cards (either T=0 or T=1). A card should be inserted with the chip connections facing down.

An LCD display is mounted on the Power Tracer front panel, which presents a selected collection of parameters and settings for the active communication channel:

- Parameters displayed for smart card communication channel

Figure 6.12. Displayed parameters for smart card channel



- Parameter 1 indicates that currently active channel is smart card channel.
- Parameter 2 indicates the clock frequency being synthesized from Power Tracer to the smart card.
- Parameter 3 indicates the voltage supply level from Power Tracer to the smart card.
- Parameter 4 indicates the trigger arming status of Power Tracer, with 'D' stands for 'Disarmed' and 'A' stands for 'Armed'.
- Parameter 5 indicates the configured offset level to the Signal port of Power Tracer.
- Parameter 6 indicates the configured delay (in microseconds) for Power Tracer Trigger signal.
- Parameter 7 indicates that the capacitor bank was completely discharged during the last measurement, and has been switched to charging automatically. This parameter could only be visible if a long measurement in the order of seconds was taken.
- Parameter 8 indicates the charging status of the capacitor bank. The symbol has three variations, as depicted in Figure 6.13, "Variations of the charging status symbol", indicating the capacitor array fully charged, half empty and empty.

Figure 6.13. Variations of the charging status symbol



- Parameters displayed for serial communication channel

Figure 6.14. Displayed parameters for serial channel

The image shows a digital LCD display with the following text:
1 SERIAL 115200bps
D BN1 00000000us
3 4 5

Red numbers 1 through 5 are overlaid on the display to point to specific parameters:

- Parameter 1: SERIAL
- Parameter 2: 115200bps
- Parameter 3: D BN1
- Parameter 4: 00000000us
- Parameter 5: The bottom right corner of the display area.

- Parameter 1 indicates that currently active channel is serial channel.
- Parameter 2 indicates the configured baudrate for the serial port.
- Parameter 3 indicates the trigger arming status of Power Tracer, with 'D' stands for 'Disarmed' and 'A' stands for 'Armed'.
- Parameter 4 indicates the configured framing format for the serial port. It is specified using Data/Parity/Stop convention.
- Parameter 5 indicates the configured delay (in microseconds) for Power Tracer Trigger signal.

6.3.2.2 Specifications Power Tracer 4

The Power Tracer 4 is designed for power and EM analysis on smart cards. The low-noise EM measurement is achieved by:

- Electrical isolation between smart card and digital control.
- EM shielded by closed aluminum housing. By moving the EM probe down towards the smart card the Power Tracer shields the EM probe tip.

This version is included as of Inspector version 4.5.1. This version has improvements. Most notably, the external appearance has changed. The LEDs are replaced by an LCD display and the power tracer needs to be connected to an external power supply.

A summary of Power Tracer 4 specifications:

- USB 2.0 High-speed (480Mbits/sec) and Full-speed (12Mbits/second) compatible.
- BNC connectors
- 15 Volt power supply
- P/S2 port to provide serial interface and +12 Volt power supply
- Data exchange speed: 50 traces/second @ 4MHz CLK.
- Configurable gain between 100% ~ 200% with resolution of 1%

- Configurable offset between -30mA ~ 0mA with 1mA resolution
- 50 Ohm signal output impedance
- Output range:
 - +/- 4 Volt for 1 MOhm input impedance oscilloscope
 - +/- 2 Volt for 50 Ohm input impedance oscilloscope

Figure 6.15. Formula to calculate the voltage level V_{out} on the Power signal port of Power Tracer 4

$$V_{out} = \frac{\text{Gain} \times 130 \times R_L \times (I_{SC} + I_{offset})}{50 + R_L}$$

- The effect of the gain and offset settings on the voltage level is shown in Figure 6.15, “Formula to calculate the voltage level V_{out} on the Power signal port of Power Tracer 4”, where:

Gain = The setpoint to the gain parameter of Power Tracer 4 (1.0 ~ 2.0 for 100% ~ 200%)

R_L = The load (in Ohm) to the Power signal port of Power Tracer 4 (e.g. input resistance of an oscilloscope)

I_{SC} = The current (in Ampere) flows through the GND of smart card

I_{offset} = The setpoint to the Offset current (in Ampere) of Power Tracer 4

- Smart card voltage should be configurable between 1.8 Volt and 6.0 Volt. Voltage on CLK, RST and I/O line should follow
- Smart card CLK is configurable between 1MHz ~ 10MHz with 50kHz resolution

An overview of the main improvements compared to Power Tracer 3:

- The smart card power is drawn from a capacitor bank to reduce measurement noise. Inspector will take the oscilloscope trigger delay into account and send a DC/DC shut-down command after all delays (device delay plus oscilloscope delay). The capacitor bank is fully charged before an acquisition starts. For standard smart cards, the capacitor bank can power the smart card for more than a second without recharging. Recharging the capacitor bank will start automatically when under voltage is detected. This **guarantees** a low noise measurement for more than a second. The state of the capacitor bank is displayed on the LCD panel(see Section 6.3.2.1, “Setup and use Power Tracer 4”).
- Smart card clock signal is **softened** to reduce higher harmonics components of the clock frequency in the smart card power signal.
- Software **configurable** smart card clock frequency (1 - 10MHz). This allows the analyst to tune the clock frequency.

- The Power Tracer 4 can be operated through a USB hub.
- Firmware can be **updated**.

6.3.3 Current Probe

Functional overview

The Current Probe provides a clean and amplified signal of the power consumption to the oscilloscope by insertion into the power line of the TOE. The Current Probe can be applied for various form factors of target devices.

Figure 6.16. The Current Probe and its amplifier



Figure 6.16, “The Current Probe and its amplifier” shows the amplifier on the left hand side and the Current Probe on the right. A plug is connected to the Current Probe that allows connection between Current Probe, Glitch Amplifier and TOE.

Setup Current Probe

The Current Probe should be connected in the power line of an embedded TOE. In order to achieve that, the power supply chain of the target must be opened. The location to open the supply chain should be chosen such that the capacitance C2 (see Figure 6.17, “Circuit diagram of current probe and connections”) between the cut and the TOE is minimal – just sufficient for proper operation of the TOE. One of the Current Probe inputs must be connected to the Vcc pin of the TOE and the other input to the power supply of the TOE or the stabilizing capacitor C1. The probe output is connected to the IN connector of the amplifier which is powered by a 12V DC power adapter. The OUT connector of the amplifier should be connected to the input of the oscilloscope (to 500Ohm). A trigger signal for the oscilloscope should be provided externally.

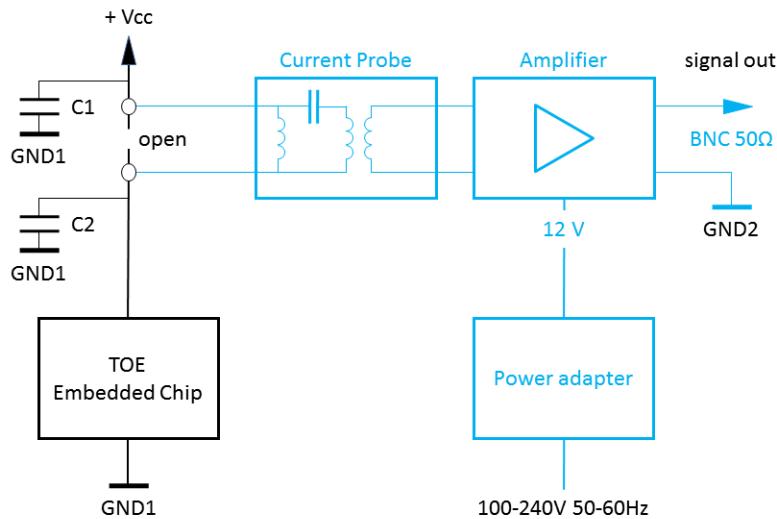
Figure 6.17. Circuit diagram of current probe and connections

Figure 6.17, “Circuit diagram of current probe and connections” shows a schematic overview of the insertion of the Current Probe into the power line of the TOE and the connection of the Current Probe to the amplifier, oscilloscope and power adapter.

Specifications

- Probe bandwidth 1 MHz – 1 GHz
- Probe impedance $60 \text{ m}\Omega + 10 \mu\text{H}$ for $f < 200 \text{ kHz}$
- Probe sensitivity is 25 mV/A without amplifier (into 50Ω impedance)
- Signal output with 50Ω impedance
- Amplifier features:
 - Frequency Range: 0.1 - 2500MHz
 - Gain @500MHz: 25dB
 - Noise Figure @500MHz: 2.4dB
 - DC Power: 12V
 - SMA Connector
- Built-in Tektronix CT1 current probe

6.3.4 EM Probe Station

The EM Probe Station is developed for measuring the Electro-Magnetic field that is emanated by a device in order to do a side channel analysis on this signal. The EM Probe Station consists of several parts of which the *EM Probe* is the most important. This probe captures the actual EM emanation and does some preprocessing for electrical signal to the oscilloscope. The *XYZ-table* on which the EM Probe is mounted, enables doing a XY-scan over a surface area. The *EM Shield or Field Deflector* is a hardware device to reduce unwanted EM signals.

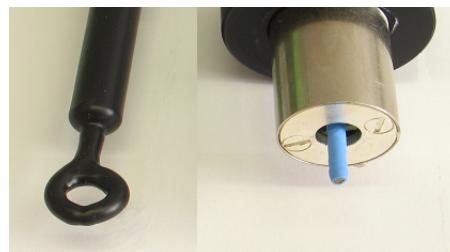
The EM Probe Station can also be used for EM Fault Injection scenarios. See Section 6.4.4, “EM-FI Transient Probe” for more information on the EM-FI probe.

6.3.4.1 EM Probe

Specifications

Calibrated ElectroMetrics EM-6992 sensor and EM-6990A-1 amplifier were used to verify sensitivity and bandwidth. Figure 6.18, “the ElectroMetrics probe and the Inspector probe” shows the sensor tips of the ElectroMetrics probe and the Inspector probe.

Figure 6.18. the ElectroMetrics probe and the Inspector probe



The specs of the Inspector probe are measured by comparison with the Electrometric probe (sensor and amplifier), both placed in the same EM field. Table 6.1, “Probe specs in comparison with calibrated Electrometric probe” gives the specs of both probes.

Table 6.1. Probe specs in comparison with calibrated Electrometric probe

| Probe | Electro-Metric | Inspector HS | Inspector LS |
|--------------------|--------------------|-------------------|-------------------|
| Sensitivity @ 1MHz | 1 mV/140nT | 100mV/1microT | 20mV/1microT |
| Bandwidth | 1.2 GHz | 0.3 GHz | 1 GHz |
| Resolution | 80 mm ² | 1 mm ² | 1 mm ² |

Figure 6.19. Time signal of ElectroMetrics probe (top) and Inspector probe LS (bottom) with equal magnetic field.

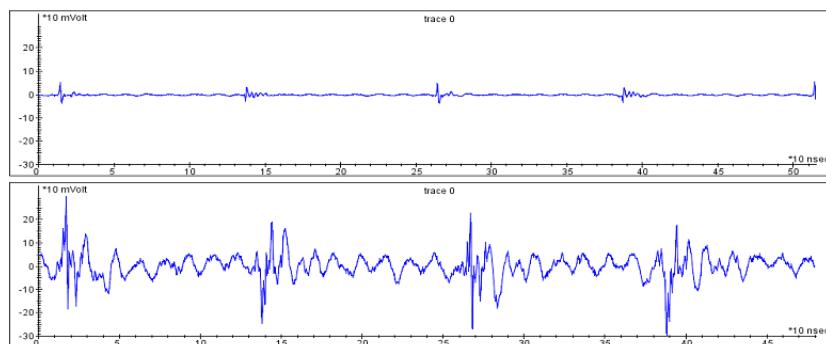
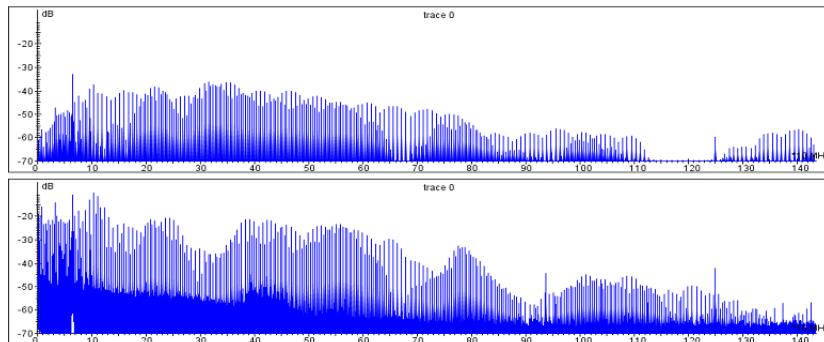


Figure 6.20. Frequency spectra of ElectroMetrics probe (top) and Inspector probe LS (bottom) with equal magnetic field.



The EM Probe is separately powered via the power jack connected to it. It requires a DC voltage supply between 5.0 V and 7 V. This can be done by connecting the EM Probe with the supplied cable to the Power Tracer. However when no Power Tracer is used in the acquisition an external supply can be used like the adapter which was supplied or a lab supply or a battery. A battery would be ideal in the fact that it does not have any additional power supply noise. When connecting an external power supply take care not to reverse the polarity. Also make sure before starting a measurement, that the battery has enough capacity to last during the complete measurement.

6.3.4.2 XYZ table

The XYZ motors move the EM probe up and down and move the probe tip horizontally over a XY plane to scan the EM emanation of the chip. The motors have a movement range of approximately +/- 25 mm, a step size of 0.0025 mm and a repeated accuracy of 0.05 mm. They are controlled by a motor controller, mounted on the back side of the EM Probe Station. The motor controller is powered by a 24 V power supply which must be connected to the mains. The motor movement is handled by the XYAcquisition module of Inspector, which communicates with the motor controller through a serial cable or USB cable.

Note

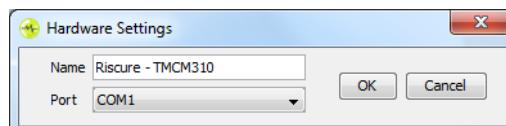


The EM Probe Station model supported by Inspector is the TMCM310. This model only has a RS-232 (serial) interface. This interface does not work properly in combination with a serial-to-usb connection because of a voltage mismatch.

Inspector configuration

Inspector is made aware of a (connected) EM Probe Station by adding it to the Hardware Manager. Figure 6.21, “Adding an EM Probe Station to the Hardware Manager” shows the Hardware Manager dialog that appears when adding an EM Probe Station. The name can be chosen freely. Under port, the port on which the EM Probe Station was connected should be chosen. The port will have a name similar to "COMx" or "Serial @COMx" where x is a number.

Figure 6.21. Adding an EM Probe Station to the Hardware Manager



6.3.4.3 Field Deflector

Functional overview

The Field Deflector is a hardware device that is suited for filtering strong EM signals. This is especially useful during EM measurements on RF cards. Positioning the sensitive EM probe in the strong RF field drives the output amplifiers of the EM probes to non-linear saturation. To prevent this the RF carrier must be attenuated before the EM probe picks up the signal. The Field Deflector is an EM shield that reduces the RF field and does not attenuate the EM emanation from the smart card chip.

The design of the Field Deflector makes use of the difference between the geometry of the RF field and the EM emanation from the smart card chip. The RF field has a plane wave front with parallel flux line in the centre of the RF reader antenna while the chip field has a spherical wave front with diverting flux lines. In order to optimize the shield design an EM field simulation was performed with the ANSYS finite element package. We used an axi-symmetrical model for simplicity of building the model see Figure 6.22, “3D view of RF reader antenna (purple), chip circuit (orange), EM probe and shielding device (pink)” and Figure 6.23, “2D view of RF reader antenna (purple), chip circuit (orange), EM probe and shielding device (pink)”. The reader antenna was modeled by a circular antenna (purple) instead of a rectangular antenna. In the model the EM chip emanations are generated by a circular loop with approximately 1.5 mm diameter. Although the model is a simplification we expect the optimized shield design also to work for rectangular reader antenna en other circuit shapes on the chip.

Figure 6.22. 3D view of RF reader antenna (purple), chip circuit (orange), EM probe and shielding device (pink)

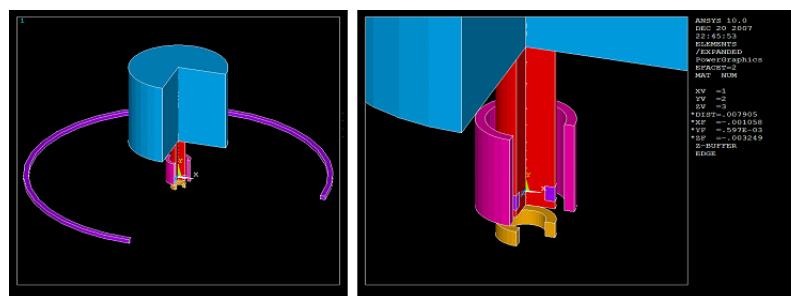


Figure 6.23. 2D view of RF reader antenna (purple), chip circuit (orange), EM probe and shielding device (pink)

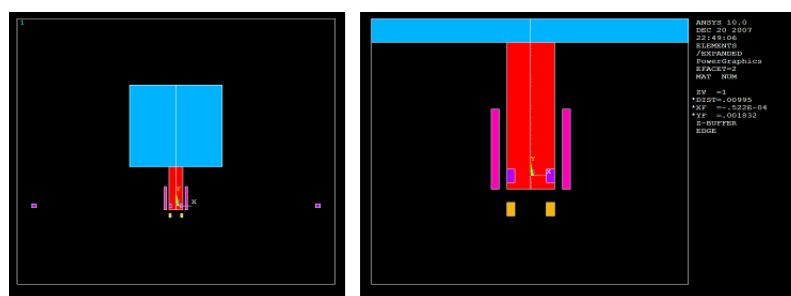


Figure 6.24, “Shielding deflects field by RF reader antenna; unshielded (left) and with probe shield (right)” shows the flux lines of the reader RF field without (left) en with (right) the shield.

The shield is an aluminum or copper tube which has the properties of a short-circuited winding. Any flux lines entering this tube cause a current to run through the winding. The shield current generates an opposite field. The shield field cancels the reader field and the RF field of the probe sensor is attenuated. Figure 6.25, “Magnetic flux of the chip field; unshielded (left) and with probe shield (right)” shows the flux lines of the chip field without (left) en with (right) the shield. The current in the shield pushes the flux lines out again but the simulations also show that the flux lines penetrate the shield and reach the probe sensor. The shield does reduce the field by the chip but to a much lesser extent than the RF field.

The simulations are verified and the shield's operation is confirmed by measurements.

Figure 6.24. Shielding deflects field by RF reader antenna; unshielded (left) and with probe shield (right)

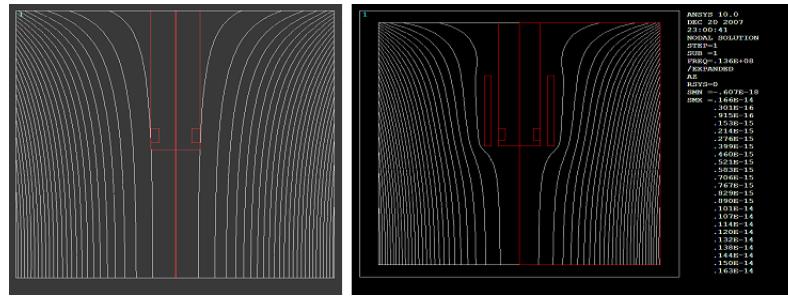
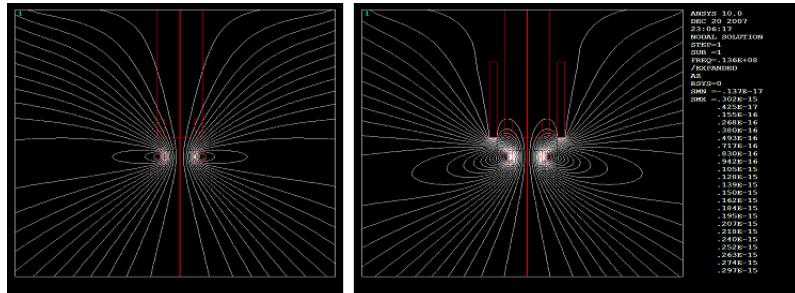


Figure 6.25. Magnetic flux of the chip field; unshielded (left) and with probe shield (right)



Setup and use

As mentioned, the Field Deflector is a small metal tube. It can be easily slotted over the tip of the EM Probe when it required, or removed from the probe when it is not.

6.3.5 MicroPress MP300

Functional overview

The Micropress MP300 is a highly configurable tester for smart cards. The MP300 driver for Inspector allows you to use the MP300 tester with a TCL1 or TCL2 resource in Side Channel Acquisition and Fault Injection setups. It allows control over most features of the MP300.

Figure 6.26. The MicroPross MP300 with TCL1 resource

Resource modules for contactless communications other than the TCL1 or TCL2 may function with this driver, but have not been tested by Riscure.

6.3.5.1 Setup and use

Note



Basic knowledge on operating procedures of the MP300 tester is required.

To use the communications DLLs from Micropross, install the Micropross drivers on the Inspector workstation following the Micropross installation procedure.

The MP300 tester supports several connection methods with a workstation. The default selected method for the Inspector driver is via a TCP/IP network. You will need to use the Micropross MPManager tool that comes with the MP300 to configure a valid IP address.

Inspector uses the default antenna configuration of the MP300. The external antenna (Micropross PN 907-2094B) can be attached with a BNC T-Splitter to the TX/RX port of the MP300 and the oscilloscope input channel.

The trigger.out and trigger.out2 ports may be used to connect to the oscilloscope trigger input. The default configuration for the triggers is shown Table 6.2, “Trigger configuration”.

Table 6.2. Trigger configuration

| # | MP300 configuration | Description | property parameter |
|---|---------------------|--|-----------------------------------|
| 1 | TX_AFTER_DELAY | After the transmission of a frame, the trigger out will be set to a logic 0 then set to a logic 1 after the delay (starting at the last rising edge of the modulation coming from the PCD) | <i>device.mp300.trigger1.mode</i> |
| 2 | TX_OUT | The trigger out is the representation of the signal that is used to modulate the carrier during a transmission. | <i>device.mp300.trigger2.mode</i> |
| 3 | UNSET | This trigger is left unconfigured by Inspector. It is only available on TCL2 resources. | <i>device.mp300.trigger3.mode</i> |

The next action is to configure Inspector to use the MP300 with the selected connection settings.



Note

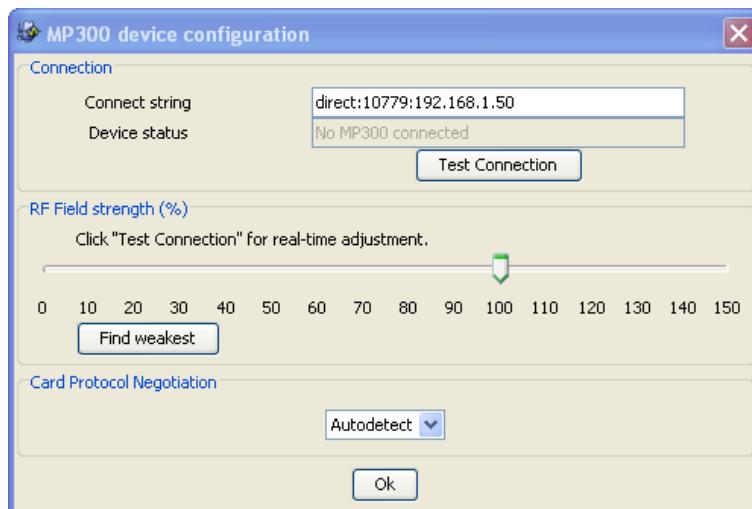
The external antenna is not tolerant to metal around it, which induces the currents to work against the RF fields. The solution is to lift the antenna from the metal plane of the EM station by using a wooden or plastic brick. The thickness of this brick should be 46mm at least.

Inspector configuration

You can configure and test the connection by opening the configuration dialog for this device in Inspector. To open the configuration dialog:

1. select an acquisition module and go to the device tab
2. select the device class com.riscure.device.mp300.MP300Device
3. Click the Configure button, the device configuration dialog as shown in Figure 6.27, “Configuration Dialog for MP300” will appear.

Figure 6.27. Configuration Dialog for MP300



- Set the *connectstring* according to the syntax as described by Micropross.
- The *Device status* field provides identification of the MP300 and the resource used by Inspector. This field is updated when the driver connects with the MP300.
- Click on the *Test Connection* button. When the driver is connected to the MP300, the button text is changed to *Disconnect*.
- The *RF Field Strength* slider allows output power adjustment of the RF carrier field in a percentage of the maximum output. When the MP300 driver is connected, the RF field is updated in real time to reflect the new setting.
- The *Find weakest* button starts detection of the weakest field strength level at which the target card still returns a power-up response (ATS or ATQ)

The target card may switch off countermeasures to reduce its power consumption when it is placed in a low-energy environment. Additionally, lowering the field strength of the RF field increases the SNR. In some circumstances the weakest field strength found by the driver is too weak for the target to support APDU processing. Increasing the field strength by a small percentage (2- 10 %) will solve this problem.

- The *Card protocol* box controls which protocol is used by the MP300 to communicate with the target card.

When *autodetect* is selected, the MP300 is configured to try selection on ISO-14443 type A and type B cards.



Note

The configuration dialog only displays the frequently used settings. Additional settings may be configured via the device specific properties.

6.3.5.2 Specifications

These specifications apply to what is supported by the Inspector driver for the MP300. Other restrictions may apply depending on the configuration of the MP300.

- **Supported resource modules**

TCL1, TCL2.

Other resource modules may function with this driver, but are not tested by Riscure.

- **Connections**

- *Micropross MP300 communications driver DLL*

USB,serial, networking (any connection string supported by your driver DLL).

- *native TCP/IP driver*

The native TCP/IP driver is fully implemented in Java and does not require external components to connect to a MP300 over TCP/IP. It is the default driver for Inspector and provides an alternative on workstations that do not have the Micropross MP300 communications driver DLLs available.

This driver is selected when the connect string starts with the *direct* keyword. The syntax for the native driver is identical to that of the Micropross *tcp* connection method:*direct:[port]:[hostname or IP address]*

resources in a *LOCKED* state are always overridden by the driver (see Micropross MP300 documentation for details.)

- **Card protocol support**

ISO-14443A, ISO-14443B

- **Triggering**

Output triggers 1, 2 and 3 (3 is only available on TCL2 module).

- **RF field properties**

RF output field strength adjustment, carrier frequency, modulation index, modulation rise and fall times

Device driver parameters

The MP300 driver supports advanced configuration by supplying device specific parameters in the *Device* tab of Inspector. Some of the frequently used parameters can be found in the device configuration dialog. All of the parameters are described in detail in the documentation of the MP300 in similar names. A table of device parameters is included in Appendix D, *MP300 Device driver parameters*.

6.3.6 RF Tracer

Functional overview

The RF Tracer is a device to enable side channel measurements of contactless smart cards. It is designed to be used together with an EM probe station to capture the side channel information. The device has the following functionality:

- External trigger for oscilloscope with programmable delay relative to communication between reader and smart card
- RF monitor signal to monitor the variation in the RF field
- Digital Tx signal to monitor the transmission from reader to smart card
- communication with contactless smart card according to standard (ISO14443-4 A/B)
- inter operable with Inspector software

Figure 6.28. The RF Tracer



6.3.6.1 Setup and use

Four LEDs indicate the status of the RF Tracer.

- **Green LED**

- Device power off: Off
- Device power on: On
- **Blue LED**
 - No acquisition in progress: Off
 - Acquisition in progress: On
- **Yellow LED**
 - Slow communication between reader and smart card or idle mode: OFF
 - Fast communication between reader and smart card: Blinking
- **Red LED**
 - Fast communication between reader and smart card: Off
 - Slow communication between reader and smart card: Blinking fast
 - Idle mode: Blinking slow

The connectors to the RF Tracer are located at the back side of the box (see figure below). From left to right:

- USB connector for communication with acquisition software and power supply to the RF Tracer
- BNC connector for trigger output signal towards oscilloscope (TTL-level)
- BNC connector for RFA monitor signal. This signal can be used to monitor variation in the RF field. We recommend not to use this signal for side channel analysis. The best way to perform side channel analysis is to measure the EM emanations of the smart card chip.
- BNC connector for digital Tx transmission signal from reader to smart card (TTL-level)

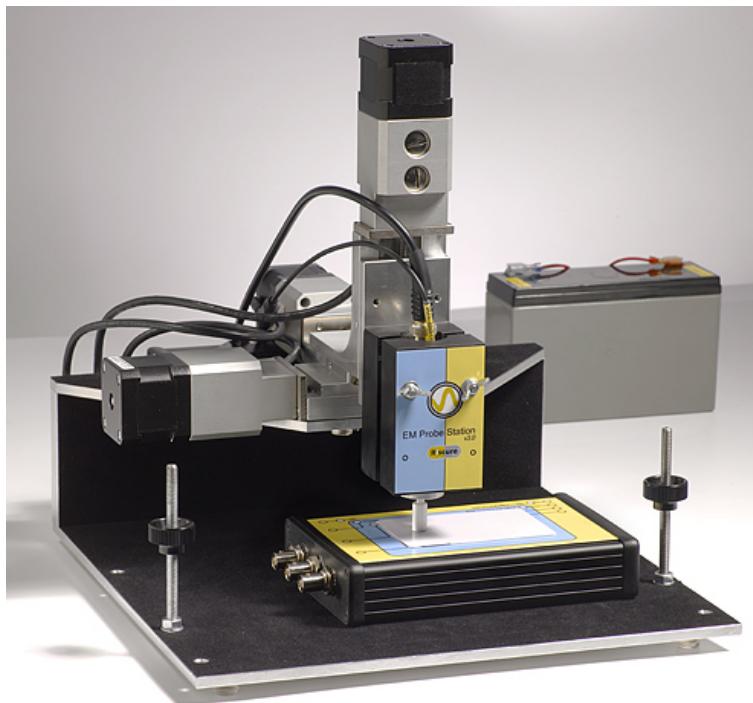
Figure 6.29. connectors on the RF Tracer



The RF Tracer is intended to be used in combination with the EM ProbeStation (see figure below) for side channel analysis. In fact it is recommended by Riscure to use the EM probe to do measurements on contactless cards.

The top side of the RF Tracer shows the area of the RF antenna in blue. The smart card should be positioned in this area. We recommend placing the smart card chip close to the centre of the reader antenna. The precise position of the EM probe above the smart card chip can be found by scanning the chip area with the XYZ table of the EM ProbeStation and the Inspector scanning module. The Section 6.3.4.3, “Field Deflector” deflects the RF field without attenuating the EM emanation from the smart card chip. The optimum tube position with maximum attenuation of the RF field must be found experimentally.

Refer to section Section 6.3.4, “EM Probe Station” for more information on performing an EMA on a contactless card.



The RF Tracer is designed for RFA and EMA-RF analysis on contactless smart cards. The low-noise RFA and EMA-RF measurement are achieved by:

- stabilized antenna driver supply voltage to suppress sub-harmonics of the RF carrier wave
- antenna filter circuit to reduce harmonic distortion of the RF carrier wave (THD < -30 dB)
- built-in separate pick-up antenna for RFA signal with built-in amplifier and high-pass amplifier for reduction of the RF carrier frequency (13.56 MHz)

6.4 Fault Injection

This section describes the components used for Fault Injection setups.

6.4.1 VC Glitcher

Functional overview

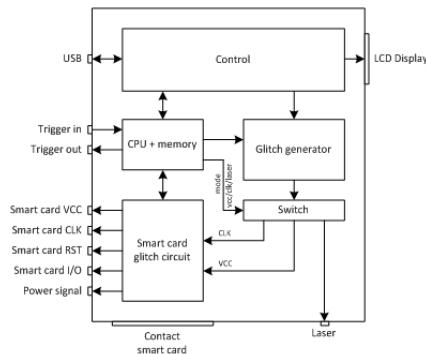
The VC Glitcher is developed to test smart card products for resistance against voltage and clock glitching. The user controls the VC Glitcher with the Inspector FI software. After the parameters are set, glitches are executed and automatically varied in a test run which can be left running unattended. The results from the card are stored for fault analysis. By zooming in on the behaviour of the chip it is possible to identify a failure in the voltage and clock sensors of the chip. For the VC Glitcher dedicated software modules are available in Inspector FI to test different scenarios, such as Differential Fault Analysis (DFA) on DES and RSA.

Figure 6.30. The VC Glitcher



The VC Glitcher uses FPGA technology for producing high resolution glitches. The following block diagram shows the architecture of the VC Glitcher.

Figure 6.31. VC Glitcher block diagram



6.4.1.1 Setup and use

From version 1.1, the VC Glitcher has 11 SMB connectors, as shown in Figure 6.32, “New VC glitcher connectors”.

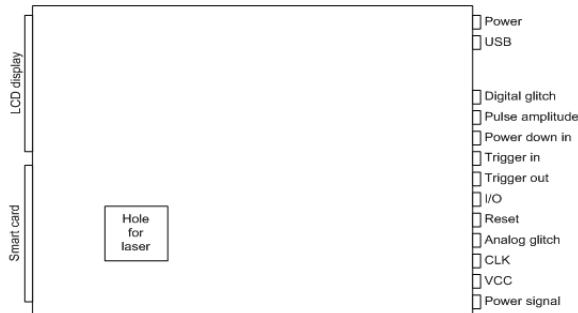
Figure 6.32. New VC glitcher connectors

Table 6.3, “VC glitcher connectors” describes these connectors in more detail.

Table 6.3. VC glitcher connectors

| Connector | Description |
|-----------------|--|
| Power | +15V (center positive) power supply connector. |
| USB | USB connector. |
| Digital glitch | Connector that provides a digital glitch signal. This can be used to control a laser diode. |
| Pulse amplitude | SMB connector that provides the pulse amplitude voltage. This can be used to control a laser diode. |
| Power down in | Input signal that can be used to power down the smart card. Note that this connector is only available on the VC Glitcher version 1.1 and above. See Section H.1, “VC Glitcher API” “Power Down Methods” for more information. |
| Trigger in | Trigger input connector. |
| Trigger out | Trigger output connector. |
| I/O | SMB connector for monitoring the smart card I/O signal. |
| Reset | SMB connector for monitoring the smart card RST signal. |
| Analog glitch | SMB connector that provides an analog glitch signal. This can be used to control a laser diode. |
| CLK | SMB connector for monitoring the smart card CLK signal. |
| VCC | SMB connector for monitoring the smart card VCC signal. |
| Power monitor | SMB connector for monitoring the smart card power consumption. |

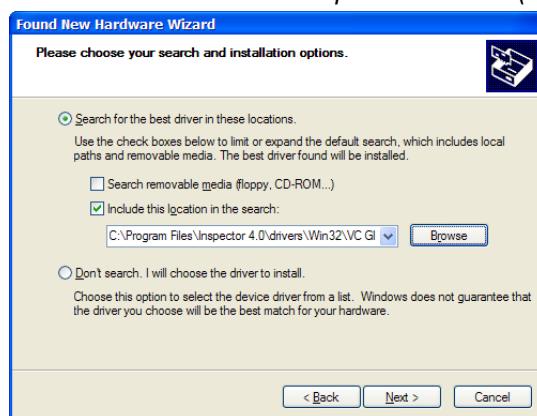
The VC Glitcher has a blue, 2 × 16 characters LCD display. The display shows the current mode of the device (CLK, VCC or OPT (optical)). In addition it shows the current state (IDLE, PARSING ATR or GLITCHING).

6.4.1.2 Driver installation

- First of all, make sure Inspector FI is installed correctly.
- Connect the VC Glitcher to the PC using the supplied USB cable and power it using the supplied power supply.
- Windows will detect the device and shows the dialog shown below. If Windows asks to search for a driver online, select *no*.



- Select *Install from a list or specific location (Advanced)* and click *Next*.



- Select *Search for the best driver in these locations*, deselect *Search removable media* and select *Include this location in the search*. Enter C:\Program Files\Inspector X.X\drivers\Win32\VC Glitcher in the text box and click *Next*.



- If Windows warns about the fact that the software did not pass Windows Logo testing, click *Continue Anyway*.
- Windows will find another VC Glitcher device. Repeat the same steps again for this device.

Once the USB drivers for the VC Glitcher are installed, Inspector will automatically install the firmware included with the Inspector Workstation version if it is different from the version installed on the VCGlitcher.

6.4.1.3 Specifications

The limits to which the CLK and VCC lines can be driven are shown in Table 6.4, "Voltage specifications".

Table 6.4. Voltage specifications

| | MIN (V) | MAX (V) |
|--------------------------------------|---------|---------|
| VCC, CLK (excluding glitch) | 0.0 | + 6.6 |
| Absolute VCC, CLK (including glitch) | -0.8 | + 7.5 |
| Relative glitch voltage | -7.4 | + 4.2 |

Technical specifications

- 2 Kb data memory (512×32 bits) for storing communication with smart card and glitch parameters
- 4 Kb instruction memory ($2k \times 16$ bits) to store the low-level glitch program
- Runs at 50 MHz
- Digital pattern generator (i.e. glitch generator), patterns of 500 samples @ 500M samples/sec

Revision history

Version 1.1

Added *Power Down input* to the device

6.4.1.4 Data base management

The glitch results are stored in a PostgreSQL data base. When Inspector FI is installed the database is also installed. It should work out of the box and for typical Inspector use the database layer should be transparent. Sometimes, however, access to the database is useful, for which we will describe a number of use cases.

Database backup

From within Inspector it is not possible to create backups of the database. You can use the PostgreSQL tool "pgAdmin III" to create a backup:

- Click Start -> Programs -> PostgreSQL 8.3 -> pgAdmin III
- Connect to the local server using the postgres user.
- The password for this user can be found in %AppData%\postgresql\pgpass.conf²
- Under Databases, right-click inspector and click "Backup...". Select a destination file and use the default options.

²The exact file path might differ, depending on your installation.

Database restore

It should be possible to restore the database using the same tool as described above.

- In the database section, look for a database named "inspector". If present, right click on it and select delete/drop. This command will delete all the contents of the database, therefore you might want to back it up first, as explained above.
- Once the database has been deleted, right click on "Databases" and select "New Database...". Set the name to "inspector" and the owner to "inspector" (case is important), then click "OK".
- Once the database has been created, right click on it and select "Restore...". Select the previously backed up database and click "OK".

The following warning message can be reported:

```
===  
pg_restore: [archiver (db)] Error while PROCESSING TOC:  
pg_restore: [archiver (db)] Error from TOC entry 8481; 2612 16386  
PROCEDURAL LANGUAGE plpgsql postgres  
pg_restore: [archiver (db)] could not execute query: ERROR: language  
"plpgsql" already exists  
Command was:  
CREATE PROCEDURAL LANGUAGE plpgsql;  
==
```

Even if the message is printed, the database will be restored correctly. Please note that if the Windows username of the old and the new PC is the same, no further operations are needed. However, if the username is different, another step is required. Select the "Inspector" database and run Tools -> Query Tool. Then, enter the following text: UPDATE logs SET username = 'a' WHERE username = 'b'. a and b need to be replaced respectively with the username of the new and the old PC (case is important). Finally, press F5 to run the query.

Export verdict color

The report window that shows the results of the perturbation attempts (see Figure E.10, "The VC Glitcher report window") uses colors to indicate the verdict of the perturbation run. For instance, green means "Normal" and red means "Successful". The color information is stored as a number in the database and is used by Inspector to color a row. It is possible to retrieve this data from the PostgreSQL database. Here is an example for importing this information into Excel 2007:

- In Excel, click Data -> From Other Sources -> From MS Query
- Select vcglitcher and click "OK"
- Add the logs table to your query and click next a few times
- Click "Finish"
- Then you'll have to select the cell where the data is loaded
- In this table you'll find the tablenames

- You can repeat the previous steps for the table names that you need
- You can then use Excel to filter on the color column

Please note this this procedure is not officially supported by Riscure. It will work slightly differently for other Excel versions.

6.4.2 Diode Laser Station

Functional overview

The Diode Laser Station (DLS) is the Inspector component designed to perform optical perturbation attacks on chips. By activating one of the available diode lasers at a precise moment and location, temporary faults can be induced in the executing program.

Figure 6.33. The Diode Laser Station



The DLS features a microscope setup optimized for optical perturbation, diode lasers for delivering the laser pulse, a camera for aiming at the target, an XY stage for precise target localization, and optionally a security cabinet for shielding the operator from the laser beam. The DLS is tightly coupled with the VC Glitcher for target control and triggering, and is controlled through the VC Glitcher software interface. For more advanced triggering and preventing permanent card damage, the icWaves device is typically used.

A typical usage scenario is to first determine whether a front or back side attack will be performed on a chip, and installing the appropriate laser for this task. Next, timing parameters can be deduced from power measurements of the card. After fixing these parameters, an automated search of the chip surface can be executed. All glitching parameters are controlled from the VC Glitcher, and the Inspector software is used to store the parameters and results of each injection.

6.4.2.1 Setup and use

Microscope

The most important components of the microscope are the focus unit, the beam splitters, the objectives and the camera tube.

The focus unit is controlled by two wheels on both sides of the focus block. During the first revolution of these wheels, the focus unit is in fine-tuning mode: the distance between objective and die changes with 350 µm per revolution. After the first revolution, the focus unit switches to coarse-tuning mode: 2400 µm per revolution. The operator can recognise the coarse-tuning mode because he needs more force to turn the wheels.

The beam splitters are within the box-shaped part of the microscope. They direct the laser beam and the illumination through the objective down to the chip and the reflected light from the chip up towards the camera tube. The operator does not need to switch between laser or camera with mirrors as is sometimes required in laser cutting set-ups.

The camera tube is the tube between the microscope and the camera. The camera tube projects the chip image on to the camera sensor.

The 5x, 20x and 50x objectives are designed for use with visible and near infra red (e.g. 1064 nm) light. Hence, the objectives are highly transparent and the focus distance is constant for this wavelength range. The objectives have a high working distance, leaving sufficient space between the chip surface and the bottom of the objectives. By turning the revolver, another objective is selected:

| Objective | Die image | Laser spot size |
|-----------|--------------|-----------------|
| 5x | 1.2 x 1 mm | 60 x 14 µm |
| 20x | 300 x 250 µm | 15 x 3.5 µm |
| 50x | 120 x 100 µm | 6 x 1.4 µm |

Illumination

The illumination of the target is provided by an external light source. This device is connected to the microscope using a fiber-optic cable. There are two versions: KL1500 LCD with halogen cold light and the KL1500 LED with LED lighting. On the first there are two controls of interest: the light color (setting 1- 6 corresponding to 2650 – 3300 K) and the light intensity (setting A – E corresponding to a double intensity by each step up). We recommend leaving the light color at setting 1-3, and using the intensity to control the desired illumination level. The light source in the device has a severely restricted lifetime when used at a color setting of 4 and higher. Please refer to the manual of the light source for details. On the KL1500 LED with LED's there is one control (0-5) for the light intensity.

Camera

The camera connects to Inspector through a USB connection. It is used to inspect the target area for fault injection. In the Classical Perturbation modules the camera view is automatically activated when a *SmartCardOpticalPerturbation* module is started. The same viewing window can be opened anytime from the Tools menu option. On the other hand, in the Dynamic Perturbation modules (Run->Perturbation2) the viewing window can be opened from the Camera tab in the module. Also more camera specific options are available there. Inspector is shipped with a uEye camera for which the drivers are installed during the installation process of Inspector.

As a trouble-shooting tip it should be noted that on some older hardware the uEye camera might not work properly, resulting in an empty viewing window. This can be fixed by changing the pixel clock of the camera in the registry. The exact path in the registry might differ, but should look something like this:

HKEY_CURRENT_USER\Software\IDS\uEye\DirectShow\Device1\Settings\UI224xSE-C\Timing

Here, the "Clock" variable (expressed in MHz) should be lowered from 0x14 (20 decimal) to 0x05 (5 decimal). If more problems occur, the user might try updating the uEye driver to the latest version. See the IDS home page (<http://www.ids-imaging.de>) for driver downloads.

XY stage

The XY stage is responsible for moving the target. It can be manually controlled using the provided joystick, or automatically using Inspector.

The stage is connected with two cables to the controller. This controller is connected to the joystick and the PC (using USB).

The joystick control is switch on only when an *OpticalPerturbation* modules is opened. This is to prevent accidentally moving the chip position.

Diode lasers

The DLS comes standard equipped with two lasers: a 808 nm Red/NIR and a 1064 nm NIR module. They are both controlled by the VC Glitcher device. The operator has to replace one laser for the other. The laser is mounted on top of the spot size reducer.

808 / 1064 nm lasers

The deep red / near infra red (808 / 1064 nm) lasers has four connections:

- 12 V: for the +12V power supply. Connect to proper power supply. DO NOT USE THE 15 V SUPPLY AS USED FOR THE VC GLITCHER / ICWAVES.
- Diode current monitor': output to monitor the current though the laser diode. The scaling is 20 A / V when connected to a 50 Ohm oscilloscope input.
- Pulse amplitude: analog input to set the laser power. 0 V corresponds to 0% and 3.3 V corresponds to 100% laser power. Connect to the pulse amplitude' output of the VC Glitcher.

Digital glitch: digital input to modulate the laser. 0 V corresponds to laser off and 3.3 to 5V corresponds to laser on. Connect to the digital glitch' output of the VC Glitcher Maximum modulation frequency is 25 MHz

- This laser can be operated in CW from 20 ns up to 100 microseconds. The laser has a thermal protection that switches off the laser until it has sufficiently cooled down below 30 C. The ambient temperature should be below 30 C.

Spot size reducer

The spot size reducer is a tube with lenses to reduce the laser spot size. The spot size reducer needs to be adjusted to focus the laser spot. The adjustment is done by turning the top part relative to the bottom part, see arrow 5 in the following figure. Turning the top part clock wise decreases the distance between the top and the bottom part. For the 808 nm laser, the top part

should be turned approximately 2 revolutions counter clock wise. The 1064 nm laser requires the maximum distance. Tightening the screw as indicated by arrow 2 locks the adjustment.

The laser can be detached from the spot size reducer by loosening the 3 screws at the top of the spot size reducer (see arrow 1).

The spot size reducer has an aperture (see arrow 3) for a filter. Tightening screws 4 fixes the filter.



Spot size reducer

Filters

The accompanying filters of 10%, 1% and 0.1% can be inserted in the spot size reducer to reduce the laser power by 10%, 1% and 0.1% respectively. This allows fine tuning of the laser power in the low power range.

Software interface

For the DLS, make sure Inspector FI is installed. In the Inspector system folder there is a drivers\Win32 folder containing the drivers for the XY stage and the camera. Please install these drivers before connecting the devices.

For more information about the integration between the DLS and Inspector, please refer to the SmartCardPerturbation and SmartCardOpticalPerturbation help files.

6.4.2.2 Specifications

Lasers

| | 808nm red laser | 1064nm NIR laser |
|---------|------------------------------------|-----------------------------------|
| Purpose | Smart card chip front-side testing | Smart card chip back-side testing |

| | 808nm red laser | 1064nm NIR laser |
|---|--|--|
| Wavelength | 808 nm | 1064 nm |
| Type | Multimode | Multimode |
| Laser pulse power | 0-14W (adjustable) | 0-20 W (adjustable) |
| Max pulse frequency | 25 MHz | 25 MHz |
| Propagation delay | 50 ns | 50 ns |
| Effective spot size (50x objective) [1] | 6x1.4 μ m | 6x1.4 μ m |
| Optics for spot size reduction and energy retention | Yes | Yes |
| Laser class | CLASS IV laser product | CLASS IV laser product |
| Diode life time | No degradation of laser power @ 3500 hrs continuous operation. In pulsed mode much more. | No degradation of laser power @ 3500 hrs continuous operation. In pulsed mode much more. |
| Laser controls | Analog input 0-3.3V for power level, TTL input for laser modulation, laser diode current monitor output 20 A/V | Analog input 0-3.3V for power level, TTL input for laser modulation, , laser diode current monitor output 20 A/V |

[1] Spot size is measured as the chip surface area in which 80% of the laser power is concentrated.

Operating requirements

| | |
|-------------------|--------------------------------|
| Temperature | 20 degrees Celsius + 5 degrees |
| Relative humidity | 20% - 80% non condensing |
| Voltage | 100 – 240 V, 50 – 60 Hz |
| Power | 700 W max |

Camera

| | |
|--------------------------------|----------------------|
| Frames per second | 15 |
| Sensor | 1/2" Sony CCD colour |
| Resolution | 1280 x 1024, SXGA |
| Die image size (5x objective) | 1.2 x 1 mm |
| Die image size (20x objective) | 300 x 250 μ m |
| Die image size (50x objective) | 120 x 100 μ m |

Microscope XY stage

| | |
|----------------------|------------------|
| Maximum travel range | 75 x 50 mm |
| Repeatability | < 1 μ m |
| Accuracy | 3 μ m |
| Step size | 0.05 μ m |
| Travel speed | 45 mm/sec |
| Controller interface | USB and joystick |

Microscope

| | |
|-----------------------------------|--|
| Combined coarse - fine focus unit | 2400 μ m / rev – 350 μ m / rev |
|-----------------------------------|--|

| | |
|---|--|
| Cold light source KL 1500 through fiber optic | |
| 5x objective | M Plan NIR NA=0.14 mm, F=40 mm, WD=37.5 mm |
| 20x objective | M Plan NIR NA=0.40 mm, F=10 mm, WD=20 mm |
| 50x objective | M Plan NIR NA=0.42 mm, F=4 mm, WD=17 mm |

Safety box (optional)

| | |
|---------------------|--|
| Purpose | Enable safe operation of the DLS in an open room |
| External Dimensions | 76.5 x 53.5 x 44.5 cm (hxwxd) |
| Indicators | Externally visible light indicator during laser power up |
| Shut down | Laser power is shut down when door is opened |
| Safety regulations | No warranty is provided that the safety box meets local safety regulations in the country in which the DLS is operated |

6.4.2.3 Wear

Endurance tests by the laser diode manufacture show no drop of laser power after 3500 hours of continuous operation at rated power level. For temporary fault injection, the laser is used less than 1 percent of the time. The laser diode manufacturer expects that the laser diode can be used much longer than 3500 hours for pulsed operation without power decrease.

6.4.3 Splitter

Functional overview

The splitter is used in a multi-area optical fault injection set-up. The multi-area set-up consists of several lasers which need to be controlled. The control device VC Glitcher has a single pair of output connectors (digital glitch and pulse amplitude) to control a single laser. The splitter is the interface between the VC Glitcher and two laser sources. Several splitters can also be concatenated to interface between the VC Glitcher and multiple laser sources.

The splitter is connected to the digital glitch, pulse amplitude and reset connectors of the VC Glitcher. The Splitter has digital glitch A & B, pulse amplitude A & B and analog glitch A & B outputs to control two attached lasers A and B.

The splitter divides the pulses from the VC Glitcher over the two lasers. The splitter counts the number of pulse that it receives from the digital glitch output. Based on the received number of pulses and two stored patterns for laser A and B, the next pulse will be transferred to laser A and/or B via the digital glitch A and B output of the Splitter. The pulse count is reset to 0 by a reset pulse coming from the VC Glitcher.

The Splitter also controls the laser power of laser A and B. The Splitter has two programmable voltage dividers for laser A and B. These voltage dividers reduce the pulse amplitude voltage from the VC Glitcher to a lower voltage on the pulse amplitude outputs A and B. The power of lasers A and B are always lower than or equal to the laser power as defined by the VC Glitcher.

Figure 6.34. The Splitter

6.4.3.1 Setup and use

The following connection should be made between the VC Glitcher and the Splitter:

- The Digital Glitch out of VC Glitcher to Digital Glitch in of Splitter
- The Pulse Amplitude out of VC Glitcher to Pulse Amplitude in of Splitter
- The Reset out of VC Glitcher to Reset in of Splitter

The following connection should be made between the Splitter and a 1064 / 808 nm Laser:

- The Digital Glitch A / B out of Splitter to Digital Glitch in of Laser
- The Pulse Amplitude A / B out of Splitter to Pulse Amplitude in of Laser

The following connection should be made between the Splitter and Laser that is controlled by an analog glitch (laser power is controlled by the height of the pulse):

- The Analog Glitch A / B out of Splitter to control input of Laser

The Splitter is powered and controlled via USB by the PC running Inspector

The tools menu in Inspector provides access to the Splitter control window. Please review documentation regarding the Splitter control in this manual.

6.4.3.2 Firmware upgrade

The Splitter is ready to receive firmware upgrades

6.4.3.3 Specifications

Specification per indicator or connector:

Front side

- Digital glitch A: switches laser A ON. Voltage output range 0 – 3.3 V.
- Pulse amplitude A: sets power level of laser A at 0 – 100%. Voltage output range 0 – 3.3 V.
- Analog glitch A: switches laser A ON and sets power level at 0 – 100%. Voltage output range 0 – 3.3 V.
- Digital glitch B: switches laser B ON. Voltage output range 0 – 3.3 V.
- Pulse amplitude B: sets power level of laser B at 0 – 100%. Voltage output range 0 – 3.3 V.
- Analog glitch B: switches laser B ON and sets power level at 0 – 100%. Voltage output range 0 – 3.3 V.

Back side

- Red led: indicates Splitter is powered
- Blue led: indicates configuration is stored
- Digital pulse: 50 Ohms input Impedance. Voltage input range 0 – 3.3 V. Minimum pulse width 20 ns, minimum pause time inbetween pulses 20ns.
- Pulse amplitude: 1 kOhms input impedance. Voltage input range 0 – 3.3 V.
- Reset: 1 kOhms input impedance. Voltage input range 0 – 3.3 V. Resets digital pulse counter
- USB connector for power and control

6.4.4 EM-FI Transient Probe

The EM-FI probe induces fast, high power, EM pulses on a user-defined location of the chip. The probe can be attached to either the EM Probe Station (see Section 6.3.4.2, “XYZ table”) or on the Diode Laser Station (see Section 6.4.2, “Diode Laser Station”). Though these XY(Z) controllers were originally designed for (respectively) EM acquisition and optical perturbation, they can both be used for EM-FI without any problems; Inspector sees the XY(Z) controller and the acquisition/perturbation source as separate and independent devices.

This section will describe the technical details of the EM-FI Transient Probe. For information on the *spatial* aspects of EM-FI attacks in Inspector we refer to Section 5.5, “XY(Z) Perturbation (Optical/EM-FI)”.

Specifications

The EM-FI transient probe comes with three connectors.

1. **Pulse amplitude** The VC Glitcher 'pulse amplitude' output can be connected to this probe input. The analog input is used to set the power of the pulse.
2. **Digital glitch** The VC Glitcher 'digital glitch' output can be connected to this probe input. The digital input is used to trigger the EM pulse.
3. **Coil current** The oscilloscope can be connected to this output. The analog output signal is used to monitor the current through the probe coil.

Figure 6.35, "The EM-FI probe tips" shows three of the four EM-FI probe tips. Table 6.5, "EM-FI probe tip specifications" gives the specifications of these. The 1.5 mm tips are used for 0 - 1.5 mm distance to the target, and the 4 mm tips are used for 1.5 - 4 mm distances to the target. Black and red tips have opposite polarity.

Figure 6.35. The EM-FI probe tips



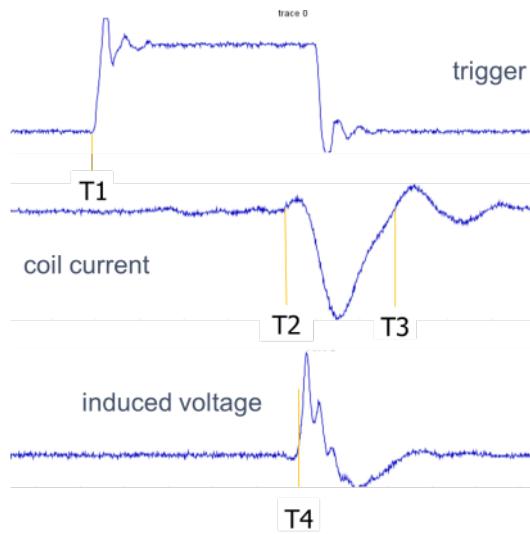
Table 6.5. EM-FI probe tip specifications

| Probe | Color | Diameter | Polarity |
|-------|-------|----------|----------|
| 1 | Red | 1.5 mm | Positive |
| 2 | Black | 1.5 mm | Negative |
| 3 | Red | 4 mm | Positive |
| 4 | Black | 4 mm | Negative |

Figure 6.36, "EM-FI probe behavior when glitch pulse width is 50ns" shows EM-FI behavior under different conditions with a 1.5mm positive polarity tip.

- The first line is digital glitch on the "digital glitch" input;
- The second line is the voltage measured over "coil current" from EM-FI transient probe with the "pulse amplitude" set at 0.33 V corresponding to 10% power;
- The third line is the output voltage of an EM antenna placed near the coil of the EM-FI transient probe.

The fixed delay (T1 to T4) between trigger signal (starts at T1) and the induced voltage of the EM glitch (starts at T2/T4 and ends at T3) is introduced by the electronic circuit and has a length of 50ns. The trigger timing (T1) and trigger width can be set by users via the perturbation parameters that control the VC Glitcher while pulse start T2 and T4 and pulse end T3 are fixed relative to trigger start T1. The strength of EM Glitch can be set by users by setting the voltage on "Pulse amplitude" input.

Figure 6.36. EM-FI probe behavior when glitch pulse width is 50ns

6.5 Signal filtering

This section describes components used for signal filtering and processing.

6.5.1 icWaves

The icWaves is developed to improve timing accuracy and reduce trace lengths by providing a trigger on hardware pattern matching. The FPGA-based device generates a trigger pulses after real-time detection of one or two distinctive patterns in the power or EM signal of a chip. In addition, the device has a special narrow band-pass filter built in to enable the detection of such a pattern, even in very noisy signals. The latter is important because side channel signals can be very noisy and detecting a pre-defined pattern is therefore not always feasible, at least, not without a tunable filtering mechanism.

Figure 6.37. The icWaves

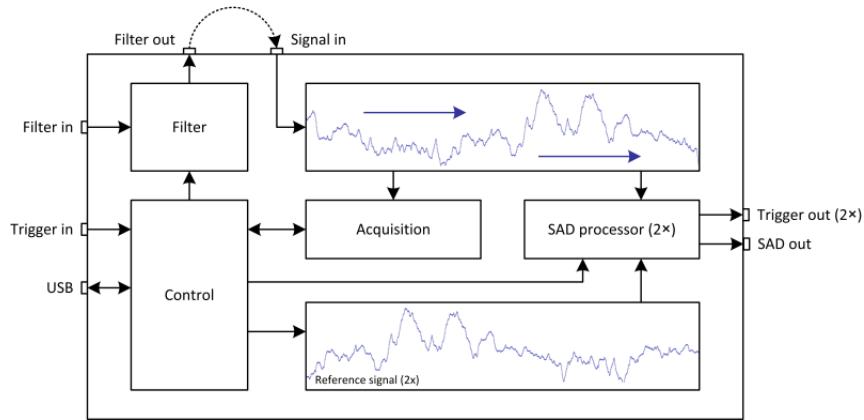
6.5.1.1 Functional overview

icWaves offers the following benefits:

- Efficient and repeatable fault injection by accurate and real-time detection of any waveform
- Efficient side channel analysis: Reduction of both the acquisition window size and alignment problems by improving the measurement quality on smart cards with significant time variations. Increase of signal quality by enabling a higher sampling speed.
- Enables side channel testing of devices that do not provide easy access to external trigger points such as I/O or other events.

icWaves works on smart cards and embedded chipsets, and can be used for improving your power and electromagnetic analysis (DPA, DEMA) and perturbation attacks with laser, voltage or clock glitching. The features of icWaves are controlled either from the software of the Inspector platform or by using the icWaves SDK. Further, it is interoperable with all existing Inspector hardware components.

The icWaves device generates a trigger output based on a real-time comparison of the input signal with one or two reference patterns. To that end, a Sum of Absolute Differences algorithm is implemented using FPGA technology. A conceptual overview of the device is shown in Figure 6.38, “Conceptual overview of the icWaves”.

Figure 6.38. Conceptual overview of the icWaves

In addition to the pattern detection capabilities, icWaves offers a narrow-band configurable filter which can be useful to highlight interesting frequency ranges in side channel signals. For instance, the frequency band around an internal clock could be selected in a noisy signal aiming to obtain better reference patterns.

Real-Time pattern matching

In order to detect the reference pattern in the input signal, icWaves uses a pattern matching technique known as the *Sum of Absolute Differences* (SAD). This is a simple metric for similarity also used for motion detection in MPEG and other image processing algorithms.

When a reference signal is stored in icWaves, the device will compare input signals with it. To that end, the absolute difference between the samples from the input signal and the reference signal is computed, and the resulting samples are summed up. This is illustrated in Figure 6.39, “Sum of Absolute Differences illustrated”.

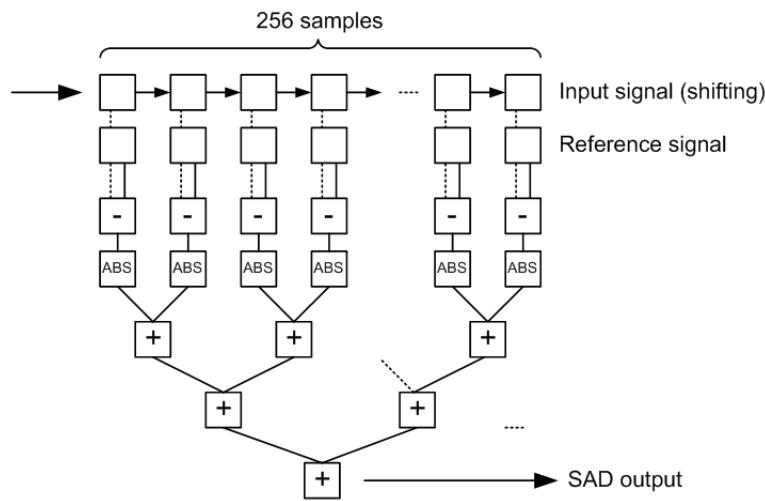
Figure 6.39. Sum of Absolute Differences illustrated

| | | | | | | |
|-----------|-----|-----|------|-----|-----|-------------|
| A | -26 | -1 | -52 | 9 | 115 | -71 |
| B | 60 | 96 | 54 | -72 | -82 | 118 |
| Diff. | -86 | -97 | -106 | 81 | 197 | -189 |
| ABS diff. | 86 | + | 97 | + | 106 | + |
| | | | | 81 | + | 197 |
| | | | | | + | 189 |
| | | | | | | → SAD = 756 |

| | | | | | | |
|-----------|----|-----|-----|----|----|------------|
| A | 74 | 111 | 123 | 64 | 58 | -35 |
| B | 78 | 110 | 128 | 66 | 54 | -39 |
| Diff. | -4 | 1 | -5 | -2 | 4 | 4 |
| ABS diff. | 4 | + | 1 | + | 5 | + |
| | | | | 2 | + | 4 |
| | | | | | + | 4 |
| | | | | | | → SAD = 20 |

A low SAD means that the signals are similar, while a higher value means that they are not so similar. Therefore, the SAD result can be used to differentiate similar signals from non-similar signals and detect distinctive patterns.

This technique is implemented by icWaves hardware in a staged architecture. For each sample pair, the absolute difference is computed. Then, an adder chain sums up the resulting absolute differences two by two. Eventually, only one result remains, which is the output SAD. This implementation is illustrated in Figure 6.40, “Hardware SAD implementation”.

Figure 6.40. Hardware SAD implementation

Filtering

Due to noise, or useless but dominant disturbance signals, it may be very difficult to identify weak signals that may be used for triggering. Also, useful signals may be too fast to be effectively sampled by the icWaves device at its maximum sampling speed of 100Msamples/s.

The icWaves built-in filter has therefore two purposes:

- To highlight weak signal features related to part of the spectrum, which are hidden by other stronger signals.
- To shift the spectrum of those features to a lower range that can be sampled.

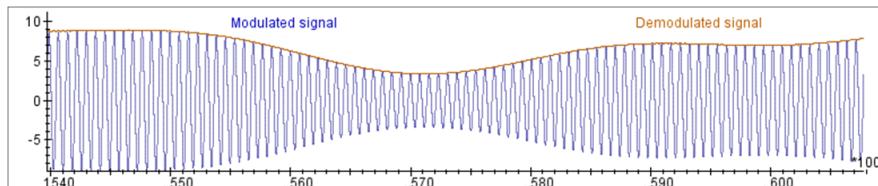
The filter is designed to show the amplitude variation of a crypto processor power or EM signal when the crypto processor is linked to a clock with little clock jitter. The filter attenuates any signal outside a selectable frequency range of approximately 2 MHz bandwidth, starting at approximately 1 MHz below the centre filter frequency (f_{filter}) to approximately 1 MHz above.

This filter is also applicable to crypto processors with an internal clock that differs from the external clock. The filter performs at its best with the following conditions:

- Internal clock between 10 MHz and 400 MHz,
- No other strong signals within the selected frequency range, from $f_{\text{filter}} - 1$ MHz to $f_{\text{filter}} + 1$ MHz,
- Variation of the signal amplitude of at least 20%,
- Duration of an amplitude peak or an amplitude dip of at least 10 us.
- Tune the filter frequency f_{filter} to approximately $f_{\text{clock}} - 0.5$ MHz or $f_{\text{clock}} + 0.5$ MHz. Tuning the filter frequency f_{filter} exactly to the clock frequency f_{clock} usually leads to an additional random low frequency signal caused by the random phase difference between the clock signal and the mixer signal inside the filter.
- Clock jitter should be less than 0.5 MHz.

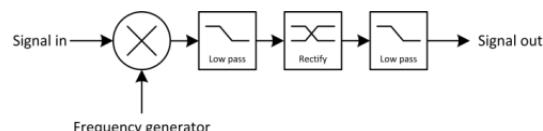
The filter is based on a common architecture used for demodulation in AM radio receivers where a carrier wave is removed from a modulated signal and the envelope signal remains (see Figure 6.41, “Signal demodulation”).

Figure 6.41. Signal demodulation



A block diagram of the filter stage is shown in Figure 6.42, “Mixer design”. First, a sine wave of the desired frequency is generated. Then it is mixed with the received signal, low pass filtered, rectified (its absolute value is obtained) and low pass filtered again. The first low pass filter has its cut-off frequency at 1 MHz, and serves to remove frequency components far from the selected band. The second low pass filter has its cut-off frequency at 1 MHz, and serves to smoothen the dips from the rectification process.

Figure 6.42. Mixer design

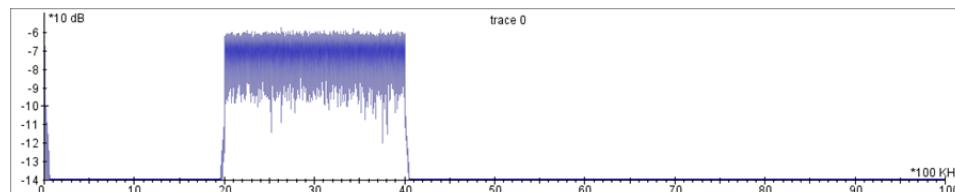


By means of the digitally controlled sine wave generator, capable of generating sine waves in frequencies ranging from 0 to 400 MHz, the interesting frequency band can be selected.

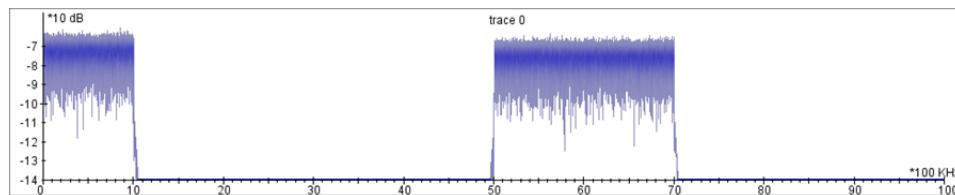
After mixing the generated tone with the input signal, a low pass filter is applied to the resulting signal. This process has an effect similar to a tunable band pass filter followed by a frequency conversion step: a low pass signal is generated with components from the selected frequency range.

To illustrate this process, Figure 6.43, “Random noise uniformly distributed in 2-4 MHz” shows the spectrum of a signal composed of random noise uniformly distributed in the 2MHz 4 MHz band:

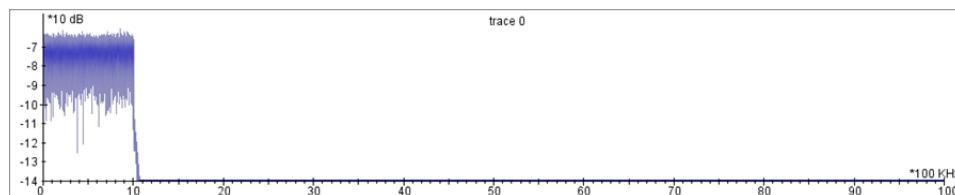
Figure 6.43. Random noise uniformly distributed in 2-4 MHz



If we multiply this signal with a 3 MHz sine wave, the resulting signal will contain the spectrum of the original signal shifted 3 MHz to the left (i.e. will be centered at 0 Hz) plus the spectrum of the original signal filtered 3 MHz to the right (i.e. centered at 6 MHz). The resulting spectrum is shown in Figure 6.44, “Example signal after frequency translation”.

Figure 6.44. Example signal after frequency translation

After this a low pass filter is used to keep only the interesting part: the conversion of the original spectrum to the base band at 0 Hz. See Figure 6.45, “Example signal after frequency translation and low pass filter”.

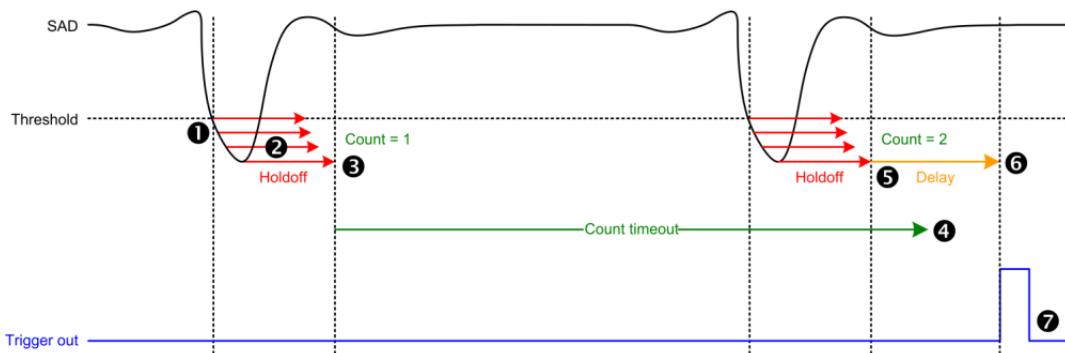
Figure 6.45. Example signal after frequency translation and low pass filter

Therefore, the selected frequency band starts at the frequency tuned with the digital tone generator and spans a frequency range equal to the bandwidth of the external low pass filter applied.

Triggering

The icWaves trigger outputs can be configured using the following settings.

- Threshold
- Holdoff
- Count
- Count timeout
- Delay

Figure 6.46. Trigger settings

A trigger from the icWaves is generated in 7 steps, as shown in the Figure 6.46, “Trigger settings”.

1. icWaves detects that the SAD signal drops under the specified threshold. It proceeds to the next step.
2. If a holdoff time is set to x , the trigger will be held off for x ns. When a better match is found in the next x ns, the holdoff timer is reset to the specified holdoff time x . If no better match is found for x ns, the trigger logic proceeds to the next step.
3. The pattern counter is increased. If the pattern counter is equal to the specified count value, the trigger logic proceeds to the next step. Otherwise it waits for another pattern (see step 1).
4. If the pattern counter was not increased in the last x ns (where x is the value of the "Count timeout" parameter), the pattern counter is reset to 0.
5. When the pattern counter reaches the specified count value, the trigger logic proceeds to step 6.
6. Generation of the trigger is delayed with x ns (where x is the value of the "Delay" parameter).
7. A 1us trigger pulse is generated.

When the icWaves is monitoring two different patterns, both patterns can be configured separately.

6.5.1.2 Setup and use

As shown in Figure 6.38, "Conceptual overview of the icWaves", the following ports are available in icWaves:

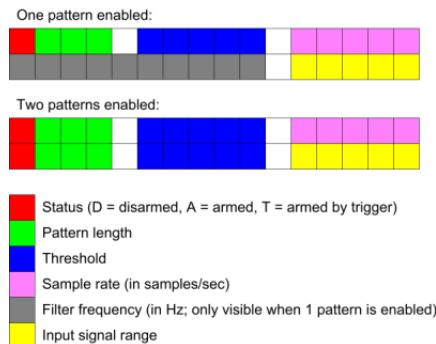
- *Filter in*: input for the configurable filter stage. This is an optional stage, which can be used to highlight weak signal features by filtering a specific frequency band in the input signal. This is particularly useful for high bandwidth and noisy signals.
- *Filter out*: The output of the filter stage. When using the filter stage, it must be connected to the *Signal in* port.
- *Signal in*: this is the input for the pattern detection feature of icWaves. Therefore, this input is always used and connected.
- *SAD*: This port provides the *Sum of absolute differences* signal of the first pattern. It can be used for checking what icWaves is actually computing. It's not recommended to use this output for selecting a proper threshold. Instead, use the icWaves Simulate module (see Section E.10.1, "Simulate") to simulate the behaviour of the icWaves from within Inspector.
- *Trigger in*: the trigger input for icWaves. This can be used when acquiring traces using icWaves as an oscilloscope. These traces can later be used to define a reference trace.
- *Trigger out 1/2*: Trigger outputs of icWaves. It produces trigger pulses for the oscilloscope when a reference pattern is detected (i.e. when the SAD value is below the specified threshold for that pattern).

Thanks to these capabilities, icWaves is able to detect up to two patterns in very noisy signals and trigger oscilloscopes at the right moment, preventing misaligned traces.

This offers several benefits with respect to the usual time-based triggering. A reduction of the acquisition window is possible, and therefore also a sample rate increase; further, it helps to reduce the processing needed for side channel acquisitions due to the reduction of misalignment.

icWaves shows relevant information on its blue 16x2 characters LCD display. Figure 6.47, “LCD display” describes the layout of the information shown by icWaves at any time in this display:

Figure 6.47. LCD display



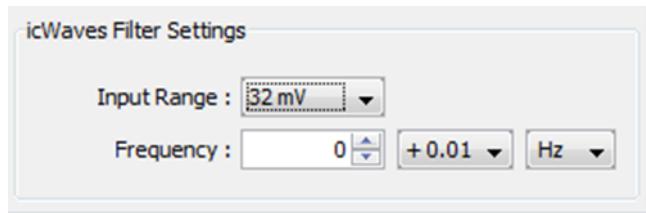
6.5.1.3 Software interface

The icWaves software is seamlessly integrated with the Inspector Platform. The software consists of a device driver included in the Inspector Platform core, plus the icWaves configuration dialog for managing and configuring the device (see Section 6.5.1.3, “icWaves Configuration dialog”).

The *icWaves Scope* can be used together with Inspector's acquisition modules to acquire a reference trace as it will be seen by icWaves.

When *icWaves Scope* is used in an acquisition or perturbation experiment, the following icWaves filter properties are visible:

Figure 6.48. icWaves filter properties



Using these settings, the icWaves filter can be enabled and configured. Additionally, it is possible to tune the filter settings real-time by selecting the *Show filter properties dialog during acquisition* checkbox.

After acquiring the reference trace, a reference pattern can be loaded into icWaves using the icWaves configuration dialog.

The icWaves configuration dialog can also be used to tune the icWaves threshold and frequency settings during an acquisition or a perturbation experiment.

Since Inspector 4.6 it is possible to add icWaves feedback in perturbation logs (see Section E.12.1, “SC Perturbation”). When this feature is being used, the icWaves device used by this feature is locked by the perturbation module. Therefore, tuning the settings for the selected can only be achieved through the same perturbation module.

In the configuration dialog, a threshold for the SAD value must be selected. SAD values below this threshold will result in a trigger pulse generated in the *Trigger out* port of icWaves.

Inspector also includes a module to simulate the action of the SAD output. Although this signal is also available as an output of icWaves, it is useful to be able to compute the SAD trace from an acquired trace set.

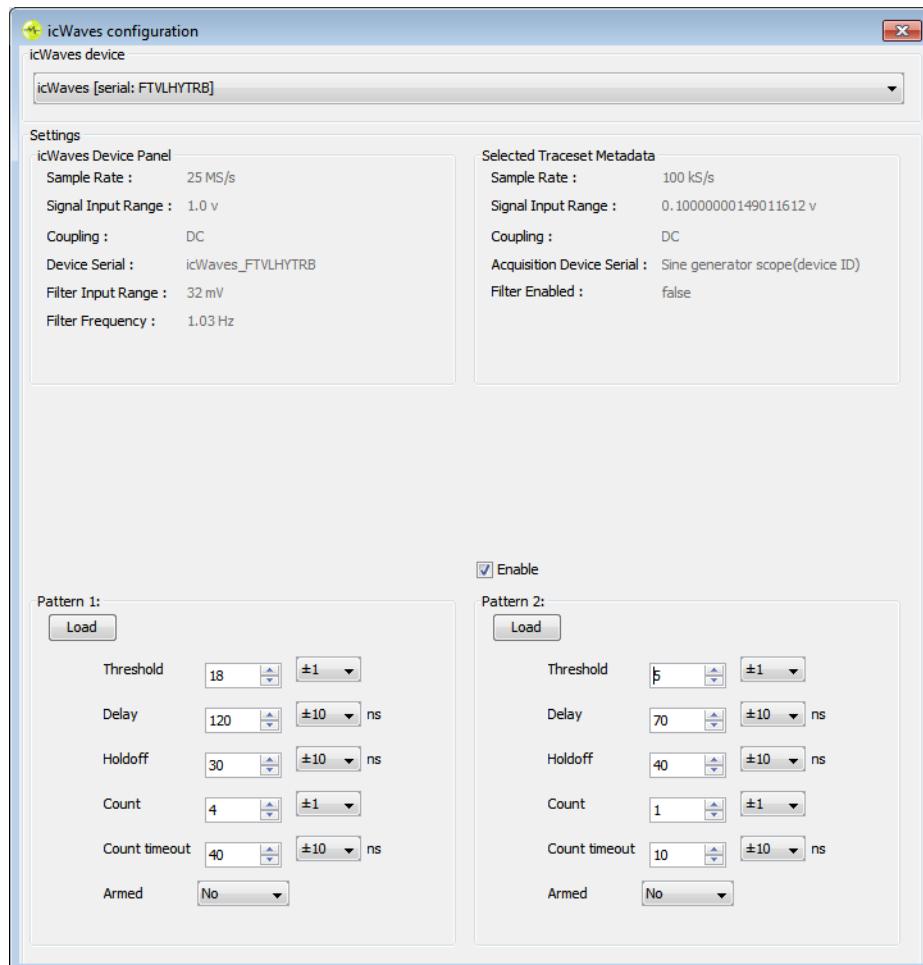
Using the Section E.10.1, “Simulate” module a user can also generate SAD traces for several input traces and decide on the threshold value to be used for differentiating the right pattern.

See the tutorials provided by Riscure together with icWaves for more information on the software, including detailed explanations of all its parameters as well as hands on exercises.

icWaves Configuration dialog

The icWaves configuration dialog is available through the Tools menu (see Section 2.6, “Tools menu”).

Figure 6.49. icWaves configuration dialog



The *icWaves device* drop-down list allows selecting which icWaves to configure. This list is automatically filled in by Inspector upon connection of new icWaves devices.

Once an icWaves device is selected, the *icWaves Device Panel* shows the current device settings:

- The sample rate configured in the device.
- The signal input range configured in the device.
- The filter input range configured in the device.
- The filter frequency configured in the device.

With the exception of the device version, these settings relate to the last acquisition performed using this device as an oscilloscope or to the last reference pattern loaded on the device.

Additionally, two identical panels allow configuring the icWaves pattern detection features. For each pattern, it is possible to enable or disable it by using a checkbox. Additionally, when the pattern is enabled and a reference trace is selected in Inspector, it is possible to press the *Load* button in order to transfer the reference pattern to icWaves.

Finally, a number of configuration settings are available for each pattern. These options can be modified at any time, including during an acquisition or perturbation experiment. These settings correspond to the parameters described in the section called “Triggering”.

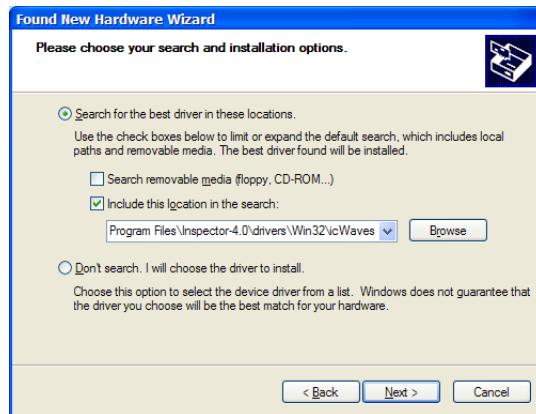
6.5.1.4 Driver installation

- First of all, make sure icWaves is installed correctly.
- Connect icWaves to the PC using the supplied USB cable and to the supplied power supply.
- Windows will detect the device and shows the dialog shown below. If Windows asks to search for a driver online, select *no*.

Figure 6.50. Installation dialog



- Select *Install from a list or specific location (Advanced)* and click *Next*.

Figure 6.51. Installation dialog

- Select *Search for the best driver in these locations*, deselect *Search removable media* and select *Include this location in the search*. Enter C:\Program Files\Inspector-4.0\drivers\Win32\icWaves in the text box and click *Next*.

Figure 6.52. Installation dialog

- If Windows warns about the fact that the software did not pass Windows Logo testing, click *Continue Anyway*.

Windows will find another icWaves device. Repeat the same steps again for this device.

6.5.1.5 Firmware upgrade

First of all, make sure the icWaves is connected and installed as described in the previous section. Then run the icWaves upgrade tool (i.e. icwaves_firmware_x_y.exe (where x and y represent the version number)).

Figure 6.53. icWaves firmware upgrade tool

Click the *Install* button to upgrade the firmware. This will take approximately 3 minutes.

6.5.1.6 Software Development Kit (SDK)

Although most users will use the icWaves from Inspector, Riscure provides a Software Development Kit (SDK) which allows the icWaves to be integrated in other tools (e.g. side channel analysis or fault injection tools).

The SDK contains an API that exports several C functions which can be used to control all features of the icWaves. The API can be used with any programming environment that supports standard C calls.

Programs that use the icWaves typically perform the following tasks:

- Open the device.
- Optionally, configure the internal bandpass filter.
- Acquire some reference traces.
- Select one or two patterns from the reference traces.
- Configure the patterns.
- Arm the device.
- When finished, close the device.

Installation

Execute the icWaves installer and select the desired destination folder. In the examples in this section, we will use C:\Projects\icwaves-sdk. The installer creates the following files:

- icwaves_api.h (C header file)
- icwaves.lib (library file)
- icwaves.dll (dynamic link library)
- icwaves_api.pdf (detailed description of the functions exported by the icWaves API)
- icwaves_user_manual.pdf (this document)

- icwaves_example.c (simple C example program)
- Uninstall.exe (Uninstaller)

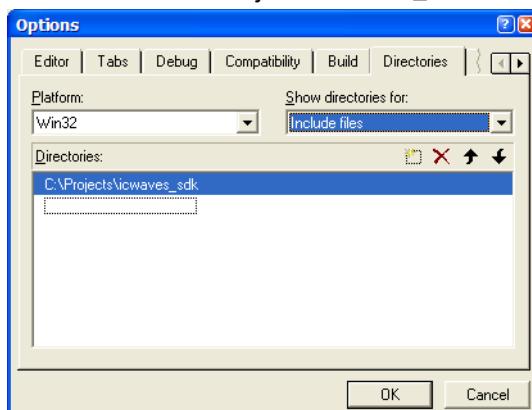
There are two different versions of the library file and the dynamic link library, and they are stored in the folders with_gui and without_gui respectively. The difference between them is that the version with GUI shows a window with information about the current state of icWaves after initializing the device.

Using the SDK in your programming environment

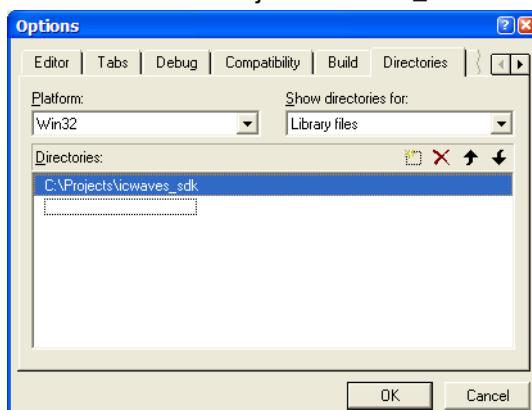
In this section, the configuration for several C programming environments is described. For other programming environments, the procedure should be similar. The examples below assume the icWaves SDK is installed under C:\Projects\icwaves_sdk. If you choose another location, please change the folder names accordingly.

Visual C++ 6

- Click Tools -> Options.
- Select the Directories tab.
- In the "Show directories for" combo box, select "Include files".
- Add the folder C:\Projects\icwaves_sdk.

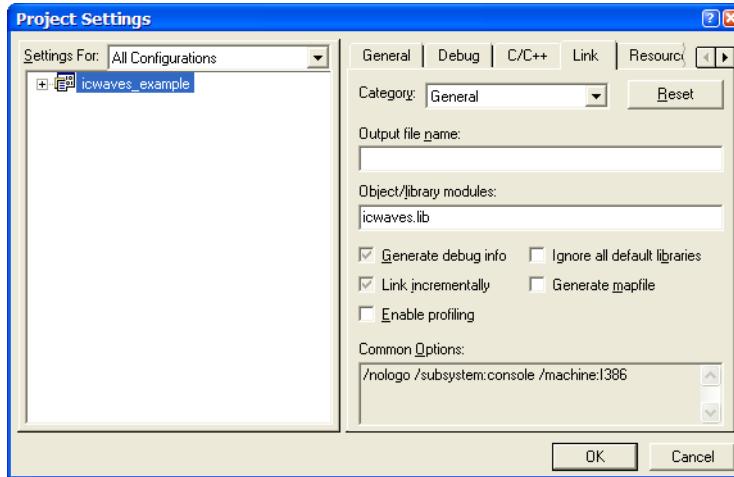


- Now select "Library files" in the "Show directories for" combo box.
- Add the folder C:\Projects\icwaves_sdk.



- Click OK.

- Click Project -> Settings...
- In the "Settings For:" combo box, select "All configurations".
- Select the Link tab.
- Add "icwaves.lib" at the end of the "Object/library modules" textfield.

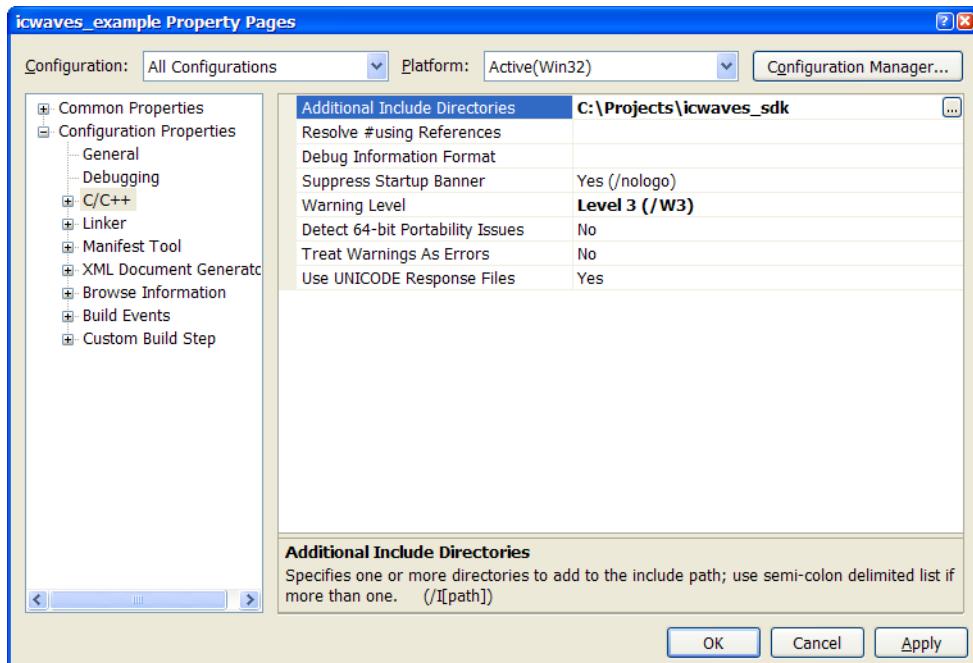


- Click OK.

You can now use all the functions exported by the icWaves API.

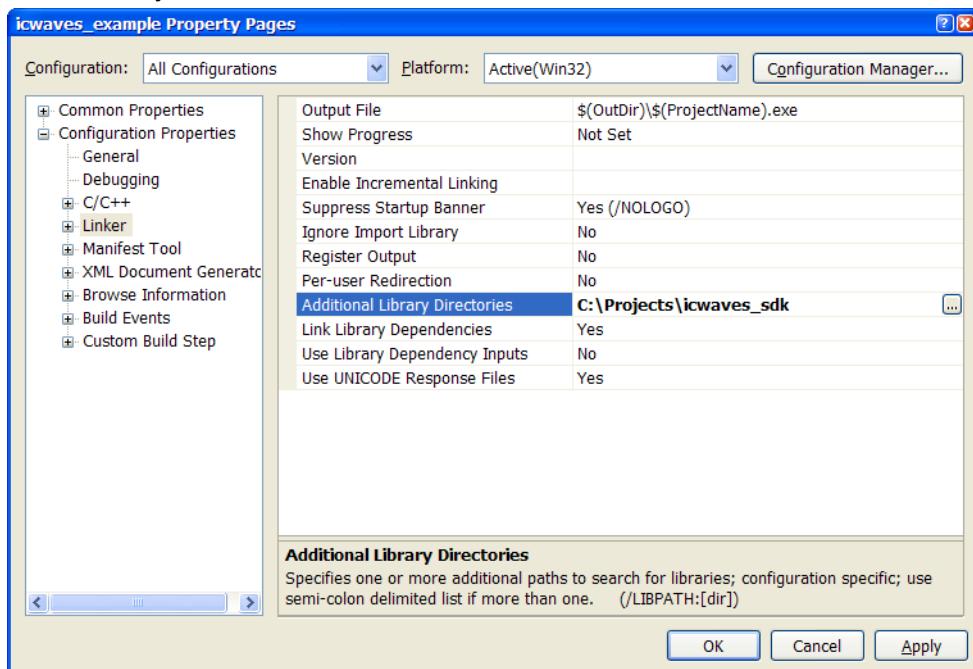
Visual C++ 2008

- Project -> Properties
- In the "Configuration:" combo box, select "All configurations".
- Under Configuration Properties -> C/C++, select Additional Include Directories and add the folder C:\Projects\icwaves_sdk.

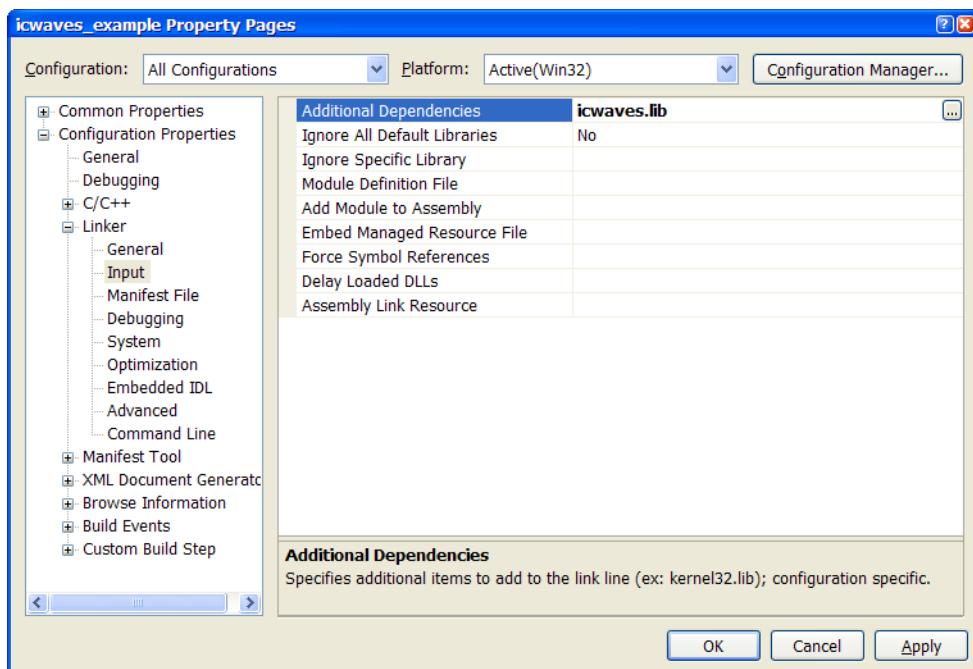


Inspector Hardware Components

- Under Configuration Properties -> Linker, select Additional Library Directories and add the folder C:\Projects\icwaves_sdk.



- Under Configuration Properties -> Linker -> Input, select Additional Dependencies and add icwaves.lib.

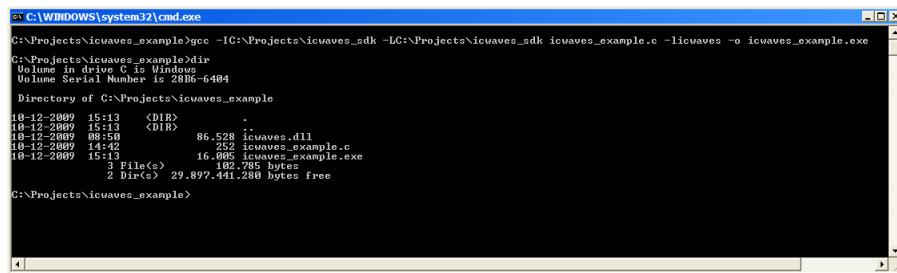


You can now use all the functions exported by the icWaves API.

MinGW (GCC for Windows)

In order to compile the test program with MinGW, use the following command on the command prompt:

```
gcc -IC:\Projects\icwaves_sdk -LC:\Projects\icwaves_sdk icwaves_example.c -licwaves
```



```
C:\>C:\WINDOWS\system32\cmd.exe
C:\Projects\icwaves_example>gcc -IC:\Projects\icwaves_sdk -LC:\Projects\icwaves_sdk icwaves_example.c -licwaves -o icwaves_example.exe
C:\Projects\icwaves_example>dir
Volume in drive C is Windows
Volume Serial Number is 2B86-6494
Directory of C:\Projects\icwaves_example
10-12-2009 15:13 <DIR> .
10-12-2009 15:13 <DIR> ..
10-12-2009 08:58 86,528 icwaves.dll
10-12-2009 15:13 16,095 icwaves_example.c
10-12-2009 15:13 16,095 icwaves_example.exe
               3 File(s)    102,785 bytes
               2 Dir(s)   29,897,441,280 bytes free
C:\Projects\icwaves_example>
```

6.5.1.7 Specifications

icWaves has the following features:

1. Generates trigger pulses on *trigger out 1/2* based on analog signal at *Signal input* terminal in three steps:
 - a. It operates as an oscilloscope in Inspector to store a reference trace.
 - b. The user can select a pattern from this reference trace and configure icWaves to compare the analogue *Signal in* signal with the reference pattern.
 - c. icWaves can now be used as a trigger source. When the reference pattern is detected in the real-time signal a trigger pulse will be generated.
2. 2 MS memory depth for acquiring a reference trace:
 - icWaves versions 1.x - 2.2. support max. 2M samples per trace.
 - icWaves versions 2.3+ support max. 8M samples per trace
3. Sample speed:
 - icWaves versions 1.x - 2.2. support max. 100 MS/s with 8 bit resolution.
 - icWaves versions 2.3+ support max. 200 MS/s with 8 bit resolution.
4. The reference signal(s)
 - for icWaves versions 1.x - 2.2. can contain up to 1x256 or 2x128 samples
 - for icWaves versions 2.3+ can contain up to 1x512 or 2x256 samples.
5. The comparison between the *Signal in* signal and the reference signal is made using the *Sum of Absolute Differences* (SAD) over the samples. The real-time calculated SAD-signal of the first pattern is available at the SAD connector.
6. AC/DC:
 - icWaves versions 1.x - 2.2. suppress only AC coupling.
 - icWaves versions 2.3+ suppress AC/DC coupling.
7. TTL-level trigger outputs (*Trigger out 1/2*)
 - Configurable hold off time (10 ns resolution)

- Configurable delay (10 ns resolution)
 - Possibility to specify number of patterns to skip before trigger
 - Fixed pulse length of 1 us
8. The centre frequency of band-pass filter is programmable between 0 and 400 MHz.
9. The Inputs/Output connectors have the following features:
- *Filter in*: analog input signal for tunable band-pass filter, signal level 1 V_{p-p}, 50 Ohm. The signal level within the filter band (from $f_{\text{filter}} - 1 \text{ MHz}$ to $f_{\text{filter}} + 1 \text{ MHz}$) should be lower than 256 mV_{p-p}, 128 mV_{p-p}, 64 mV_{p-p}, 32 mV_{p-p} depending on the settings of the filter input range: 128 mV, 64 mV, 32 mV or 16 mV respectively.
 - *Filter out*: analog output signal from tunable band-pass filter, signal level 1 V_{p-p}, 50 Ohm
 - *Signal in*: analog input signal for the ATG, signal level 2 V_{p-p}, 50 Ohm. The range of the input signal can be set to 1000 mV, 500 mV or 250 mV.
 - *SAD out*: output signal from DAC representing real-time SAD value of the first pattern, signal level 1V_{p-p}, 50 Ohm
 - *Trigger in*: TTL-level trigger input
 - *Trigger out 1/2*: TTL-level trigger pulse outputs



Heat Warning

The icWaves generates heat during operation. To prevent the device from overheating, please make sure that ventilation holes are not obstructed to allow free air flow.

Turn the icWaves off when it is not in use

6.5.1.8 Prevent card damage using VC Glitcher and icWaves

Optical fault injection is prone to cause permanent card damage. Countermeasures can be triggered, but also random code execution can corrupt the code stored in EEPROM. A measure to prevent this is to use the icWaves to detect and generate a trigger upon continued normal operation of a card after firing the laser. The VC Glitcher can be programmed to reset the card if normal operation is not detected, thereby reducing the probability of permanent card damage.

6.5.2 CleanWave

Functional overview

Contactless cards are powered by a RF field. To provide sufficient power, even when the card is at a distance of approximately 10 cm to the card reader, the RF field is relatively strong. The RF field is primarily a sinusoidal varying magnetic field with a frequency of 13.56 MHz. Due to disturbance by the card reader or contactless smart card, the RF field also contains higher harmonics at 2, 3, 4, .. times the fundamental frequency of 13.56 MHz.

Figure 6.54. The CleanWave

The two most popular side channels for contactless cards is the EM emanation of the chip (EMA-RF) and the variation of the RF field due to the power consumption by the card (Radio Frequency Analysis - RFA).

The EM emanation of the chip is a continuous process in time and is independent from the RF field (although the internal process clock may be derived from the RF field frequency). The EM emanations are measured by the EM probe together with the RF field. Usually the EM emanations are much smaller than the swing of the RF field.

The power consumption of a contactless smart card is not continuous in time. Due to the internal circuitry (a bridge rectifier is most common) the contactless card draws power when the RF field is at its maximum or minimum value. The power consumption of the card only influences the RF field at these moments. The RF field with variation is measured by an antenna. Usually, the variation is small compared to the swing of the RF field.

The analog probe or antenna signal is measured by a digital storage oscilloscope. The input range of the oscilloscope must exceed the large signal swing due to the RF field. For the EMA-RF and RFA signals, the side channel leakage appears as a small variation on top of a strong RF signal. The small side channel leakage is measured with a low resolution and a relative large quantization error.

The CleanWave improves the resolution and reduces the quantization error by reducing the signal swing due the RF field. The methodology to reduce the swing differs. The multi-notch filter in the CleanWave reduces the signal strength at the fundamental frequency and the 2nd, 3rd and 4th harmonics by using four notch filters. The demodulator rectifies the input signal and samples and the maximum peak of the rectified signal (two sampled per RF period) with a sample and hold circuit.

6.5.2.1 Setup and use

The CleanWave is powered by a 15 V DC power supply of at least 10 W. The power connector is on the back of the box. The green LED on the front of the box lights up when the CleanWave is powered.

The multi-notch filter of the CleanWave is designed for use in combination with the EM probe for EMA-RF. The demodulator is designed for use in combination with the RF antenna for RFA.

The multi-notch filter and demodulator share two input connectors. One input connector should be used when the amplitude of the input signal is below 1 V (or 2 V peak-to-peak). The second

input connector should be used when the amplitude is between 1 V and 5 V (or between 2V and 10 V peak-to-peak). For input signals with an amplitude above 5 V an attenuator must be used. For input signals with an amplitude below 100 mV an amplifier is recommended. When the input signal exceeds the voltage range (1V or 5 V amplitude) the red overload LED lights up.

The impedance of both inputs equals 50 Ohms. The EM probe or RF antenna can be connected directly via coaxial BNC cable to the multi-notch filter and demodulator.

It is recommended to insert a 50 MHz low-pass filter between EM probe and the input connector. The 50 MHz low pass filter reduces the 5th and higher harmonic of the RF field. Usually the data leakage in the EMA-RF signal of contactless cards is primarily in the frequency range from 0 to 50 MHz.

The multi-notch filter and demodulator circuits have each a 50 Ohms output that can be directly connected via coaxial BNC cable to the oscilloscope with 50 Ohms input impedance.

The CleanWave does not have an USB connector and does not require a software driver.

See Figure 6.55, “Connecting the CleanWave” for two examples of a CleanWave schematic setup. On the left there is a regular RFA setup with a MP300 device, an RF antenna, and an oscilloscope. On the right hand side the oscilloscope is replaced with an icWaves device. This setup can be used to acquire a reference trace. It is possible to connect both an oscilloscope and an icWaves device, as is depicted in Figure 6.56, “Connecting the CleanWave (continued)”.

Figure 6.55. Connecting the CleanWave

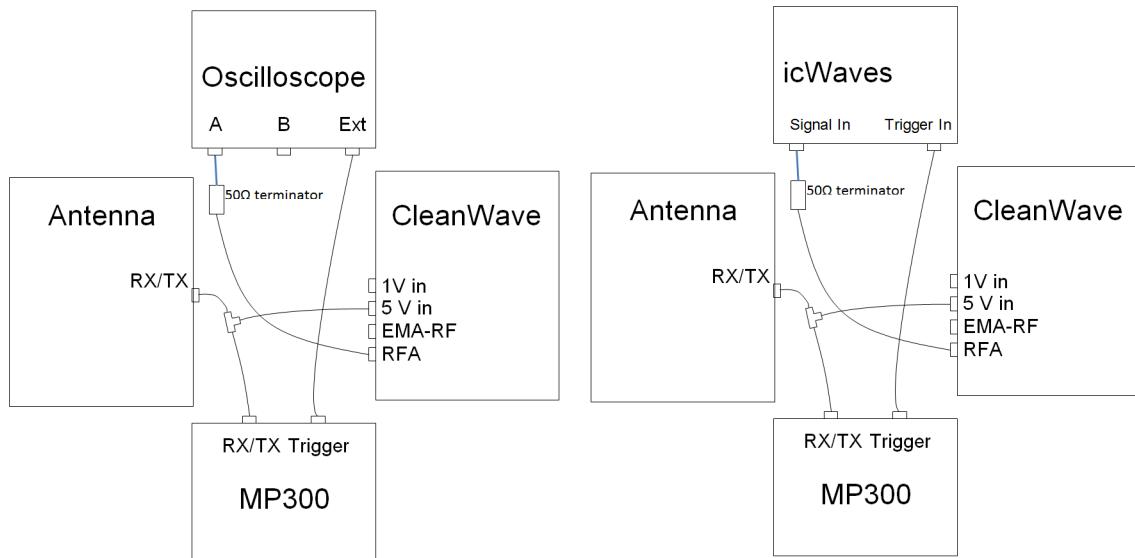
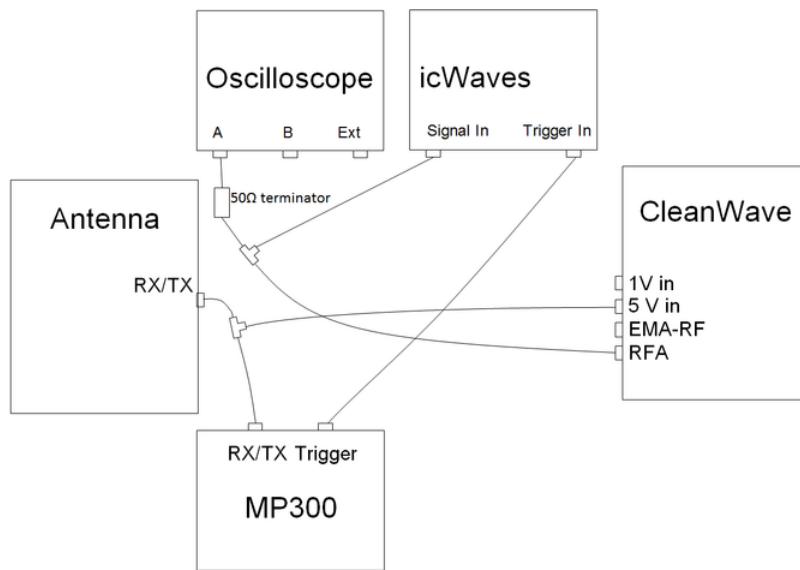


Figure 6.56. Connecting the CleanWave (continued)

6.5.2.2 Software interface

Two software modules are developed for the CleanWave.

- InvNotchFilter inveres the effect of the multi-notch filter while keeping the improved measurement resolution
- RFResample resamples the output signal from the demodulator to a resample frequency of 27.12 MHz. (two samples per RF period)

6.5.2.3 Specifications

- Power: 15 V DC, 10 W max
- Green LED indicates power ON
- 1V input connector: BNC, 50 Ohms impedance, AC-coupling, max voltage 2 V peak-to-peak
- 5V input connector: BNC, 50 Ohms impedance, AC-coupling, max voltage 10 V peak-to-peak
- Red LED indicates input overload
- Multi-notch filter output connector: BNC, 50 Ohms impedance, max 4 V peak -to -peak when terminated by 50 Ohms
- Gain of multi-notch filter:
 - 0 MHz: 0 dB
 - 11 MHz: +6 dB
 - 13.6 MHz: -27 dB

22.5 MHz: +3 dB

27.1 MHz: -13 dB

32.5 MHz: +4 dB

40.7 MHz: -16 dB

48 MHz: -3 dB

54.3 MHz: -16 dB

100 MHz: 0 dB

- Demodulator output connector: BNC, 50 Ohms impedance, max 2 V peak-to-peak when terminated with 50 Ohms
- Sample and hold interval of demodulator: 10 ns
- Sample frequency: 2 times fundamental frequency of RF carrier (typical 27.12 MHz, depends on input signal).

6.5.3 Hardware Filters

Depending on the ordered setup, hardware filters can be a part Inspector shipment. Hardware filters are used to attenuate the signal before acquisition.

Figure 6.57. Hardware filter



RF frequency filter

The RF frequency filter is 25 MHz High Pass Filter that is delivered with the purchase of an RF Tracer. This filter is useful to attenuate the RF carrier signal. It is connected between the EM probe (or the RFA output terminal of the contactless card reader) and the oscilloscope. The cut-off frequency of this high-pass filter is 25 MHz. At 13.56 MHz an attenuation of 40 dB is reached. A 40 dB attenuation corresponds to an amplitude attenuation by a factor 100. The RF filter is a BNC in line filter. The filter characteristic is shown in Figure 6.58, "Filter characteristic of 25 MHz high-pass filter".

Figure 6.58. Filter characteristic of 25 MHz high-pass filter

Low pass filters

The 50 MHz and 90 MHz low pass filters are supplied to avoid aliasing. Aliasing is the effect that a 75 MHz sine wave sampled with 100 Ms/s will give an undesired 25 MHz wave instead of the original 75 MHz signal. The Nyquist–Shannon sampling theorem says that if you are sampling at frequency f_s , then the input signal should be limited to $f_s/2$ and have no signal above $f_s/2$. You can limit the frequency range of the input signal by applying a low pass filter at $f_s/2$. In practice you select the cut-off frequency f_c of the low pass filter well below $f_s/2$. Hence, when sampling at 200 Ms/s (maximum sample rate of the PicoScope 3206) you select a low pass filter at 50 MHz ('MiniCircuits' BLP-50+).

Note the filter should be terminated by the 50 Ohms input impedance of the oscilloscope. Therefore, select the 50 Ohms input impedance on the LeCroy oscilloscope. The PicoScope 3205 and 5203 have a fixed input impedance of 1 M Ω hms. Therefore you should use the BNC-in-line 50 Ohms terminator ('Probe Master Inc' PM1024 50ohms pass-through unit). The overview below should clarify things:

For LeCroy oscilloscope:

- Power Tracer v3.0 with 50 Ohms output impedance -> 50 Ohms BNC cable -> low pass filter -> 50 Ohms input impedance of LeCroy oscilloscope.

or

- EM probe with 50 Ohms output impedance -> 50 Ohms BNC cable -> low pass filter -> 50 Ohms input impedance of LeCroy oscilloscope.

For PicoScope 3206 or 5203:

- Power Tracer v3.0 with 50 Ohms output impedance -> 50 Ohms BNC cable -> low pass filter -> 50 Ohms terminator-> 1M Ω hms input impedance of PicoScope oscilloscope.

or

- EM probe with 50 Ohms output impedance -> 50 Ohms BNC cable -> low pass filter -> 50 Ohms terminator-> 1M Ω hms input impedance of PicoScope oscilloscope.

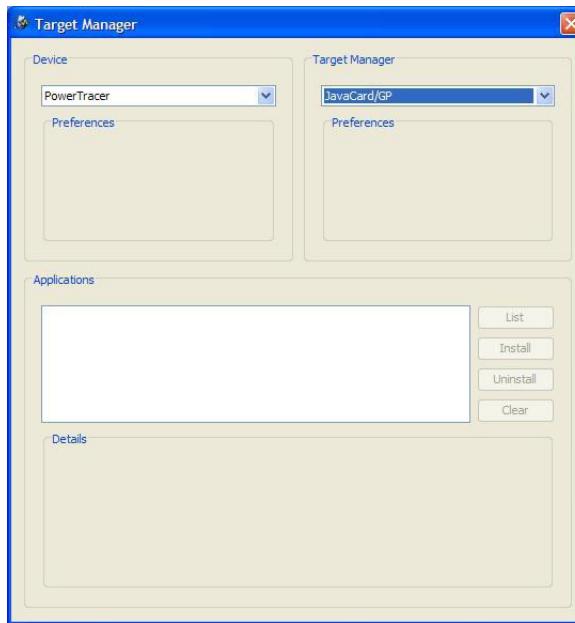
6.6 Managing Targets

This section describes the setup for connecting and using the target of evaluation (TOE) with the Inspector workstation.

6.6.1 TargetManager

The Target Manager in Inspector allow the user to prepare a Target of Evaluation (TOE) from within Inspector. The Target Manager can be found in the Tool menu.

Figure 6.59. Main dialog of the Target Manager



First select a device which is connected to the target. After the device has successfully opened select the appropriate target manager.

Clicking the list button will show a list of installed applications on the target (if supported by the target manager). Selecting install and uninstall will allow the user to manage the configuration of the target. The clear button will clear the target of all configuration.

6.6.2 Java Card as target

This section describes the setup of connecting a Java Card to Inspector and use it as a target.

Only Java Cards supporting Global Platform (GP) are supported, which should cover all modern Java Cards. The Java Card (JC) target manager supports installing CAP files and removing instances/packages from cards.

6.6.2.1 Installing CAP files

A CAP file corresponds to a Java package which in GP terms is a (executable) load file. A Java package may contain several instantiable classes inheriting from JC's Applet class. These classes are called (executable) modules in GP. An instance of a class inheriting from

Applet is called an application in GP. In GP each of these three components (load files, modules, applications) can be identified by Application Identifiers (AIDs). The CAP file contains the AIDs of the load files and modules. The AID of an application can be set by the user during CAP file installation, but defaults to the same AID as the module from which it is instantiated.

Installation of a CAP file on a Java Card actually consists of several stages. The first stage is the load stage in which all CAP components (a CAP file is actually a ZIP file with several predefined components) are loaded onto the card. The user is offered a user interface which allows setting GP load parameters (these parameters are optional).

In case the CAP file is a library (i.e. it does not contain applets) the installation process of the CAP file is finished after loading.

In case the CAP file contains applets, the user is offered the possibility of instantiating applets (or creating an application from a module). For each applet the user can enter GP install parameters (these parameters are optional).

6.6.2.2 Creating applications from modules already loaded on the card

Not supported by the GUI

6.6.2.3 Removing applications/load files

Users can remove load files (together with all applications) or individual applications

6.7 XMEGA

6.7.1 XMEGA A3 training board

6.7.1.1 Communication overview

The training board is setup to be powered over USB when delivered. When connecting the training board to your computer a USB Serial Port will show up over which the communication will take place.

The communication protocol Inspector uses is called EmbeddedProtocol which is based on Tag-Length-Value (TLV) components. Its implementation details are described in the Inspector Manual. The code on the training board also implements this protocol.

6.7.1.2 Firmware

In order to modify the default firmware, ATMEL studio is required. This development environment can be obtained from the Atmel website [http://www.atmel.com/microsite/atmel_studio6/]. Source code of the default application can be found in `<inspector_installation_folder>/doc/manual/samples/TrainingTarget/code/`.

Building

A project solution called `EmbeddedTestApp.atsln` is present in the previously mentioned folder. This file can be opened by ATMEL studio which then allows you to quickly modify and build the target code.

Building the code can be done by pressing F7 on your keyboard or by navigation via the menu to Build -> Build Solution. The output files are in the <source_root_folder>/default/ folder and can be programmed on the target as described in the Bootloader section.

Source tree layout

When looking in the root folder you will see the following folders: acquisition, include and target. Their functionality is described below:

- acquisition - contains code which is generic to all target boards. E.g. A software AES implementation, utility functions for parsing communication etc.
- include - contains the header files for all code in the tree.
- target - contains code which is specific for the target board. E.g. the UART driver, code to control the hardware encryption engines etc.

Looking further in the root folder we find two source files. One is called testapp.c, which contains the main receive & response loop. The second file is called EmbeddedTestApp.c, which contains the low level initialization routines for the target board such as for the clock, UART and GPIO.

6.7.1.3 Bootloader

In this section, we explain how to program the XMEGA A3 training board with custom firmware. Using Atmel Studio, two files can be produced:

- <Project>.hex (HEX file of flash contents)
- <Project>.eep (HEX file of EEPROM contents)

The training platform is programmed with a bootloader during production. This means that it can be programmed in the field using a bootloader application, without the need for a programmer.

Figure 6.60. Jumper positions for application (left, default) and bootloader mode (right)



The bootloader checks the state of bit 0 of port E (i.e. PE0) when the board is booting. If the input is high (i.e. connected to VCC), it jumps to the programmed application code. If the input is low (i.e. connected to GND) the bootloader will wait for programming commands.

When starting the bootloader.exe program (which can be found in <inspector_installation_folder>/doc/manual/samples/TrainingTarget/) without command line switches, it will pop up two file open dialogs asking for the flash HEX file and EEPROM HEX file respectively. After two files are selected, the actual programming sequence begins. The output should be similar to the following:

```
C:\.....>bootloader.exe
Riscure XMEGA bootloader 1.0

Entering programming mode...
Signature matches device!
Erasing chip contents...
Reading HEX input file for flash operations...
#####
#####
#####
Programming Flash contents...
Using block mode...
#####
#####
#####
Reading Flash contents...
Using block mode...
#####
#####
#####
Comparing Flash data...
Equal!
Reading HEX input file for EEPROM operations...
#####
Programming EEPROM contents...
Using block mode...
#
Reading EEPROM contents...
Using block mode...
#
Comparing EEPROM data...
Equal!
Leaving programming mode...

C:\.....>
```

After the programming succeeded, make sure to place back the jumper to select the application mode.

For more bootloader options, please run the bootloader.exe program using the -? command line switch. This application is also described in Atmel application note AVR911 [<http://www.atmel.com/Images/doc2568.pdf>].

For more information about the installed bootloader, please refer to Atmel application note AVR1605 [<http://www.atmel.com/Images/doc8242.pdf>]. The tools referred to in this document should be compatible with the bootloader programmed on the training boards.

7 Real-Time pattern matching

This section details the way of doing pattern matching with the icWaves to, for example, improve the triggering during an analysis.



Warning

The current description of acquiring traceset is deprecated, please find relevant functions from Acquisition2 modules and Section 6.5.1, “icWaves”

7.1 Introduction

Clock jitter and random program interrupts can make Side Channel Analysis and Fault Injection very difficult. The lack of accurate timing in a perturbation attack or misalignment of the measurements in a DPA attack can complicate attack execution.

Figure 7.1. Misalignment in side channel analysis

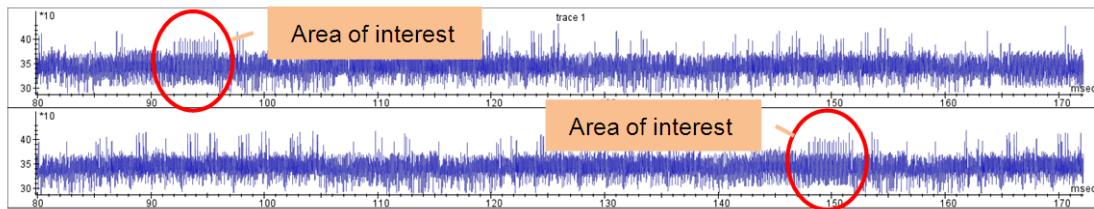


Figure 7.1, “Misalignment in side channel analysis” shows a side channel analysis example. The temporal offset for the target area of interest for two consecutive traces varies 55 ms, while the target area itself is small. With a fixed time delay trigger pulse for an oscilloscope this would result in traces that are either rather long, or have a rather low time resolution. This results in a suboptimal attack performance and quality.

For fault injection the situation is even worse: often a fault must be injected a very precise moment in the code processing. With unpredictable delays this results in tedious experimental repetitions in order to get a lucky shot.

For these situations, a fixed time-delay trigger pulse is not suitable. It is much better when a pattern in the signal is detected just before the point a measurement should start or a fault should be injected. This can be achieved with the icWaves. This device generates trigger pulses after real-time detection of one or two distinctive patterns in the signal. In addition, the device has a special narrow band-pass filter built in to enable the detection of such a pattern, even in very noisy signals. The latter is important because side channel signals can be very noisy and detecting a pre-defined pattern is therefore not always feasible, at least, not without a tunable filtering mechanism.

The built-in filter is an optional component in the measurement chain. If the measured signals can be captured with sufficient detail, it is recommended not to use the built-in filter. In that case, a simple external low pass filter satisfies.

7.2 Software

The icWaves software is seamlessly integrated with the Inspector Platform. The software consists of a device driver included in the Inspector Platform core, plus the *icWavesConfiguration* Inspector module for managing and configuring the device.

The icWaves can be used as an oscilloscope in acquisition modules to acquire a reference trace as it will be seen by icWaves. After acquiring the reference trace, a reference pattern can be selected and loaded into icWaves using the *icWavesConfiguration* module.

In this module, a threshold for the SAD (Sum of Absolute Differences) value must be selected. SAD values below this threshold will result in a trigger pulse generated on the *Trigger out* connector of icWaves.

The *icWavesConfiguration* module can also be used to simulate the SAD computation performed in hardware by the icWaves. This supports choosing a proper threshold value to be used for differentiating the right pattern. In addition to the module, Inspector provides an icWaves configuration menu. It will open a configuration window which can be used to tune the icWaves threshold and frequency settings during an acquisition.

See the tutorials provided by Riscure together with icWaves for more information on the software, including detailed explanations of all its parameters as well as hands on exercises.

7.3 Detecting patterns in raw signals

Using the icWaves can be broken-up into the following steps:

1. Determine area of interest
2. Acquire reference traces
3. Filtering (optional)
4. Select a suitable pattern
5. Load the reference pattern and arm icWaves
6. Determine the trigger threshold
7. Use the icWaves trigger output

The filtering step is optional, and should be performed in the event that no suitable pattern is found in the raw signal. This step is further explained in the next section. In this section we concentrate on the rest of the steps.

When the built-in filter is used there is no need for any external filtering. The output from the signal source (e.g. Power Tracer, or EM probe), can be connected directly to the filter input of icWaves, and the output of the built-in filter can be directly coupled into the icWaves signal input. The filter input is sensitive. The maximum signal within the filter frequency band is $250\text{mV}_{\text{p-p}}$. If the signal exceeds this maximum, the 20 dB attenuator should be used. The 20 dB attenuator is a BNC-BNC in-line component that is delivered as a separate component with the icWaves. The 20 dB attenuator reduces the signal amplitude by a factor 10.

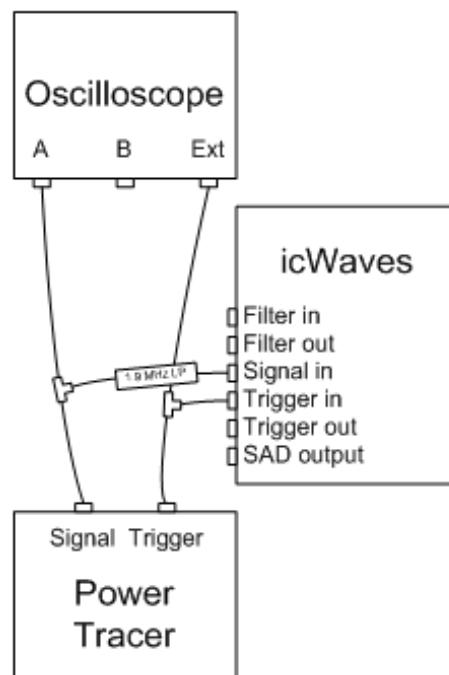
When the built-in filter is not used, the signal cable should be connected to the signal input of icWaves. The signal input is connected to an analog-to-digital converter with a fixed sample frequency of 100 MHz. When a lower sample frequency is selected (e.g. 1 MHz), all samples within the corresponding sample period (100 samples during 1 μ s for 1 MHz) are averaged to one sample. In this way undesired anti-aliasing effects are minimized. Samples with a maximum sample frequency for the signal input is 100 MHz. Another way to reduce anti-aliasing effect is to use a low pass filter between signal source an icWaves. The icWaves is delivered with a BNC in-line low pass filter with a cut-off frequency of 1.9 MHz.

For the first step, the user just configures the side channel measurement setup (Power Tracer, EM Probe Station, RF Tracer,...) and acquires raw traces. From there, an interesting region must be determined. This usually implies selecting a region of the trace where the analyst believes the interesting operations can be found.

After this step, icWaves must be inserted into the measurement setup, as shown in Figure 7.2, “Acquiring a reference signal with icWaves”. Optional (only when built-in filter not used) a 1.9 MHz analog low pass filter is inserted at the Signal in reduce high frequency noise.

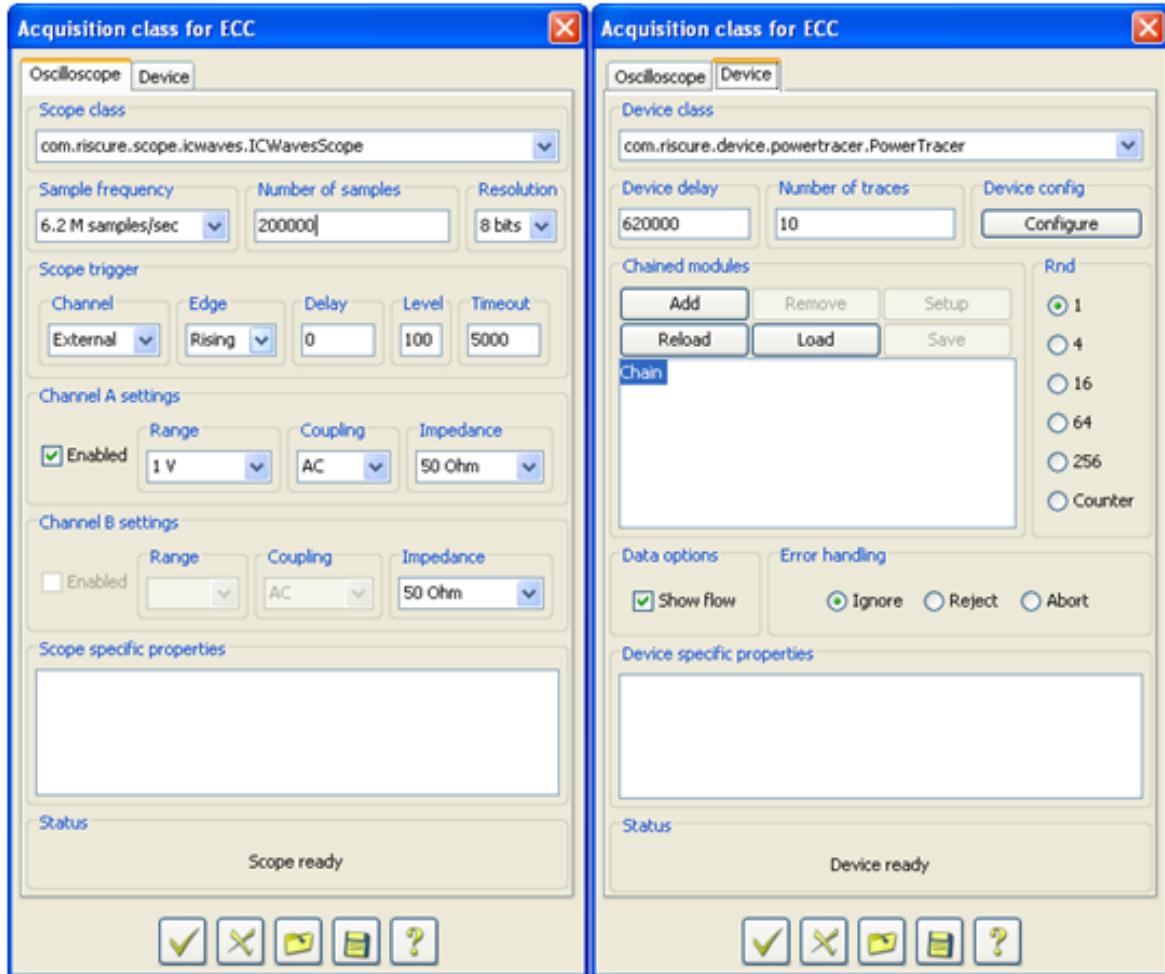
The side channel signal must be connected to this input port, as well as the oscilloscope. The trigger output from the measuring device (Power Tracer, RF Tracer, VC Glitcher ...) must be connected to the Trigger in of icWaves, as well as to the trigger input of the oscilloscope.

Figure 7.2. Acquiring a reference signal with icWaves



With this setup, icWaves is ready for acquiring a reference trace. This is done using an Inspector Acquisition module, such as *DesAcquisition*, and selecting the icWavesScope device in the scope panel.

The acquisition module must be configured in such a way that the interesting region falls into the acquired trace, so that it can be uploaded to icWaves as a reference signal. See Figure 7.3, “Acquiring a reference trace”. If the traces are significantly misaligned, more traces can be acquired until the desired reference signal is visible in one of the traces.

Figure 7.3. Acquiring a reference trace

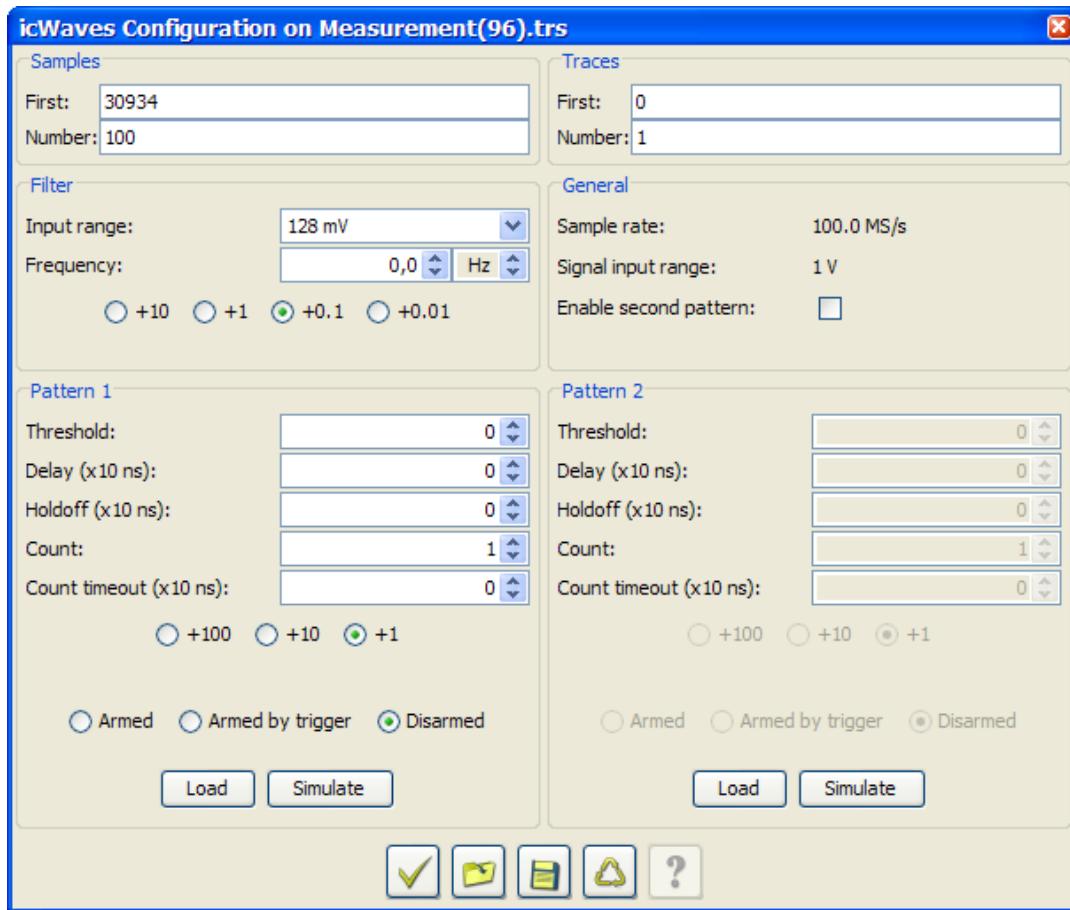
Then, a pattern must be selected from the measured trace. This is the pattern that will be uploaded to icWaves, and can contain at most 256 samples when using a single pattern or 128 samples when using two patterns. For obtaining a measure of the quality of the selected pattern, the *icWavesConfiguration* module can be used to simulate the SAD trace (see Figure 7.4, “icWaves configuration module”).

With noisy reference traces it is also possible to use several aligned traces and create an average trace. Then a part of the average trace could be uploaded to icWaves as a reference pattern.

If the result of the SAD computation does not show negative peaks in the desired trigger instance, an analog filtering step could be performed on the input traces aiming to obtain distinctive patterns. See the next section for information on the filtering stage.

Assuming a suitable pattern has been found, it must be sent to icWaves. To that end, we use the *icWavesConfiguration* module as shown in Figure 7.4, “icWaves configuration module”.

When the module is launched with an input trace, it is possible to load the selected pattern as a reference pattern to icWaves. This can be done using the *Load* button for one of the patterns.

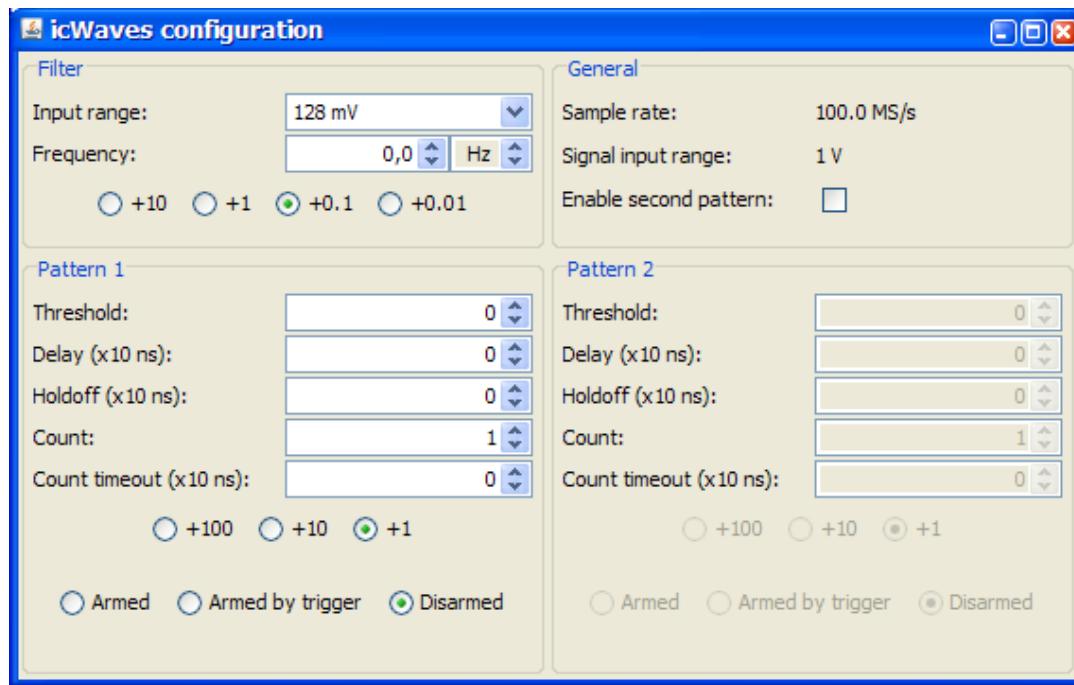
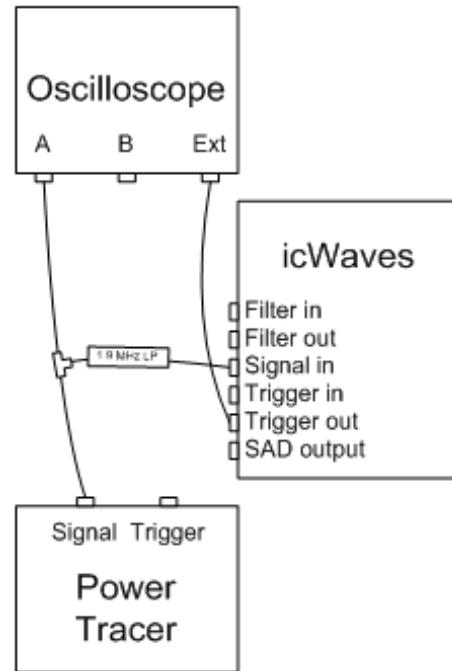
Figure 7.4. icWaves configuration module

After pressing the *Armed* option button, icWaves will be armed and it will compare input signals with the loaded patterns and produce a trigger pulse when it is below the specified threshold. However, the threshold still needs to be determined.

To that end, the *Simulate* button can be used to produce an SAD trace set. A threshold value can normally be determined by looking at the generated signals and selecting a value higher than the SAD peaks in the desired trigger point, but lower than the noise level. For a concrete example, please see the tutorials provided by Riscure. Besides the threshold, icWaves features some other more advanced configuration options, which are described in the section called “Triggering”.

After determining the proper threshold, it can be set using the *icWavesConfiguration* module. Afterwards, a new acquisition can be performed, using icWaves for triggering the oscilloscope based on the selected pattern.

If the trigger generated by icWaves occurs before or after the desired pattern appears in the input signal, it is likely that the threshold value is not correct or the selected pattern is not distinctive enough. During an acquisition, it is possible to tune the threshold value by opening the icWaves Configuration dialog from the *Tools > icWaves Configuration...* menu:

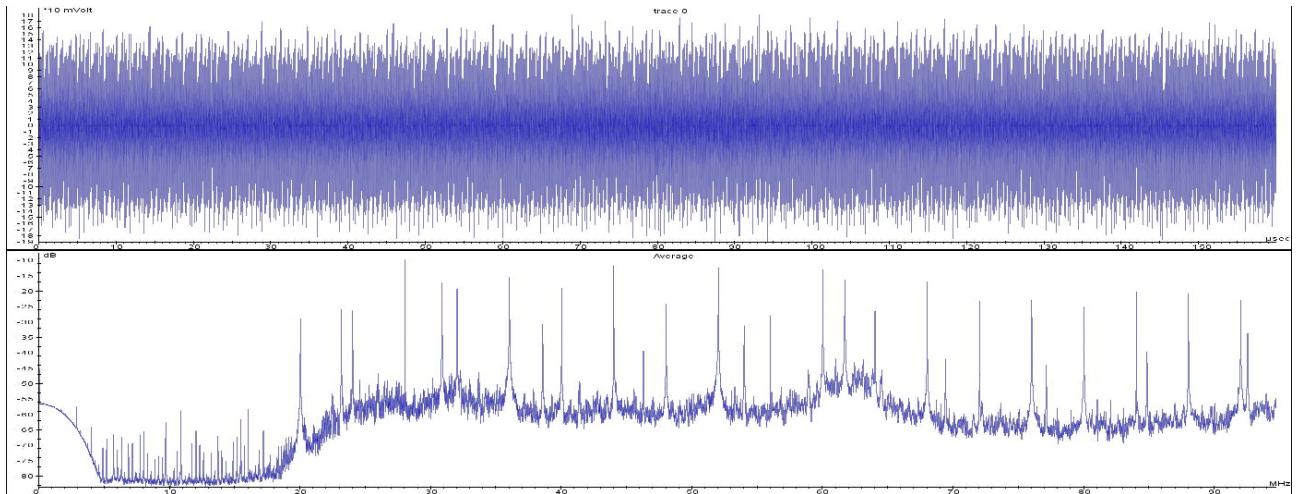
Figure 7.5. icWaves configuration window**Figure 7.6. Triggering on Trigger out**

7.4 Reducing noise with the filter

When the side channel signal is very noisy, detecting patterns in the raw signal as explained in the previous section can turn out to be very difficult. To help to detect patterns in these cases, the built-in tunable filter in icWaves can be used.

A very noisy signal will have frequency content over a very wide spectrum, and taking only part of these frequency components can show interesting patterns. Figure 7.7, “Example power trace and average spectrum” shows a noisy trace together with an average spectrum of 25 traces.

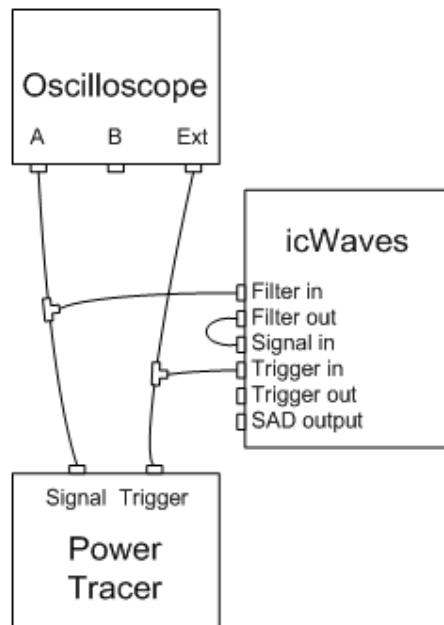
Figure 7.7. Example power trace and average spectrum



In this case, the frequency range around 30.8 MHz seems to contain interesting information. One would like to look only at that range of the signal for identifying interesting patterns.

To that end, the signal input is connected first to the icWaves Filter in port, instead of connecting it to the Signal in (Figure 7.8, “icWaves using tunable filter”).

Figure 7.8. icWaves using tunable filter

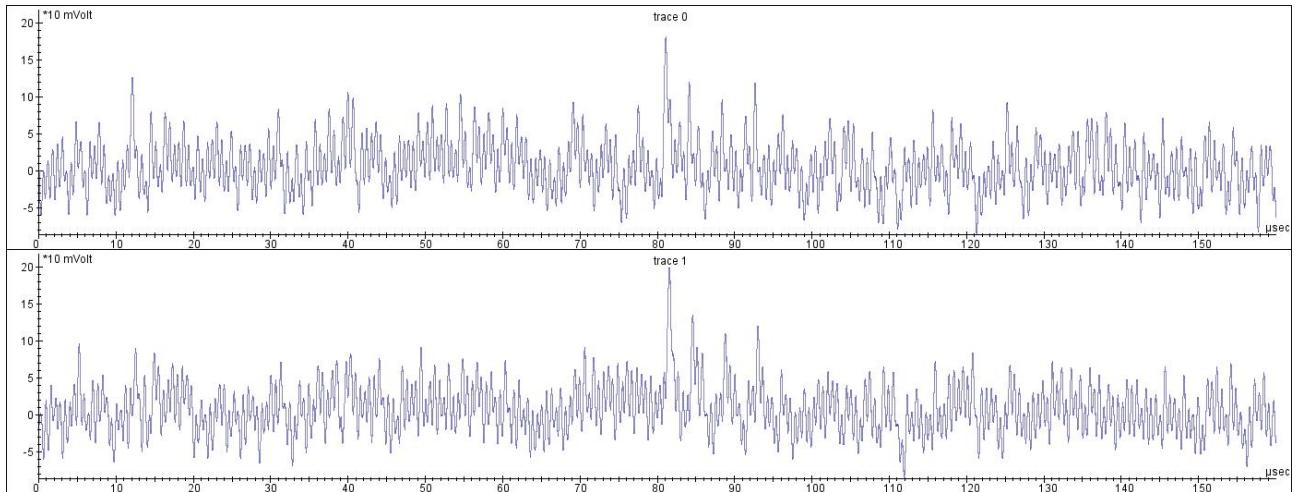


The tuning frequency needs to be indicated to icWaves by means of the *icWavesConfiguration* module from Inspector. It is even possible to tune the frequency during an acquisition by

means of the *Tools > icWaves Configuration...* menu. Note that due to the design of the filter stage (see Figure 6.42, “Mixer design”) the selected frequency should be within a 1 MHz distance of the target frequency.

With the setup shown in Figure 7.8, “icWaves using tunable filter” and a filter frequency of 30.0 MHz, we can recognize peaks in the signal where the DES operation occurs. Figure 7.9, “Filtered traces” shows the resulting trace, with a peak around 80us. See the tutorial on the use of the filter for a more detailed example.

Figure 7.9. Filtered traces



8 Software Development for Inspector

Inspector is an open platform and can be extended with new functionality at any level. These extensions need to be developed in the Java language and link to Inspector APIs.

Starting from version 4.4, significant changes are made in shifting the internal software architecture of Inspector towards an OSGi model. These changes add new programming interfaces and methods to Inspector. These changes do not modify the existing APIs, but the implementation of these APIs may be different from earlier versions of Inspector. Any module developed for earlier versions of Inspector should still compile and run without problems in this (and future) versions.

8.1 Overview

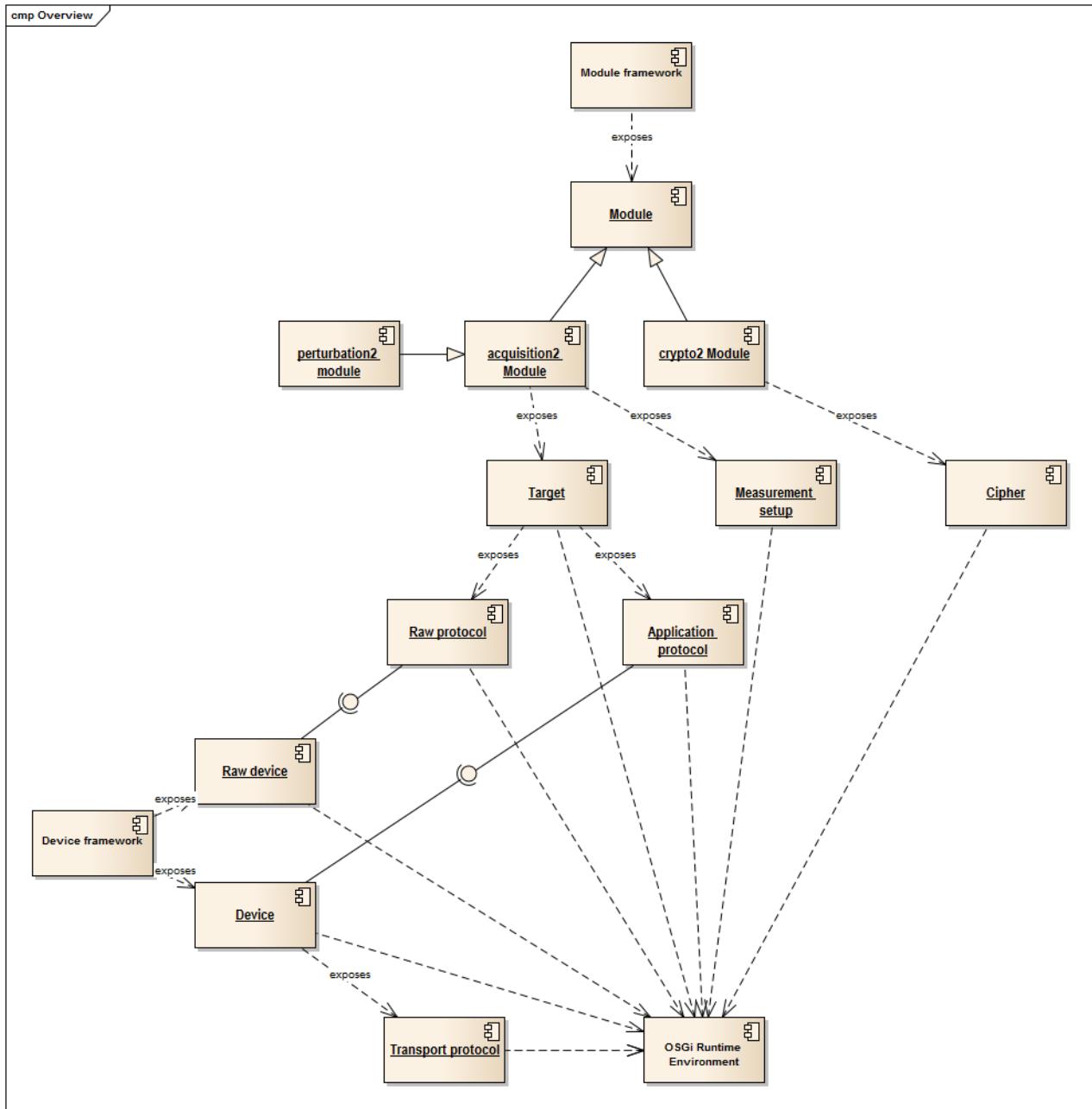
Inspector was designed with the idea of extendibility by users in mind. The *module* was introduced as a building block in which the most common workflows, such as data acquisition, filtering and analysis could be implemented easily. The module is seen as a *plug-in* because a user can create a new or adapt an existing module and load and run the newly created module in Inspector.

Over time it became clear that most modules have commonalities which could not be easily abstracted or hidden within the module framework. The result of this is that a majority of Inspector's modules are copies of one another, with minor differences to deal with specific circumstances. An example of this are the DesKnownKeyAnalysis and AesKnownKeyAnalysis modules, which in principle only differ in the specifics of the cipher that is being analysed by the module.

In order to address the issue mentioned above, Inspector introduced new plug-ins from version 4.3 onwards, which more accurately represent common Inspector workflow concepts: such as ciphers, application protocols, measurement setups and more. To continue with the earlier example, the result of this introduction of new plug-ins is that when known-key analysis functionality of a cipher that is not yet known by Inspector is desired, all that is needed is to implement the new cipher; no new module needs to be created. An added benefit is that when the new cipher is implemented, other functionality for that cipher also becomes available; such as first-order analysis, the calculator and more.

Figure 8.1, “Inspector plug-in overview” gives an overview of the most important plug-ins that are available in Inspector (plug-ins are underlined). It can be seen from this figure that plug-ins themselves can be a plug-in framework for another type of plug-in. An example of this is the target plug-in, which defines the application and raw protocol plug-ins. Dependencies are also marked in the figure. It can be seen for example that application protocols depend on devices.

Figure 8.1. Inspector plug-in overview



This chapter provides architectural, design and wherever applicable implementation details of Inspector's plug-ins. Section 8.3, “Developing Classic Modules” describes modules, the traditional way for introducing new functions in Inspector and how to use the Eclipse IDE for module development. Subsequent sections describe the other plug-ins shown in Figure 8.1, “Inspector plug-in overview”.

Note



Chapter 9, *The Developer How-Do-I* covers the development of extensions on the source code level. It for example describes how to use the Automatic GUI

Generator (AGG), and the development of custom acquisition2/perturbation2 device drivers and protocols.

8.2 Module Wizard

Inspector comes with a *Design Wizard*, which enables the user to create boilerplate code for new software components. For historical reasons this manual generally refers to this component as the *Module Wizard*. Note however, that the repertoire of the wizard has been extended through the years and now not only creates (acquisition or perturbation) modules, but also cryptographic ciphers, communication protocols, VC Glitcher control programs, trace sort algorithms and other custom modules.

The wizard can be found under the "File" dropdown menu. It will show a list of options. The options diverge considerably, and as a result are discussed in more detail in scattered sections across this manual document.

- *Acquisition module*: This refers to creating an acquisition1 module. In Section 8.3.1, "Module development" a detailed description is given for creating such a "classic" module, both by adapting an existing module and by using the wizard. Creating an Acquisition2 module is currently not possible with the wizard. For this we suggest the method of opening a system module (under the "File" menu), saving it (by clicking the compile button) with a new name, and then extend the class file. Further reading about the acquisition2 and perturbation2 modules can be found in Chapter 9, *The Developer How-Do-I*.
- *Perturbation module*: Like above, this refers to creating a perturbation1 module. Follow the references given in the above bullet point for more information.
- *Cipher*: The Crypto2 framework allows the user to specify custom ciphers. More information about working with the Crypto2 building blocks (like ciphers, intermediates, and leakages) can be found in Section 8.3.4, "Developing custom cryptographic ciphers".
- *Protocol*: The Acquisition2/Perturbation2 framework separates the protocol from the module. This means that the user can specify his own custom protocol and use that in any existing acq2/pert2 module. More information is available in Section 9.8, "How do I implement an acquisition2/perturbation2 application protocol?".
- *Glitch Snippet*: The user can write his own glitch control code for the VC Glitcher (in transparent mode). Such a custom glitch snippet allows for glitch behavior that exceeds the standard wait/glitch sequences offered by the default glitch program. See Section E.12.3, "SC Perturbation with Glitch Program" for more information on how to create and use the snippets.
- *Perturbation Program*: The VC Glitcher can also be programmed directly (instead of via the module). This allows even more fine-grained behavior for expert users. See Section E.12.4, "Perturbation Advanced Program".
- *TraceProcessor*: The Section E.13.1, "TraceSort" can rearrange trace sets according to a certain criterion. The TraceProcessor wizard will provide boilerplate code that the user then can adjust to his wishes.
- *Other module*: Apart from an acquisition or perturbation module, it is also possible to create a (lightweight) module for data processing. The wizard proposes three options for the sort of module that you want to create. A *producer* module creates new trace sets without any

input; A *filter* module typically generates one trace for each trace of an input trace set; And a *analyzer* module first analyzes a number of input traces and then generates traces.

- *Class*: This option simply creates an empty java class file, which can be used as auxiliary data structure. Note that when this class is compiled it will give an "error" popup saying that the target is neither a module nor a service. This is a known issue; Please ignore this message.

8.3 Developing Classic Modules

Inspector allows advanced users to write their own modules using the Java programming language, for instance to support a new analysis technique. This section explains how to write function modules and specifies the API and development options that are available to the user.

A list of all public classes is available in the Inspector API documentation [[..../javadoc/index.html](#)]. In addition, the complete Java API documentation [[..../..../jdk/docs/api/index.html](#)] is available offline.

8.3.1 Module development

One of the most interesting features of Inspector is the ability to develop new library modules and apply them on the fly within Inspector. This feature permits a very rapid development cycle of new analysis techniques. Via Inspector, the user can directly open existing Java source files and generate new function modules.

How to edit, compile and load a function module? A source code module is opened by pressing the 'Open System Module...' in the File menu and selecting a source code file (*.java) in one of the folders. Next, the source can be edited in the open window. To test your changes,

you press the compile button . Inspector will ask for the filename and save it in the user

 space. This can be followed by the run button , to open the module dialog. Note that the compile button not only compiles your new source code, but also automatically loads the new function module into Run menu, user sub menu of Inspector, ready for execution on a selected trace set. When closing a source code file that you edited, you are asked to save your changes.

Inspector offers quite a few additional features while editing a Java module source file:

- Undo / Redo last actions: available as Ctrl-z and Ctrl-y
- Find text: available as Ctrl-f, find again is available as F3
- Replace text: available as Ctrl-r
- Compile: available as F9. Compiler errors are reported as hypertext and can be clicked for quick code finding
- Save: available as Ctrl-s
- Block indent / de-indent for changing the indent of a selected block of lines: available as Ctrl-t and Ctrl-d

Riscure has developed a base class called *Module* [[..javadoc/com/riscure/signalanalysis/Module.html](#)] that implements Inspector's API. Inspector links the methods in this class after compilation to its internal code and invokes them when the module is run. Users should extend the *Module* class to implement their own analysis functions. Further, users can use the *Trace* [[..javadoc/com/riscure/signalanalysis/Trace.html](#)] class to access traces contained in a trace set and the *ProgressInterface* [[..javadoc/com/riscure/signalanalysis/ProgressInterface.html](#)] to report the progress during and after function execution to Inspector.

Module developers can use full advantage of the powerful Java API. This allows a developer to create functions to a high degree of complexity. To keep module management simple it is preferable to implement only one class per module. However, for very complex modules, or modules that want to share functionality, it may be preferable to define additional classes in a library package which can be stored as a jar file in the lib folder under the Inspector home. Inspector automatically adds jar files in the lib folder to the class path. Jar files stored outside the lib folder can be explicitly added to the class path in the settings form.

When a class that extends Module is successfully compiled, it will be loaded in Inspector (as indicated in the title bar) and can be invoked. Compiled and Loaded modules are included in



the module menu and can be invoked via the module menu button or from the "Run"->"User" menu. Modules are sorted by name and can be further grouped by putting the source and class file in a subdirectory. Each subdirectory becomes a sub menu. Modules can set the value of String moduleDescription which appears as a menu item hint. If you do **not** want your module to appear on the module menu you need to implement *ModuleNotOnMenu* [[..javadoc/com/riscure/signalanalysis/Module.html](#)] marker interface.

Modules should be written according to the JDK 1.6 specification. Warnings may result if an older specification is used. As a side note: Inspector 4.5 runs on Java 7, but only in "1.6 compatibility mode". All user modules are compiled with the options "-source 1.6 -target 1.6".

How to use the development wizard? Inspector has a wizard to create skeleton code for various types of functions. The wizard is invoked by pressing the menu item "New Module Wizard" in the "File" menu..

The wizard guides the user through a series of questions to prepare a suitable starting point for further development. The wizard is controlled by a file called *module.template* in the *modules* folder. Advanced users can modify this self-explaining template file to optimize the wizard and adapt it to their requirements.

Here is some explanation of the questions that the wizard may ask:

1. Do you want to design a module or another class? Here you can choose to create a new module, or just any class
2. Do you want the module to present a confirmation dialog? If you want the module to start immediately choose 'No', and choose 'Yes' if you want to be able to select the scope or set specific parameters.
3. Do you want to customize the dialog? Choose 'No' if there are no additional parameters to be set for this module, otherwise choose 'Yes'.
4. Do you want the module to handle input traces? Choose 'No' if you intend to create a generator that produces traces without processing input traces, otherwise choose 'Yes'.

5. How many output traces will your module produce? Choose 'One per input trace' if you create a filter module. Choose 'None or several per input trace' if you create a generator (no input processed) or analysis module (multiple input traces analyzed before producing an eventual result trace).
6. Will your module need significant memory resources? Choose 'Yes' if your module will allocate large objects and may exceed the total available memory, otherwise choose 'No'.

New modules extend the *Module* [[..../avadoc/com/riscure/signalanalysis/Module.html](#)] class, which contains the basic module functionality. An overview of the commonly used module methods is given in the Module Methods section.

The third exercise in the tutorials document provides the developer with a jump start on development of new function modules. Developers may further appreciate the source code included with the standard function modules.

Module developers may sometimes be puzzled by unexpected module behavior. Check the support website [<http://www.riscure.com/tools/inspector/support>] for more info to quickly resolve basic problems. If your question is not answered address your problem to: inspectorsupport@riscure.com [<mailto:inspector@riscure.com>]

8.3.2 Module compatibility

The Module interface has been quite stable over previous versions of Inspector. However the following incompatibilities and solutions should be noted:

Changes in Version 4.0

- Most methods in Module now throw the checked exception DataOperationException. The Module Wizard has been changed to create the correct signatures for all generated methods. In your own code you do not have to throw DataOperationException, unless you are calling a method that does throw DataOperationException. The latter is for instance the case if you call super.init(...) from your code. The solution is to simply add "throws DataOperationException" to the signature of your method.
- The Trace.sample variable has been deprecated. You should now use the following methods to access this variable:
 - `getSample()`
 - `setSample()`
 - `getSampleData()`if you ONLY use those methods and no longer access `Trace.sample` directly from your module you should implement the marker interface Inspector400 to let Inspector optimize data access.
- The classloader has changed in Inspector and in Chain. You should ONLY copy the modules and classes from 3.x that you have changed or written yourself to the User area of 4.0. If you copy all classes you are likely to get some strange error messages as all classes will be found in the User area, which may not all be up to date with 4.0.
- The Chain class has been changed quite a bit. Be especially careful if you used your own Chain class in 3.x. It likely needs some changes.

8.3.3 Module Methods

The most important methods defined in the *Module* [..]/javadoc/com/riscure/analysis/*Module.html*] class that may be overloaded in sub classes are:

- `initModule()` set global variables like the moduleTitle and moduleDescription
- `initDialog()` extend the confirmation dialog with task specific components
- `setDialogValues()` set the values of visible components in the confirmation dialog, e.g. text fields or check boxes.
- `getDialogValues()` get the user selected options from the confirmation dialog
- `getRequiredMemorySize()` return the amount of memory (in bytes) a module needs. This is useful to avoid a system crash when the amount of memory the module requires may exceed the total available memory.
- `initProcess()` create and initialize data needed during a module process
- `analyze()` analyze the samples and data included in one trace
- `generate()` generate a result trace
- `process()` shortcut method that combines analysis and generate methods to produce one result trace for each input trace

Inspector handles a fixed protocol for each module. This protocol knows three stages:

1. Module loading during this stage the methods `initModule()` and `initDialog()` are invoked.
2. Process preparation during this stage the dialog is displayed and the process is parameterized with user options. The methods `setDialogValues()`, `getDialogValues()`, `getRequiredMemorySize()` and `initProcess()` are invoked sequentially.
3. Trace processing during this stage each trace is processed by invoking either `process()` or `analyze()`, optionally followed by `generate()`.

Global data

The module class defines quite a few data elements used during processing. The most important ones are:

- `boolean aborted`: whether the user tried to abort this module process
- `boolean inputRequired`: whether this process requires input, set this boolean to false for generator modules
- `boolean showDialog`: whether this process needs to show a dialog
- `String moduleTitle`: Module title displayed in the dialog title bar

- `String prefix`: prefix of resulting trace set name
- `String moduleDescription`: description of module, used a menu item hint in module menu
- `String helpFile`: path to the html file that explains the module
- `int firstSampleIndex`: firstSample selected in confirmation dialog
- `int numberOfSamples`: numberOfSamples selected in confirmation dialog
- `int firstTraceIndex`: first trace to be analyzed / processed
- `int currentTraceIndex`: index of trace to be analyzed / processed
- `int lastTraceIndex`: index of last trace to be analyzed / processed
- `int numberofTraces`: number of traces to be analyzed / processed
- `int numberofResultTraces`: number of available traces, increasing this variable will cause the generate method to be invoked
- `int sampleCoding`: coding of resulting samples
- `int newWindow`: whether a new window is to be opened for the result traces
- `boolean logScale`: whether the resulting traces are to be represented in a logarithmic scale
- `float sampleFrequency`: sample frequency of result traces
- `float xScale`: inverse of sampleFrequency
- `float yScale`: y-scale of result traces
- `ProgressInterface pi`: interface to report module progress
- `Trace resultTrace`: current result trace

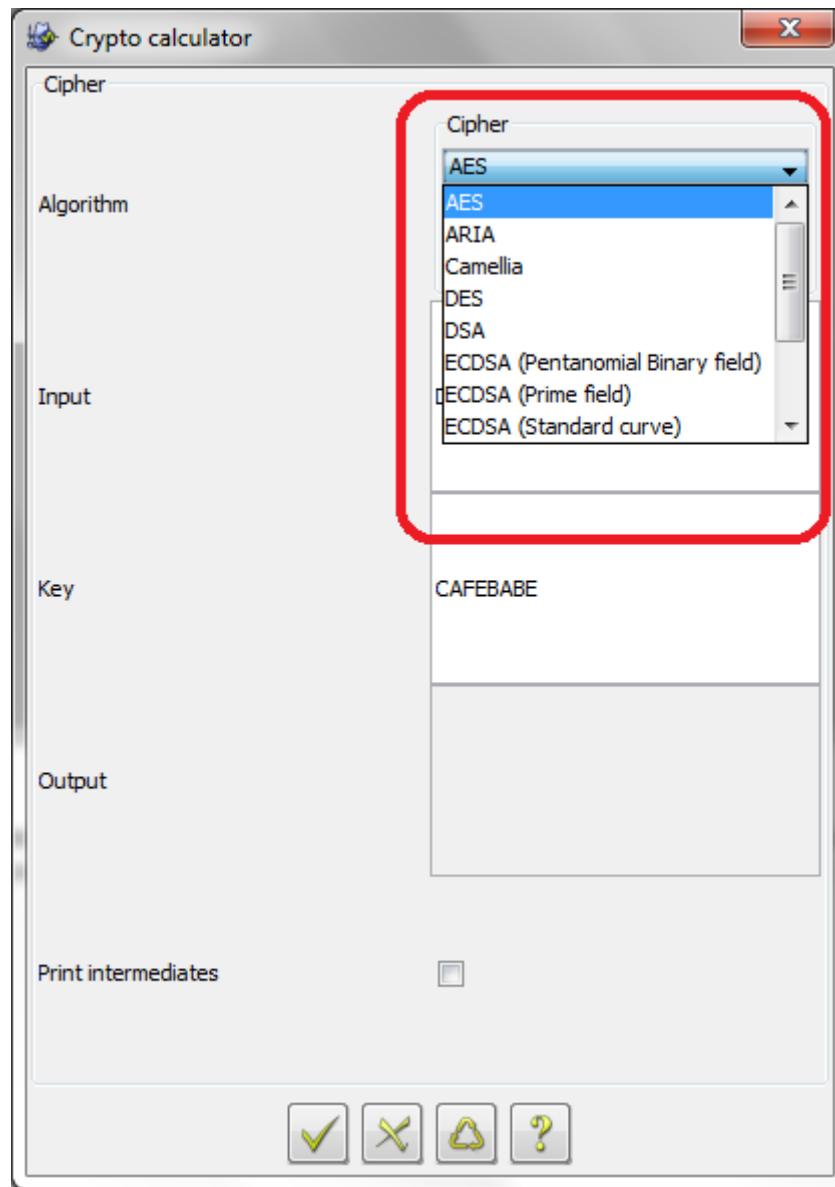
More data is defined in the documentation of the Module class [..]/javadoc/com/riscure/signalanalysis/Module.html].

8.3.4 Developing custom cryptographic ciphers

Cusom cryptographic ciphers are can be plugged into Inspector by implementing the Cipher [..]/javadoc/com/riscure/signalanalysis/crypto2/Cipher.html] interface. An implementation of Cipher defines a class of cryptographic algorithms, for example the DES, the AES, the DSA, etc., that can be used by Inspector's crypto2 framework. A Cipher instance defines an instance of a cryptographic algorithm, e.g. DES, 2TDEA/TDES2, AES-128, AES-256, etc.

All classes implementing the Cipher interface result in a library of supported ciphers in Inspector which is shared by all crypto2 modules. Figure 8.2, "Cipher list in the Calculator module." shows the ciphers listed in the Calculator module as an example.

Figure 8.2. Cipher list in the Calculator module.



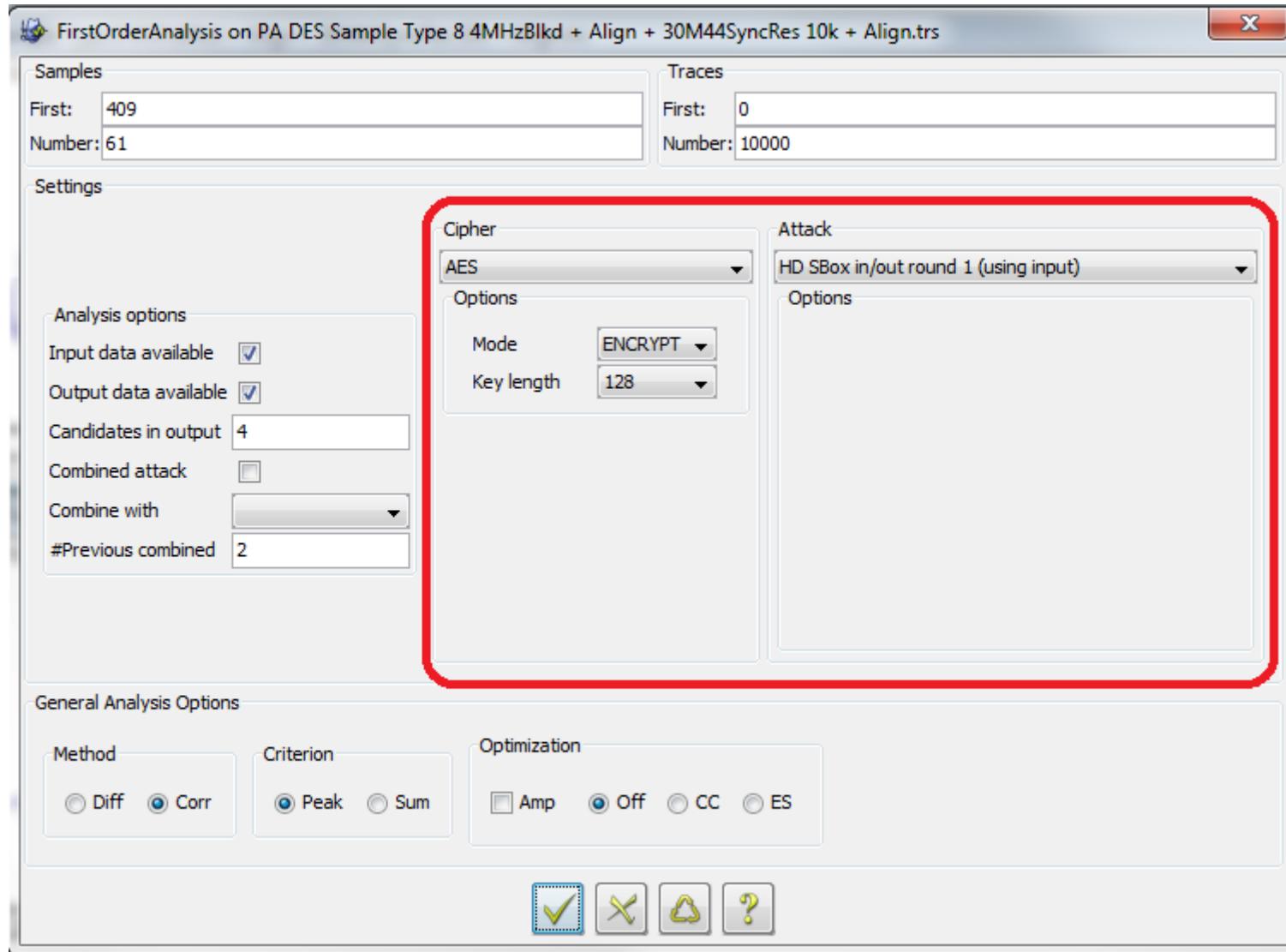
Besides being used as a crypto calculator, Cipher is used both by differential analysis modules (e.g. the FirstOrderAnalysis module) to perform differential leakage analysis and by the Simulator module to generate simulated leakage traces. As such, the Cipher interface does not only specify methods for encrypting and decrypting data, but also specifies methods for obtaining hypothetical side channel leakages. Side channel leakages can be defined in Inspector by implementing the Intermediates [..javadoc/com/riscure/signalanalysis/crypto2/Intermediates.html] interface and the Leakages [..javadoc/com/riscure/signalanalysis/crypto2/Leakages.html] interface.

An Intermediates object holds all intermediate results of a cryptographi algorithm that are in particular often targeted in side channel attacks. A Leakage object describes which and how the information of the intermediate results specified in the Intermediates object are leaked through side channels.

For instance, the `Intermediates` class for DES could include the input and the output data of all the S-boxes of the first encryption round. The `DES Leakages` class could specify that the hamming weight of the output of those S-boxes or the hamming distance of the input and the output of the same S-boxes could be leaked and may be examined during a side channel attack.

The `FirstOrderAnalysis` module gives a good presentation of the usage of the `Cipher`, `Intermediates` and `Leakages` interfaces in the context of side channel analysis.

Figure 8.3. Cipher and attacks panels in FirstOrderAnalysis module UI.



To summarize, to add another cipher in the library one needs to implement three interfaces: `Cipher` [..[/javadoc/com/riscure/signalanalysis/crypto2/Cipher.html](#)], `Intermediates` [..[/javadoc/com/riscure/signalanalysis/crypto2/Intermediates.html](#)] and `Leakages` [..[/javadoc/com/riscure/signalanalysis/crypto2/Leakages.html](#)]. In the remainder of this section we will describe how to implement these interfaces using the DES cipher as an example.

8.3.4.1 Getting started

The Inspector coding standard encourages that the implementation of these interfaces happen in a single subpackage of `crypto2.cipher`. For DES we can use the package name `crypto2.cipher.des`.

The `crypto2` framework offers several abstract classes that ease the implementation of the `Cipher` interface. It is recommended to use any of these subclasses rather than implementing the interface from the bottom-up. Currently, three abstract `Cipher` classes are available - `AbstractCipher`, `AbstractBlockCipher` and `AbstractDSACipher`. The latter two are subclasses of the first, serving as parents to block ciphers and DSA-like ciphers, such as DSA and ECDSA, respectively. To implement a block cipher such as DES, we can create a new class extending the `AbstractBlockCipher` class and implementing the remaining methods. This will give us a skeleton like below.

```
package crypto2.cipher.des;

public class DESCipher extends AbstractBlockCipher {

    public DESCipher(Mode mode) {
        super(mode);
    }

    @Override
    public String getShortName() {
    }

    @Override
    public int inputBlockSize() {
    }

    @Override
    public int keySize() {
    }

    @Override
    public void setKey(BigInteger key, Intermediates intermediates) {
    }

    @Override
    public Leakages createLeakages() {
    }

    @Override
    public Intermediates createIntermediates() {
    }

    @Override
    public int rounds() {
    }

    @Override
    protected BigInteger encrypt(BigInteger plain, Intermediates intermediates) {
    }

    @Override
    protected BigInteger decrypt(BigInteger cipher, Intermediates intermediates) {
    }
}
```

}

To ease the implementation of the `Intermediates` interface, one can extend the `AbstractIntermediates` class, which results in the following skeleton.

```
package crypto2.cipher.des;

public class DESIntermediates extends AbstractIntermediates
{

    public DESIntermediates() {
    }

    @Override
    public PartialKey getPartialKey() {
    }

}
```

In a similar way, to implement the `Leakages` interface, one can subclass the `AbstractLeakages` class, which results in the following skeleton.

```
package crypto2.cipher.des;

public class DESLeakages extends AbstractLeakages {

    public DESLeakages(Cipher cipher) {
        super(cipher);
    }

    @Override
    protected void createLeakageModel(Intermediates intermediates) {
    }

}
```

8.3.4.2 Implementing the Cipher interface

In this section we only discuss the implementation of a selected set of methods in the `Cipher` interface. The other methods are trivial to implement following the JavaDoc of the `Cipher` [./javadoc/com/riscure/signalanalysis/crypto2/Cipher.html] interface.

- `DESCipher` constructs the object. The `crypto2` framework generates the user interface using the arguments in the constructor of the `Cipher` object and their annotations. To this end, special rules apply to the constructor.

The constructor of a `Cipher` must be public and contain arguments of the types `boolean`, `int`, `BigInteger` or `Enumeration`. Each argument must be annotated with the `@CipherParam` to indicate abbreviation, full name, description, min, max, default and base. See also the JavaDoc of `CipherParam` [./javadoc/com/riscure/signalanalysis/crypto2/CipherParam.html] interface.

For example:

```
public SomeCipher(
    @CipherParam(name = "Size", defaultInt = 160) int hashSize,
    @CipherParam(name = "Order", defaultString = "OUTER") Order order,
```

```
        @CipherParam(name = "Curve", base = 16, defaultString = "fffffc") BigInteger
curve
)
```

given enumeration:

```
public enum Order {
    INNER,
    OUTER
}
```

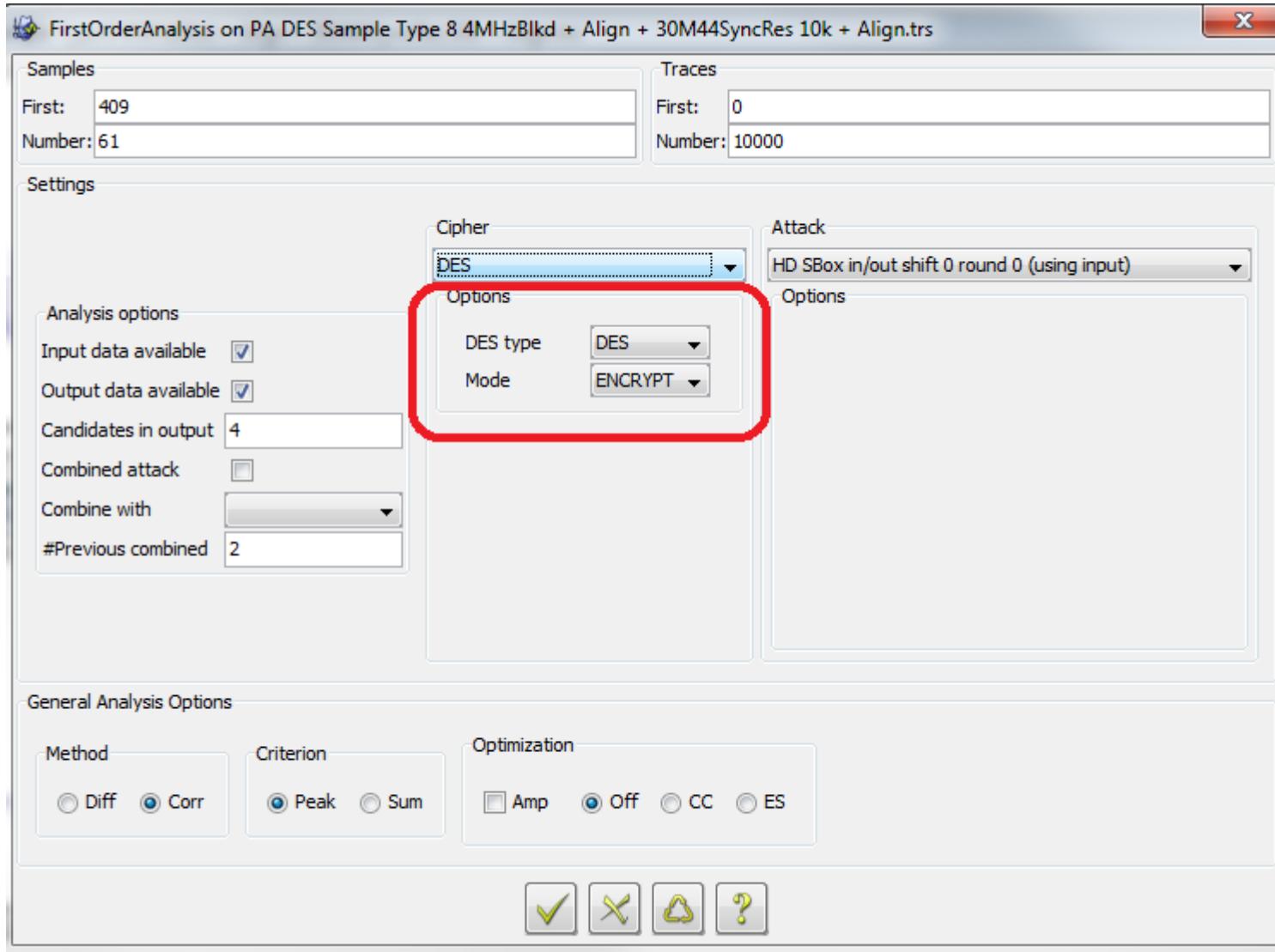
will result in a UI with an integer field called *Size* with default value 160, a drop down box called *Order* with options *OUTER* and *INNER* and with default value *OUTER*, and a hexadecimal integer field with default value FFFFFC.

For the DES Cipher, there is the argument *Mode* inherited from the *AbstractBlockCipher* class. Additionally, one may want to distinguish the type of DES operation - single DES or triple DES with one, two and three keys. To this end, we may write the DES Cipher constructor as follows.

```
public DESCipher(
    @CipherParam(name = "DES type") DESType desType,
    @CipherParam(name = "Mode") Mode mode
)
```

This will result in the "Options" list in the "Cipher" panel as shown below.

**Figure 8.4. Options panel of DES cipher
in FirstOrderAnalysis module UI.**



- `setKey` sets the internal session keys (argument `Intermediates intermediates`) with a given secret key (`BigInteger key`). It is called during a known-key scenario, such as trace simulation, known-key analysis, and key verification given encryption (or decryption) input and output. To this end, for this DES Cipher one needs to calculate the sessions keys based on the given secret key in this method and differentiate between DES and 3-DES.
- `encrypt` implements the encryption of the given plaintext (`BigInteger plain`) using the key stored in the provided `Intermediates` object. Intermediate states of the crypto algorithm are also updated during this encryption and are captured in the `Intermediates` object. For DES, one needs to distinguish between different types of DES operations.
- `decrypt` is analogous to `encrypt` but for decryption of messages.

- `createIntermediates` creates an empty `Intermediates` object for this cipher. It may be called during the initialization phase of e.g. a differential analysis or a key verification. Hence, simply:

```
public Intermediates createIntermediates() {  
    return new DESIntermediates(this);  
}
```

The `DESIntermediates` will be explained in the following session.

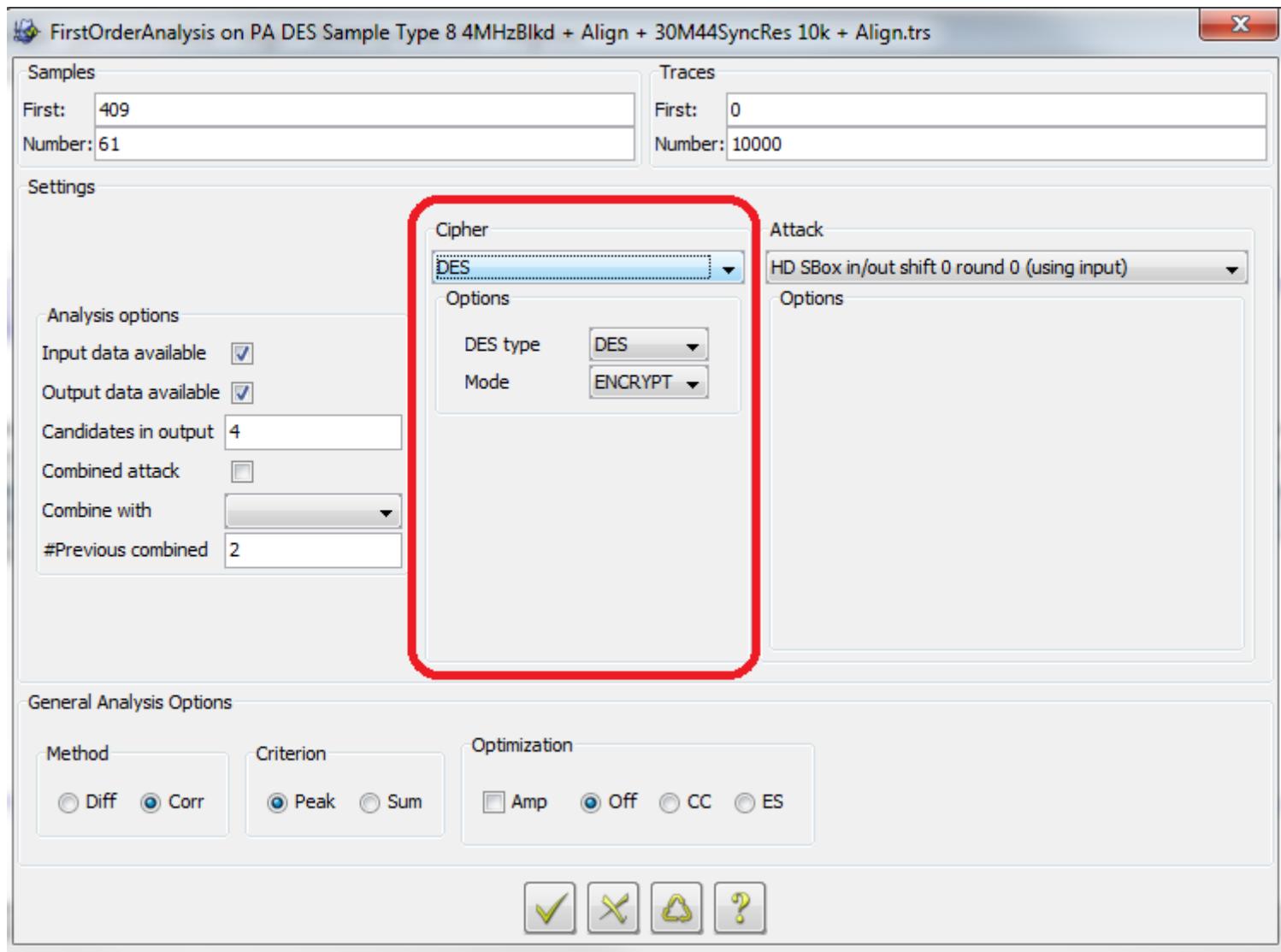
- `createLeakages` creates a `Leakages` object associated with this `Cipher`. It may be called during the initialization phase of e.g. a differential analysis, a key verification, a trace simulation or a known-key analysis. Hence, simply:

```
public Leakages createLeakages() {  
    return new DESLeakages(this);  
}
```

The `DESLeakages` will be explained in later session.

As a result of this `DESCipher` implementation, one should find "DES" in the drop-down list of the "Cipher" panel in the `FirstOrderAnalysis` module, as well as in all the other `crypto2` modules which use ciphers.

Figure 8.5. DES cipher in the FirstOrderAnalysis module.



8.3.4.3 Implementing the Intermediates interface

Objects implementing the `Intermediates` interface contains all interesting intermediate results (for side channel analysis), including the round keys (aka. session keys or sub-keys) used during a crypto operation.

- `DESIntermediates` constructs this `Intermediates` object. This is where the intermediate states of the cipher are defined and initialized. An `Intermediates` object is meant to hold the intermediate values of a crypto algorithm that are relevant to a side-channel attack, namely session keys and those that can be used for key recovery and of which physical leakages can be observed through side channels. Hence, it is only necessary to capture the interesting intermediate values that can be used for a side-channel attack rather than all the intermediates states of the algorithm.

For example, in case of DES, intermediate values commonly used in side-channel attacks are the input and output of the S-boxes, and the left and right halves of the input of every round. To this end, we create the following fields in the DESIntermediates class.

```
/** 32-bit right halve round input, +1 for last round output */
public DirectIntermediateValue[] right;

/** 32-bit left halve round input, +1 for last round output */
public DirectIntermediateValue[] left;

/** S-box outputs */
public DirectIntermediateValue[][] sboxOut;

/** Round subkeys */
public DirectIntermediateValue[][] roundKey;
```

Each array holds the specified intermediates values of all rounds. Note that for speed purposes it is advised not to use getters and setters for the fields, but to access the fields directly from the Cipher; hence the public fields.

Every intermediate state of a crypto algorithm is an object that implements the [IntermediateValue](#) [..javadoc/com/riscure/signalanalysis/crypto2/IntermediateValue.html] interface. The crypto2 framework provides several classes of `IntermediateValue`. In particular, there are `DirectIntermediateValue` that directly holds an intermediate value of an algorithm, `LinearTransformKey` which implements a session key that is a linear transformation of a number of underlying intermediate values (used for e.g. MixColumns in AES) and `CombinedIntermediateValue` which represents a combination of two intermediate states. In case of DES, `DirectIntermediateValue` shall be used.

The declared fields need to be initialized during the constructing of the object. The `AbstractIntermedites` provides method `createDirectIntermediateValue` that creates arrays of intermediate values with given properties: name, maximum value of the elements, if is a key, and dimension of the array. The `Intermedites` interface requires an implementation of the `getAll` method which returns all the intermediate values related to a specific cipher. To this end, one needs to store all created intermediate values. The `AbstractIntermedites` contains a list of `IntermediateValue` that serves for this purpose, and it provides a method `registerArray` that can be used to add elements to this list. Hence, the constructor of `DESIntermedites` may be implemented as follows.

```
protected DESIntermedites(DESCipher cipher) {
    this.cipher = cipher;
    int rounds = cipher.rounds();

    left = (DirectIntermediateValue[]) createDirectIntermediateValue(
        "left",
        new BigInteger("100000000", 16),
        false,
        rounds() + 1
    );
    right = (DirectIntermediateValue[]) createDirectIntermediateValue(
        "right",
        new BigInteger("100000000", 16),
        false,
        rounds() + 1
    );
    sboxOut = (DirectIntermediateValue[][][]) createDirectIntermediateValue(
        "sboxOut",
```

```

        new BigInteger("10", 16),
        false,
        8,
        rounds()
    );
    roundKey = (DirectIntermediateValue[][][]) createDirectIntermediateValue(
        "roundKey",
        new BigInteger("40", 16),
        true,
        8,
        rounds()
    );
}

registerArray(left);
registerArray(right);
registerArray(sboxOut);
registerArray(roundKey);
}

```

The most important methods to implement in an `Intermediates` class are `getPartialKey` and `setSubkeys`.

- `getPartialKey` returns bits of the secret key that are known up to now based on the session key information captured in this `Intermediates` object. For DES, this means to revert the key scheduling using the available session keys and returns the obtained secret DES key with possibly some bits unknown.
- `setSubkeys` sets the session keys respectively with their best candidates according to the `candidateScores` obtained after e.g. a differential analysis. The implementation in the `AbstractIntermediates` class only sets the session keys that are directly targeted in an attack, e.g. the first round key of a DES encryption. In case more session keys can be determined, additional steps can be implemented the `Intermediates` class of the individual cipher to set more session keys. For example, in case we aim to recover the 3 (or 2) DES keys in a 3-DES cipher, in the `DESIntermediates` we may want the `setSubkeys` method to set all session keys of a single DES once the first (or the last) two round keys are known, so that it is possible to attack the next DES key. For more details of the implementation, please see the source code `modules\crypto2\cipher\des\DESIntermediates.java` under your Inspector installation directory.

8.3.4.4 Implementing the Leakages interface

- `DESLeakages` constructs an object and initializes the fields specified in its super class. In addition, one may keep a reference of the given cipher for future use:

```

public DESLeakages(DESCipher desCipher) {
    super(desCipher);
    this.cipher = desCipher;
}

```

Objects implementing the `Leakages` interface must define how exactly what information of one or a set of intermediate values of an algorithm may be leaked through side channels. This is mostly described in the implementation of the `createLeakageModel` method.

- `createLeakageModel` creates leakage models based on the given intermediates, for example, the Hamming weight of each S-box output, or the Hamming distance between the inputs of two consecutive rounds.

In crypto2, every leakage model is an object that implements the `LeakageModel` [../avadoc/com/riscure/signalanalysis/crypto2/LeakageModel.html] interface. The crypto2 framework implements two classes of `LeakageModel`: `DefaultLeakageModel` which serves as a default implementation of `LeakageModel` and `CombinedAttack` which extends the former and is used for combined differential attacks. For the implementation of `DESLeakages`, `DefaultLeakageModel` can be used.

Similar to the `IntermediateValues`, all `LeakageModel` should be registered. To this end, the `createLeakageModel` method should call `addLeakageModel` in `AbstractLeakages` for all applicable models. Method `void addLeakageModel(String name, Map<IntermediateValue, PowerModel> keyPower, InputGenerator generator, DataType inout, Set<IntermediateValue> dependencies)` takes the following arguments: the name of the leakage model, a map between a key target and the power model that determines the key target, the input generator required, input or output needed, and the intermediate values needed to be set before this leakage model can be executed.

The following code serves as an example of implementation of `createLeakageModel` for `DESLeakages` when only encryption is considered. In this example, we demonstrate for only two leakage models: the Hamming weight of the S-box outputs of every round and the Hamming distance between the right halves of the inputs of every two consecutive rounds.

```
protected void createLeakageModel(Intermediates ints) {
    DESIntermediates intermediates = (DESIntermediates) ints;
    if (this.cipher.isEncrypt()) {
        for (int r = 0; r < cipher.rounds(); r++) {
            // Using input
            addLeakageModel(
                "HW SBox out round " + r,
                sboxHw(r, intermediates),
                null,
                DataType.INPUT,
                roundKeysUpTo(r - 1, intermediates)
            );
            addLeakageModel(
                "HD Left/right round " + r,
                roundXOR(r + 1, r, intermediates),
                null,
                DataType.INPUT,
                roundKeysUpTo(r - 1, intermediates)
            );

            // Using output
            addLeakageModel(
                "HW SBox out round " + r,
                sboxHw(r, intermediates),
                null,
                DataType.OUTPUT,
                roundKeysDownTo(r + 1, intermediates)
            );
            addLeakageModel(
                "HD Left/right round " + r,
                roundXOR(r, r, intermediates),
                null,
                DataType.OUTPUT,
                roundKeysDownTo(r + 1, intermediates)
            );
        }
    }
}
```

```
    );  
}
```

Method `Set<IntermediateValue> roundKeysUpTo(int round, DESIntermediates intermediates)` returns a set of round keys from the first round to the $(r-1)$ -th round (incl.) held in the `Intermediates` object. In other words, when using input session keys of rounds $[1, r-1]$ need to be known before the round key in round r can be attacked.

Method `Set<IntermediateValue> roundKeysDownTo(int round, DESIntermediates intermediates)` is analogous to `roundKeysUpTo` but is called when output data is used, in which case, session keys of rounds $[r+1, \text{this.cipher.rounds}()]$ must be known before the r -th round key can be attacked.

The maps between the key targets and the corresponding leakage models are built by the following methods.

```

/**
 * @return Attack map relating sbox hamming weight leakage to round keys
 */
private Map<IntermediateValue, PowerModel> sboxHw(
    int round,
    DESIntermediates intermediates
) {
    Map<IntermediateValue, LeakageModel> keyLeakage = new
TreeMap<IntermediateValue, LeakageModel>();
    for (int k = 0; k < 8; k++) {
        keyLeakage.put(
            intermediates.roundKey[k][round],
            new HWLeakageModel(intermediates.sboxOut[k][round])
        );
    }
    return keyLeakage;
}

/**
 * @return Attack map for relating hamming distance of round XOR leakage to round
keys
 */
private Map<IntermediateValue, LeakageModel> roundXOR(
    int leakRound,
    int keyRound,
    DESIntermediates intermediates
) {
    Map<IntermediateValue, PowerModel> keyPower = new TreeMap<IntermediateValue,
PowerModel>();
    IntermediateValue l = intermediates.right[leakRound];
    IntermediateValue r = intermediates.left[leakRound];

    for (int k = 0; k < 8; k++) {
        // Create mask for this S-box
        long mask = 0xF << (4 * (7 - k));
        mask = DESCipher.P.permute(mask);
        keyPower.put(
            intermediates.roundKey[k][keyRound],
            new MaskedHDPowerModel(l, mask, 0, r, mask)
        );
    }
}

```

```

    );
}

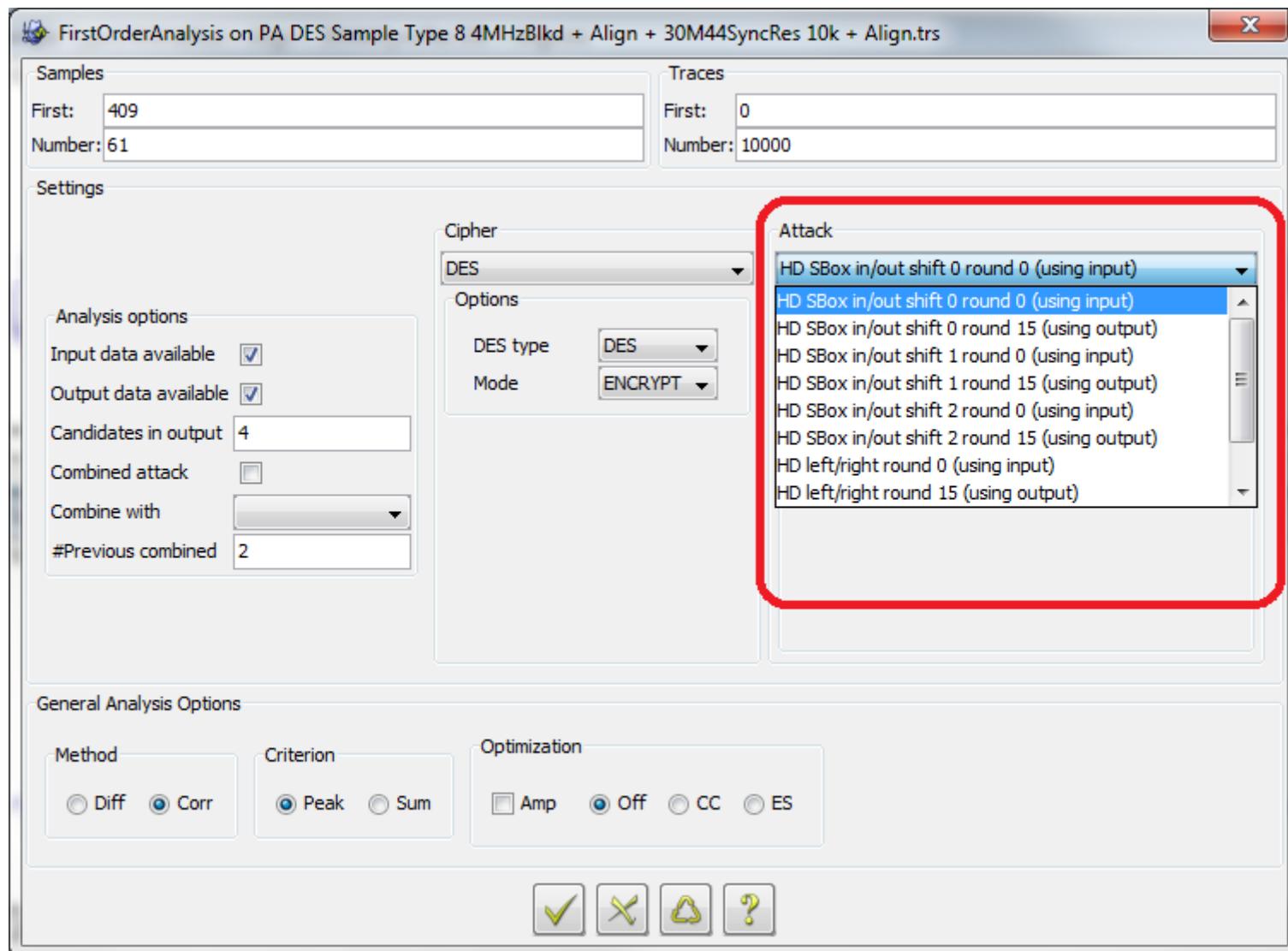
return keyPower;
}

```

The HWLeakageModel and MaskedHDLeakageModel classes represent the Hamming weight leakage model and the Hamming distance leakage model respectively. See the LeakageModel [../javadoc/com/riscure/signalanalysis/crypto2/LeakageModel.html] interface for details.

Having implementing the Intermediates and the Leakages interfaces for a cipher, one should see all the specified leakage models of the selected intermediate values in the "Attack" panel of the FirstOrderAnalysis module's UI once the cipher is selected.

Figure 8.6. The "Attack" panel for DES in FirstOrderAnalysis module.

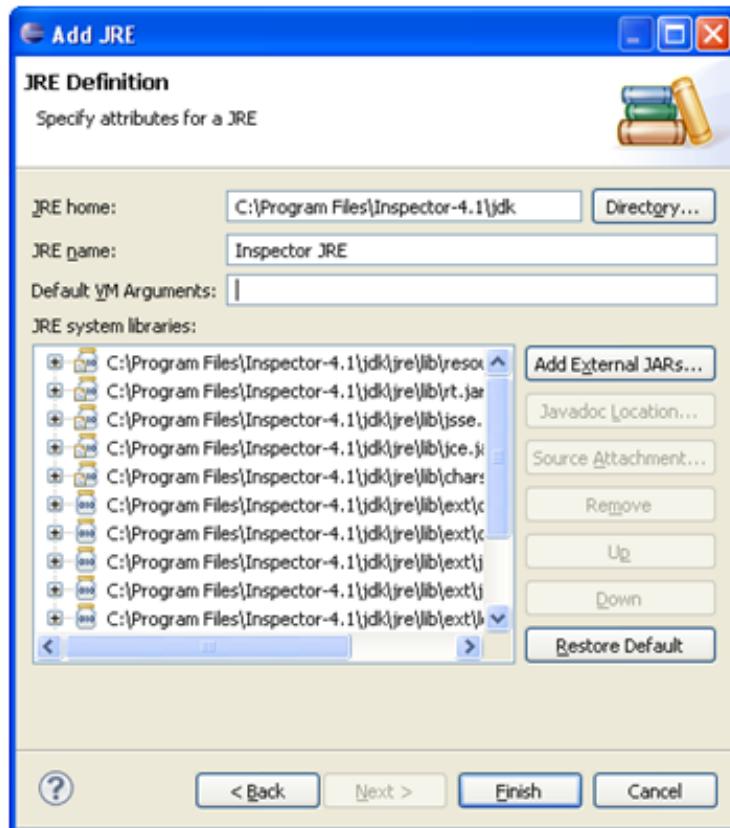


8.3.5 Module development with Eclipse

In addition to using the development environment integrated with Inspector, it is possible to use any other Java development environment. This section describes how to configure Eclipse for Inspector module development.

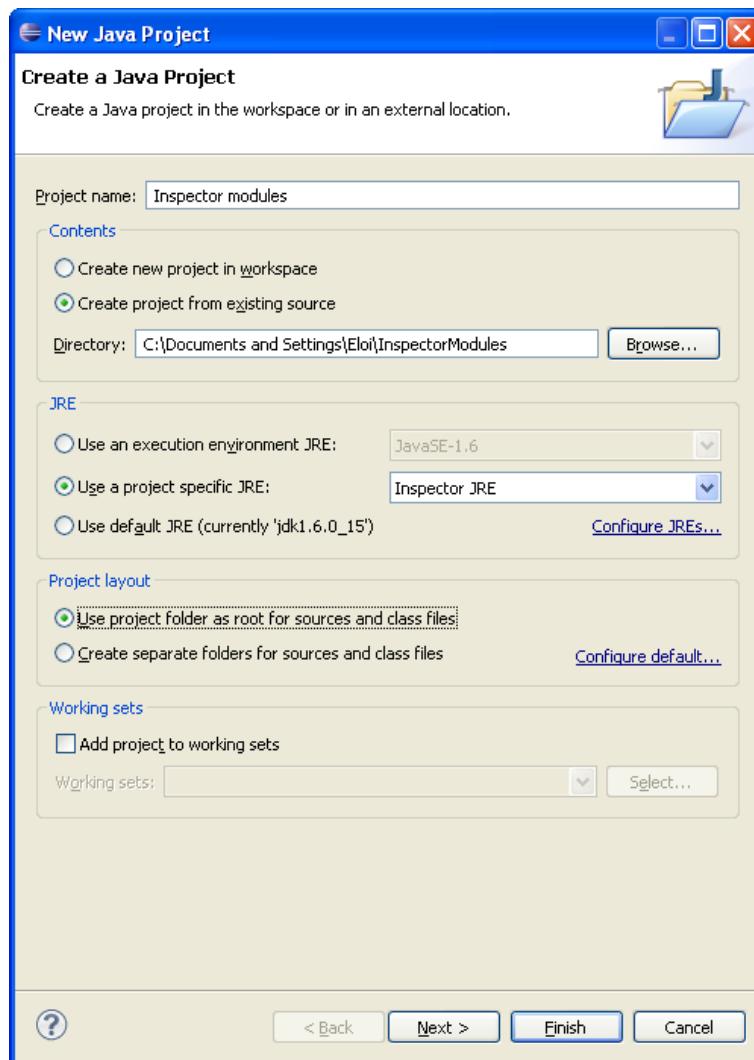
Before creating a new Project for the modules, we will set up Eclipse to use the Java SDK shipped with Inspector. To that end, go to *Window > Preferences* and select *Java > Installed JREs*. Click 'Add' and point the JRE home to the *jdk* folder inside the Inspector installation directory.

Figure 8.7. Eclipse new JRE dialog



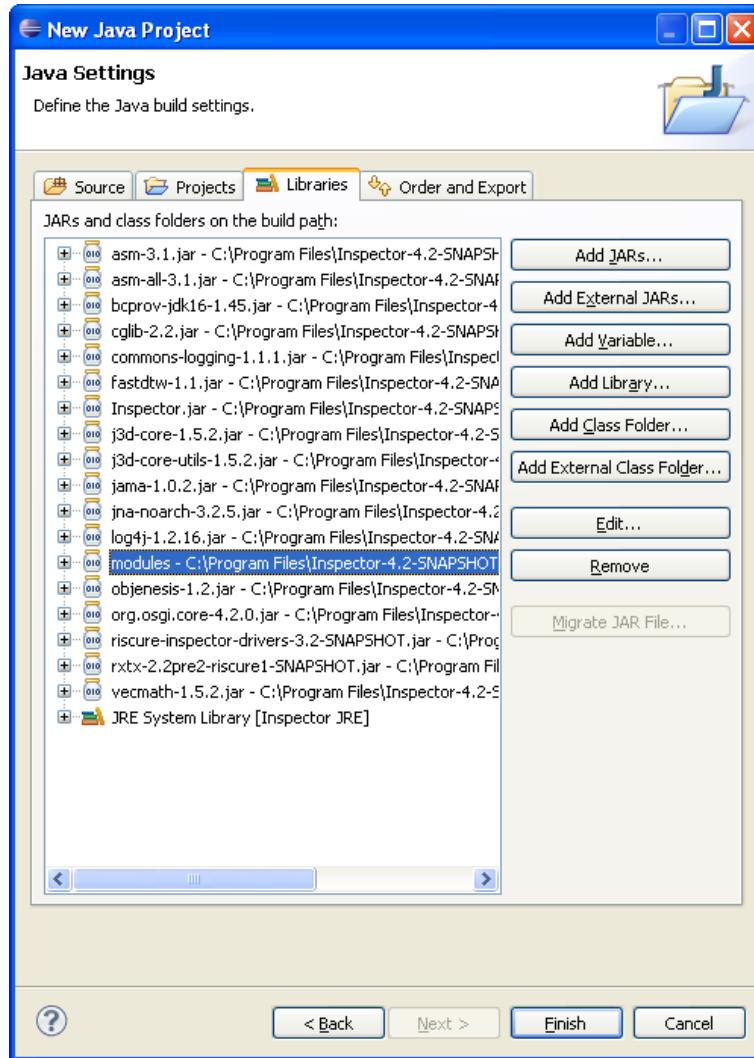
Next, we can create a new process. Go to *File > New > Java Project* and create a project from existing source. Point it to the folder where your user modules are located. You should select the Inspector JRE and the option *Use project folder as root for sources and class files*.

Figure 8.8. Eclipse new Java Project dialog



After pressing next, go to the Build path tab and add the Inspector.jar file and all the jar files in the lib folder under the Inspector installation directory. Also, press 'Add External Class Folder..' and add the modules/ folder which contains the system modules. If the JRE selected is not the Inspector JRE, select it and press *Edit...* to select the Inspector JRE.

Figure 8.9. Eclipse new Java Project dialog. Class path settings



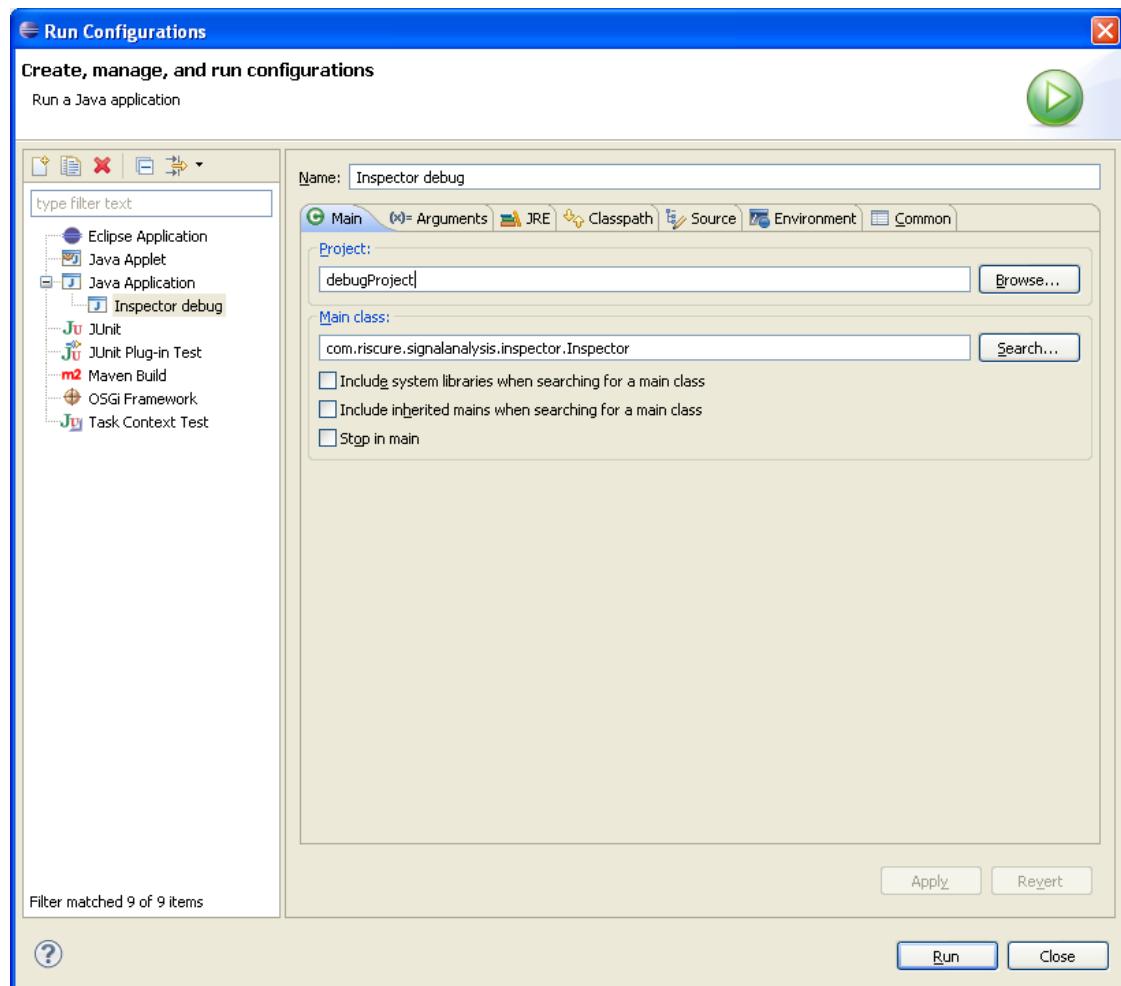
At this point, the project setup is done. After pressing Finish, you will be able to create and compile new modules using all the features of Eclipse, such as syntax highlighting, auto-complete of variable names, support for javadoc documentation, etc.

Debugging Inspector modules with Eclipse

Although the Eclipse editor adds a number of features to the standard Inspector editor, a very useful feature of Eclipse is the integrated debugger. With it, we can ask Eclipse to break at a given line in our modules, inspect the values of variables and step through the code in order to figure out what might be wrong in our code.

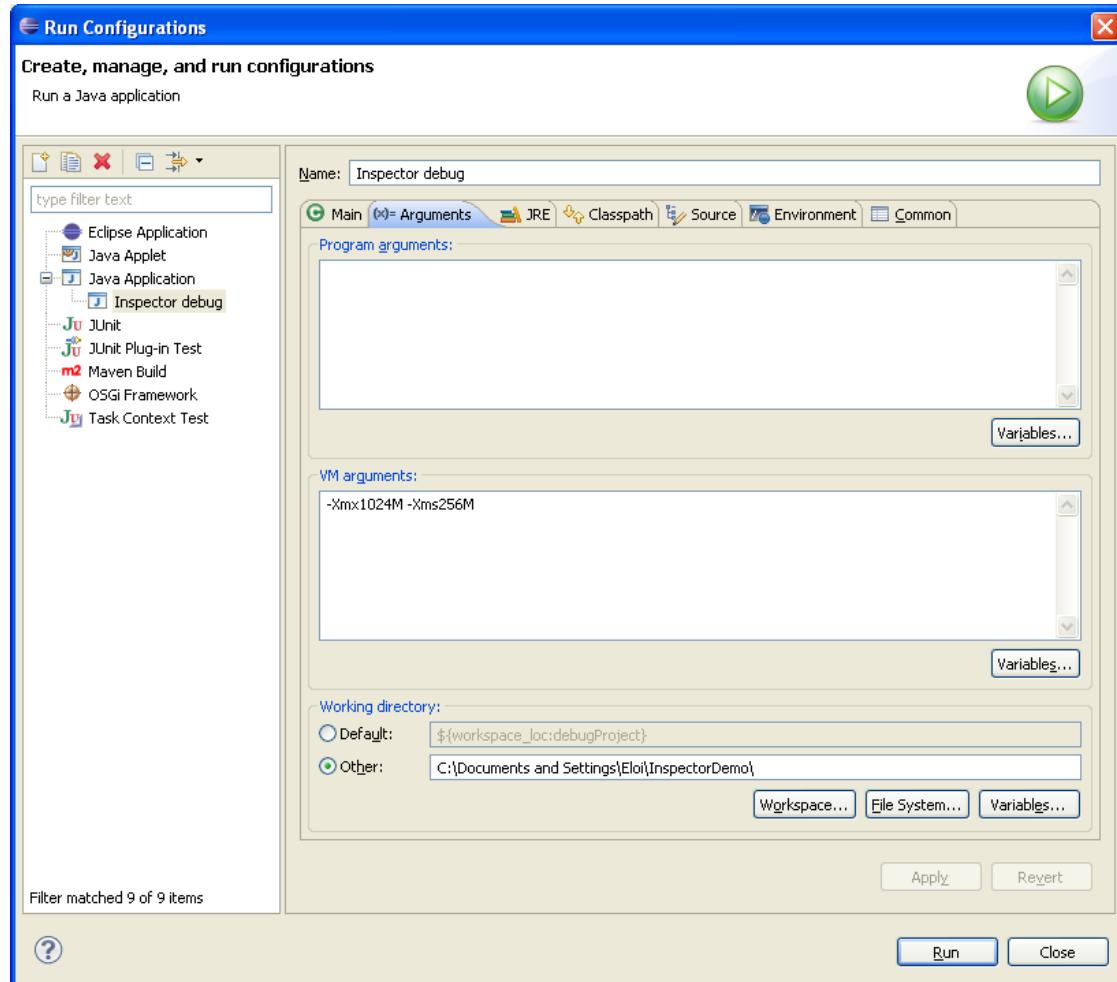
In order to be able to do debug modules, we need to create a *Run configuration* for Inspector. To that end, first create a new Java Project on the current workspace and then go to *Run > Run configurations* and double-click on *Java Application*. Select the new project and set the Main class to *com.riscure.signalanalysis.inspector.Inspector*.

Figure 8.10. Eclipse new Run configuration dialog



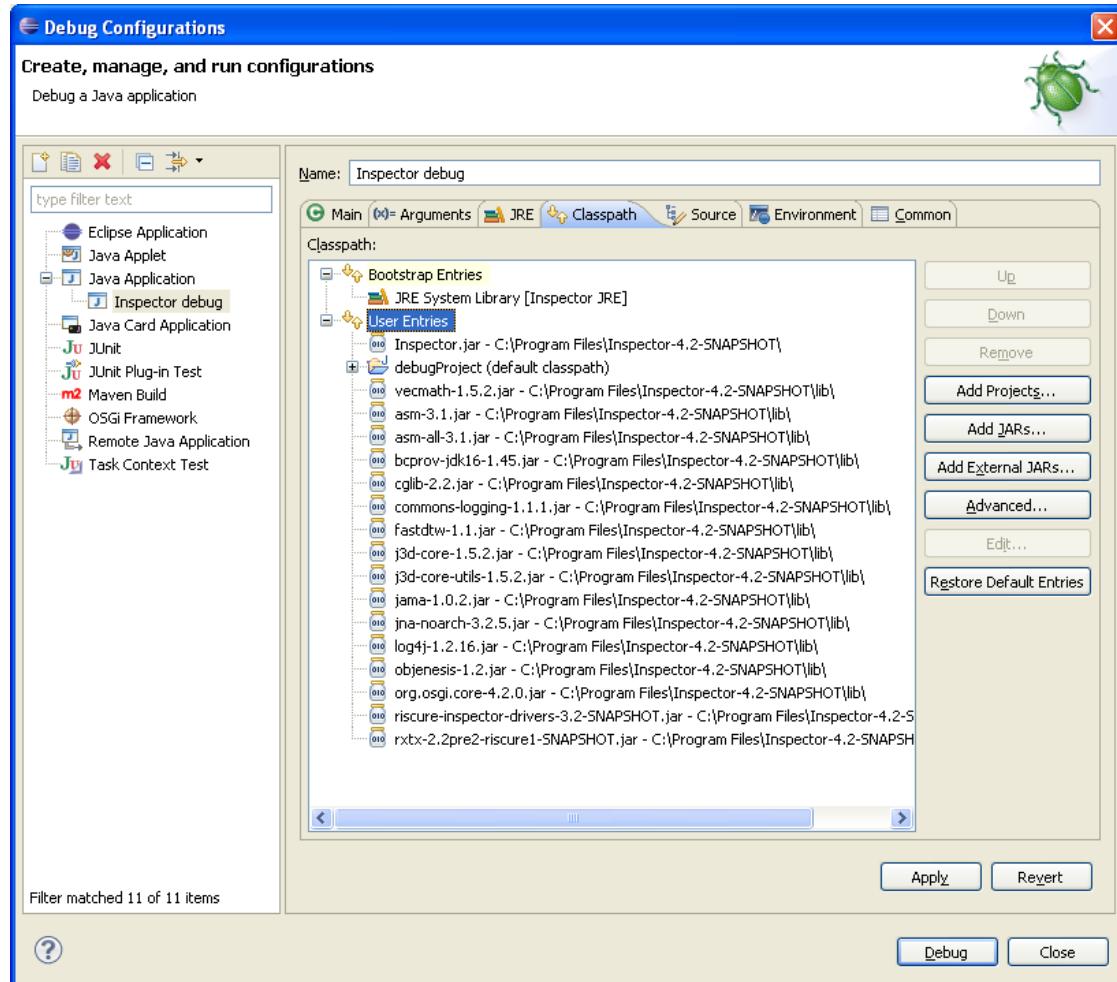
In the Arguments tab, set the VM arguments field to `-Xmx1024M -Xms256M -XX:PermSize=64M -XX:MaxPermSize=128M`. Configure the Inspector working directory to point to the folder where you want to store your Inspector settings for debugging purposes. In our case, we select the parent directory to our user modules directory.

Figure 8.11. Eclipse new Run configuration dialog. Arguments settings



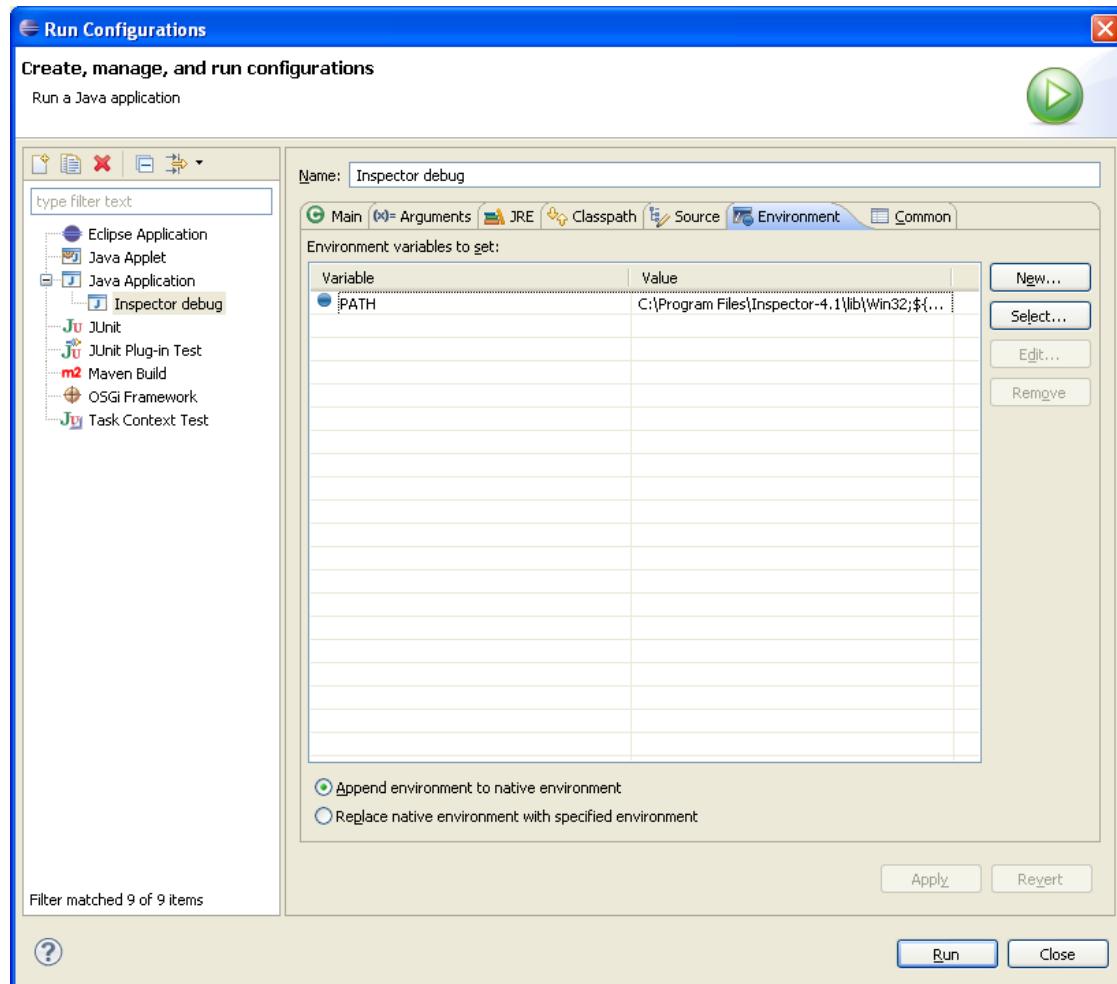
In the JRE tab, select the Inspector JRE. In the Classpath tab, press Add External JARs and add the Inspector.jar archive and all the jar files in the lib/ directory. Make sure Inspector.jar is the first item on the list of User Entries by selecting it and pressing Up as many times as needed.

Figure 8.12. Eclipse new Run configuration dialog. Classpath settings



Finally, go to the Environment tab and add a new variable. Set the name to PATH and the value to <InspectorDirectory>\lib\Win32;\${env_var:PATH}. This will set prepend the lib\Win32 directory to the system path so that Inspector can find the required libraries.

Figure 8.13. Eclipse new Run configuration dialog. Environment settings



Finally, add your module project in the Sources tab, so that Eclipse can find the code for your user modules while debugging. After this process, you should be able to launch Inspector in debug mode (*Run > Debug* or F11) and ask Eclipse to break the process at a given line in your modules by going to that line and enabling a breakpoint (Ctrl+B). For more information on how to use the Eclipse debugger, see the debugging section in the Eclipse user manual.

8.3.6 Writing perturbation modules

Perturbation modules control the glitch process. There are two categories of perturbation modules:

- SmartCardPerturbation, SmartCardOpticalPerturbation:

This category uses the internal cardreader of the VCGlitcher to connect with the target. Perturbation Programs are used for communications and glitching the target. This section describes writing modules for this category.

- TriggeredPerturbation, TriggeredOpticalPerturbation:

This category uses an external reader device to communicate with the target, using the trigger from that device as an input to the VCGlitcher. Writing perturbation modules of this category is described in Writing triggered perturbation modules.

All perturbation modules using the VCGlitcher internal card reader must extend the SmartCardPerturbation class for VCC/CLK perturbations, or the SmartCardOpticalPerturbation class for optical perturbations. Five abstract methods must be implemented:

- `void initPerturbationModule()`
- `void initPerturbationProcess()`
- `void runPerturbation()`
- `int filter(byte[] logData)`
- `void preparePerturbationProgram(PerturbationProgram p)`

In this section, these methods are described in more detail. The timing diagram at the end of this section shows the order of invocation of these methods.

8.3.6.1 Class fields

All perturbation modules use parameters to configure their perturbation program. Each parameter has its own unique identifier which is used by Inspector to save and load their value. A new identifier can be defined as follows:

```
public static final String PARAM_GLITCH_CYCLES = "glitchcycles";
```

Note that this parameter is already included in most perturbation modules by default.

In addition, all commands (i.e. APDUs) must be declared, for example:

```
private Command doDesCommand;
```

The actual bytes of the APDU will be defined in the `initPerturbationModule` method.

8.3.6.2 initPerturbationModule

This method is executed when loading the perturbation module.

8.3.6.3 Commands (APDUs)

Commands (i.e. APDUs) should be defined as follows:

```
selectCommand = new Command("00 A4 04 00 07 A0 00 00 00 FF F0 01");  
doDesCommand = new Command("A0 04 00 00 08 C7 39 D7 EA FA E4 ED A3 00");
```

Note that the commands may only be defined in the `initPerturbationModule` method, as information about the APDUs is used when compiling the perturbation program.

8.3.6.4 Parameters

In this method the perturbation parameters can be defined. The constructor of Parameter requires 6, 7 or 8 arguments depending on the desired parameter type.

Table 8.1. Creating a new parameter

| Argument | Description | FIXED | RANDOM | RANGE |
|-------------|---|-------|--------|-------|
| name | The name of the parameter | V | V | V |
| title | The title of the parameter (will be shown in the GUI) | V | V | V |
| description | The description of this parameter | V | V | V |
| type | The type of this parameter: <ul style="list-style-type: none">• ParameterType.FIXED• ParameterType.RANDOM• ParameterType.RANGE | V | V | V |
| format | The format of this parameter: <ul style="list-style-type: none">• ParameterFormat.INT• ParameterFormat.FLOAT• ParameterFormat.DOUBLE | V | V | V |
| min | | V | V | V |
| max | | N/A | V | V |
| step | | N/A | N/A | V |

Suppose that we need a parameter that sets the number of consecutive clock cycles that contains a glitch. This parameter can be created as follows:

```

fixedGlitchCycles = new Parameter(PARAM_GLITCH_CYCLES, "Glitch cycles", "The number
of consecutive cycles to glitch",
ParameterType.FIXED, ParameterFormat.INT, 1);

randomGlitchCycles = new Parameter(PARAM_GLITCH_CYCLES, "Glitch cycles", "The number
of consecutive cycles to glitch",
ParameterType.RANDOM, ParameterFormat.INT, 1, 20);

rangeGlitchCycles = new Parameter(PARAM_GLITCH_CYCLES, "Glitch cycles", "The number
of consecutive cycles to glitch",
ParameterType.RANGE, ParameterFormat.INT, 1, 20,
1);

// In addition these parameters should be added to the parameter manager using the
// addGlitchParameter() method
// this line adds the 'randomGlitchCycles' parameter to the parameter manager.

addGlitchParameter(randomGlitchCycles);

```

8.3.6.5 Smart card clock speed

By default, the smart card clock speed is set to 1 MHz. Alternatively the clock speed can be set to 2, 3 or 4 MHz. For setting the smart card clock speed to 2 MHz:

```
setClockSpeed(Pattern.PATTERN_2MHZ_CLOCK);
```

In order to synchronize the VC Glitcher with the smart card clock, the actual clock frequency is not always exactly the same as the specified. Table 8.2, “Smart card clock frequency” shows the actual frequencies corresponding to the constants.

Table 8.2. Smart card clock frequency

| Constant | Actual frequency | Glitch pattern length |
|----------------------------|------------------|-----------------------|
| Pattern.PATTERN_1MHZ_CLOCK | = 1.000 MHz | 500 |
| Pattern.PATTERN_2MHZ_CLOCK | ~ 2.083 MHz | 240 |
| Pattern.PATTERN_3MHZ_CLOCK | ~ 3.125 MHz | 160 |
| Pattern.PATTERN_4MHZ_CLOCK | ~ 4.167 MHz | 120 |

The length of the glitch pattern, as described in Section E.11.1, “Perturbation Module”, also depends on the chosen smart card clock frequency.

8.3.6.6 Other settings

`atrParseVoltage` can be set to the voltage that is used when parsing the ATR. By default 5V is used during the ATR parsing. If you require a different voltage, you can use the `atrParseVoltage` variable to change the voltage. For example:

```
atrParseVoltage = 3.0f;
```

The `disableLogging` boolean can be set to prevent Inspector FI from creating a report. For example:

```
disableLogging = true;
```

8.3.6.7 void initPerturbationProcess

This method is executed before the perturbation module is started (i.e. after clicking the start button on the perturbation module dialog). For most standard perturbation modules, this method does not contain code.

8.3.6.8 void runPerturbation

This method is executed for each iteration in the test run. Normally, four tasks are performed in this method:

- Setting the new glitch pattern
- Randomizing the APDUs (optional)
- Executing the perturbation program
- Storing (parts of) the response data in the resulting trace set

Setting the new glitch pattern

In order to set the new glitch pattern, one can use the `glitchPattern` object. Before each iteration the glitch pattern is cleared. Table 8.3, “Methods in the Pattern class” shows the available methods for manipulating the glitch pattern.

Table 8.3. Methods in the Pattern class

| Method | Description |
|--|--|
| <code>void setBit(int bit, boolean value)</code> | Set the specified bit to specified value |

| Method | Description |
|--|---|
| <code>void setBit(Parameter bit, boolean value)</code> | Set the specified bit to specified value. The bit position is specified using a perturbation parameter. |
| <code>void setBits(int startBit, int length, boolean value)</code> | Set the specified bit range to specified value |
| <code>void setBits(Parameter startBit, Parameter length, boolean value)</code> | Set the specified bit range to specified value. The bit range is specified using perturbation parameters. |

Note that both `setBit` and `setBits` can be used with integers or by using Parameter objects.
Some examples:

Table 8.4. Examples of setting the glitch pattern

| Method | Description |
|---|--|
| <code>glitchPattern.setBit(100, true);</code> | Set bit 100 to 1 |
| <code>glitchPattern.setBit(100, false);</code> | Set bit 100 to 0 |
| <code>glitchPattern.setBits(100,10,true);</code> | Set the bit range 100-110 to 1 |
| <code>glitchPattern.setBits(glitchOffset, glitchLength, true);</code> | Set the bit range defined by the two parameters to 1 |

We recommend using the parameterized version in combination with the GUI.

Randomizing the APDUs (optional)

If an APDU needs to be randomized each measurement, you can use the `randomize(int offset, int length)` method of the Command class. For example:

```
doDesCommand.randomize(5,8);
```

This will randomize bytes 5 – 12.

It is also possible to change individual bytes. For example:

```
doDesCommand.setByte(5,0xAB);
```

This will change the byte at offset 5 to 0xAB.

Executing the perturbation program

Executing the perturbation program is straightforward:

```
response = executeGlitchProgram(2000);
```

The value 2000 indicates the perturbation program timeout value in milliseconds. In this example, the perturbation program will be terminated if it is not finished after 2 s. The `executeGlitchProgram` method returns the data received while executing the program. Note that this includes both the received as the sent data (i.e. the ATR, commands and responses including protocol overhead).

Storing (parts of) the response data in the resulting trace set

For DFA attacks response data needs to be stored with the trace sets. Observe the following example:

```
setData(response, 23, 8);
addData(response, 36, 8);
```

This will add the command (at offset 23, length 8) and the response (at offset 36, length 8) to the current trace. The actual values depend on the smart card/application under evaluation.

8.3.6.9 filter(byte[] logData)

The filter method is called before storing the data of the current iteration in the database. It allows to user to highlight or exclude results. For comparing received data with the expected results, one can use the utility functions of the Util class.

Table 8.5. Methods in the Util class

| Method | Description |
|---|---|
| String toHexString(byte[] data) | Converts byte array into a hexadecimal string. |
| byte[] toByteArray(String hex) | Converts a hexadecimal string into a byte array. |
| boolean startsWith(byte[] a, byte[] b) | Checks if the byte array b is a prefix for byte array a. |
| boolean endsWith(byte[] a, byte[] b) | Checks if the byte array b is a suffix for byte array a. |
| boolean compare(byte[] a, byte[] b) | Checks if two byte arrays are equal. Returns true if the two arrays are equal, false otherwise. |
| boolean compare(byte[] a, byte[] b, int offsetA, int offsetB, int length) | Checks if two partial byte arrays are equal. Returns true if the two partial arrays are equal, false otherwise. |
| int firstDifference(byte[] a, byte[] b, int offsetA, int offsetB, int length) | Returns the offset of the first difference between two byte arrays. |

This function should return one of the values listed in Table 8.6, “Possible return values of the filter method”.

Table 8.6. Possible return values of the filter method

| Return value | Description |
|--|---|
| Filter.EXCLUDE | The data will not be shown or stored in the database. |
| Filter.INCLUDE | The data will be shown and stored in the database. It will not be highlighted. |
| Filter.HIGHLIGHT_RED, Filter.HIGHLIGHT_GREEN, Filter.HIGHLIGHT_YELLOW | The data will be shown and stored in the database. It will be highlighted red, green or yellow. |

An example:

```
static String sDefaultData = "3B E6 00 FF 81 31 FE 45 4A 43 4F 50 33 31 06 00 00
0E A0 04 00 00 08 C7"+
                    "39 D7 EA FA E4 ED A3 00 31 00 00 0A C5 D9 1B 0E 2A
D5 6F BE 90 00 BD";

static byte[] bDefaultData = Util.toByteArray(sDefaultData));

public int filter(byte[] logData) {
    int rValue=Filter.HIGHLIGHT_RED;
```

```
if (Util.compare(logData, Util.toByteArray(defaultData))) {  
    rValue=Filter.HIGHLIGHT_GREEN;  
}  
  
return rValue;  
}
```

For consistency, we recommend to use the same color coding as described in Glitcher Report of the previous section. For long test runs it is recommended to exclude common results from the report by returning Filter.EXCLUDE.

8.3.6.10 void preparePerturbationProgram(PerturbationProgram p)

This method is used by the VC Glitcher to prepare the perturbation program. Note that by calling methods of the PerturbationProgram object p, instructions are being added to the perturbation program that will run on the VC Glitcher. Therefore, be careful with using conditional Java statements.

Suppose we want to receive 5 bytes from the smart card. A possible solution would be:

```
for (int i=0; i<5; i++) {  
    p.receive(p.R0);  
}
```

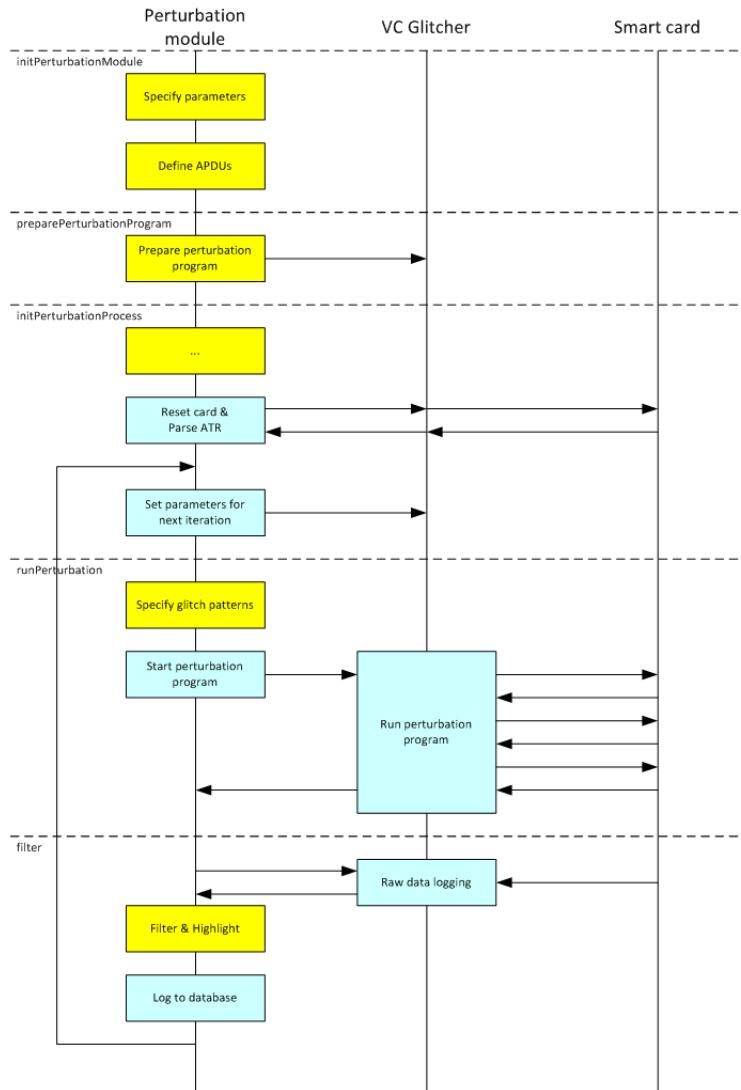
However, this would result in a program with 5 consecutive receive instructions. The actual loop will not be part of the program. In order to include the loop in the program, use:

```
p.load(5, p.R0);           // Loads 5 in register R0  
p.label("loop");          // Defines a Label named "Loop"  
p.receive(p.R1);           // Receive and store byte in R1  
p.subtract(p.R0, 1);       // Decrements R0  
p.jumpIfNotZero(p.R0, "loop"); // Jump to Loop if R0≠0
```

Note that pseudo instructions exist for many common tasks. For example, one can also use p.receiveBytes(5). For more information on writing perturbation programs and more information about the available (pseudo-)instructions, refer to Section H.1, “VC Glitcher API”.

Figure 8.14, “Timing diagram of a typical test run.” shows the timing diagram of a typical test run. The yellow tasks have to be implemented by the user. The blue tasks are automatically performed by Inspector FI.

Figure 8.14. Timing diagram of a typical test run.



8.3.7 Writing triggered perturbation modules

Triggered perturbation modules use an external event to trigger a perturbation attempt.

A custom triggered perturbation module should extend the TriggeredPerturbation or TriggeredOpticalPerturbation class depending on the type of perturbation required. The TriggeredOpticalPerturbation class initializes the Laser Diode Station equipment and adds a Microscope tab.

Developing a custom triggered perturbation module is similar to developing an acquisition module with elements taken from the SmartCardPerturbation classes. It is recommended to read the section on Writing Perturbation modules before reading this section.

Not all of the methods required for SmartCardPerturbation are required for TriggeredPerturbation: The preparePerturbationProgram(PerturbationProgram p) method is still present, but it now holds a default implementation of the perturbation program that suits the need for most triggered perturbation modules.

Eight additional methods can be used in the custom module:

- void logData(String data)
- void armVCGlitcher()
- String cookedCommand(String hexString)
- byte[] cookedCommand(byte[] data)
- ResponseData rawCommand(byte[] data)
- ResponseData rawCommand(byte[] data)
- ResponseData rawCommand(string[] data)
- ResponseData rawCommand(string[] data)

The method void runPerturbation is executed for each iteration in the test run. Normally, five tasks are performed in this method:

- Setting the new glitch pattern
- Randomizing the APDUs (optional)
- Arm the VC Glitcher
- Send a (sequence of) APDUs
- Storing (parts of) the response data in the resulting trace set

Setting the new glitch pattern

In order to set the new glitch pattern, one can use the glitchPattern object. Before each iteration the glitch pattern is cleared. Table 8.3, “Methods in the Pattern class” shows the available methods for manipulating the glitch pattern.

We recommend using the parameterized version in combination with the GUI.

Randomizing the APDUs (optional)

If an APDU needs to be randomized each measurement, you can use randomize(hexString data, int offset, int length) inherited from Acquisition.

Arming the VC Glitcher

This done by calling armVCGlitcher() immediately before the command sequence to be perturbated.

armVCGlitcher() loads the new parameter values and the patterns into the VC Glitcher and starts the perturbation program. The program causes the VC Glitcher to wait for a trigger.

The perturbation program supports four trigger modes: RISE, FALL, HIGH and LOW. HIGH and LOW are level-based triggers, RISE and FALL are edge-based. A different trigger mode can be selected by modifying the value for the built-in variable trigger. This variable should

be changed in the initPerturbation() method, and takes one of the values TriggerMode.RISE, TriggerMode.FALL, TriggerMode.HIGH, TriggerMode.LOW. Example :

```
trigger = TriggerMode.FALL
```

causes the VC Glitcher to wait for a HIGH-to-LOW transition on the trigger input.

The default is a rising edge trigger.

Send a sequence of APDUs

The command sequence is similar to how an acquisition module would send commands to the target. However, due to the purpose of perturbation modules versus acquisition modules a distinction is made in how the commands are sent to the device.

- in *cooked* mode, the command is sent as it would have been sent using the command(...) method of the Acquisition module. Protocol error checking applies, and permanent device errors cause a termination of the acquisition process. The cookedCommand(...) methods implement this mode.

Cooked commands are suitable for preparation tasks, for example to select the target applet on a smart card.

- in *rawmode*, the command that is sent is assumed to be perturbated, and might cause a device error. The acquisition should not halt when this happens, and any available data should be returned to the perturbation module. The rawCommand(...) methods return a ResponseDataobject, which contain the (optional) response data and the conditions in which the data was received.

Storing (parts of) the response data

Triggered perturbation modules output a trace set and a perturbation attempts table. Different types of data are stored for each of these outputs, resulting in different methods to attach data to a single attempt.

- For DFA attacks response data needs to be stored with the trace sets.

```
setData(command, 23, 8);  
addData(response, 36, 8);
```

This example will add the command input parameter (at offset 23, length 8) and the response (at offset 36, length 8). The actual values depend on the smart card/application under evaluation.

- The perturbation table is fed with data using logData(byte[] data, boolean timedOut):

```
response=rawCommand(command);  
  
logData(response.toBytes(),  
        response.getResult() == ResponseData.Result.TIMEOUT);
```

This example logs the response data and sets the *timedOut* boolean to *true* if the response timed out.

responseData

A ResponseData object is returned by rawCommand(...) methods and holds the command and response data as well as the conditions in which the data was received.

The conditions are expressed in the Result enumeration, which is returned by the getResult() method and can have the following values:

Table 8.7.

| Value | Description |
|-------------------------------|--|
| <code>Result.OK</code> | There were no errors in receiving the response. |
| <code>Result.TIMEOUT</code> | No data received (device timings apply) |
| <code>Result.UNDERFLOW</code> | Received less data than expected |
| <code>Result.OVERFLOW</code> | Received more data than expected |
| <code>Result.UNDEFINED</code> | An error has occurred but none of the other result codes match the error condition. |
| <code>Result.EXCEPTION</code> | The device has thrown an exception, and the exception message can be retrieved by calling getConditionMessage(). |

other methods

- `toBytes()`

Returns the response as an array of bytes

- `toHexString()`

Returns the response as a String representing the response in hexadecimal encoding. Each byte is separated by a space

- `toFlowString()`

Returns a String showing the command and the response in hexadecimal encoding, prepended with a > to indicate the command and < for the response.

8.4 Inspector OSGi framework

Starting from version 4.4, Inspector is using a new software architecture based on OSGi. This architecture requires a different approach to implementing new features or extending existing functionality in Inspector. The following sections describe various aspects of software development, focusing on what is specific for Inspector.



Expert Knowledge

The following sections assume that the reader has a profound knowledge of the Java programming language and familiarity with OSGi concepts.

8.4.1 OSGi plugins

OSGi plug-ins include device drivers, target managers, and transport protocols. An OSGi plug-in is bundled in a Java ARchive including a text file containing meta-data about how the plug-in should be loaded.

The meta-data file resides inside the JAR file in the directory META-INF\services and is called `org.osgi.framework.BundleActivator`. This file contains a single line with the

full name (including package) of the BundleActivator class which is responsible for loading the plug-in. The mechanism used is in accordance with the OSGi Service Platform [<http://www.osgi.org/>]. The BundleActivator should register the plug-in with the service registry in order for it to be used by Inspector. At start-up all JAR files in the lib directory are scanned for the specified meta-data and if found the plug-in is loaded.

8.4.1.1 Properties

All plug-ins, except for modules, expose their user configurable settings through JavaBeans. JavaBeans allow for exposing properties to the user without the need to explicitly code user interface elements. JavaBeans also allow for automatic validation of parameters.

A JavaBean is a Java class with fields together with getter and setter methods to access these fields.

```
public class SimpleProperties {  
  
    private int number;  
    private String name;  
  
    public int getNumber() {  
        return number;  
    }  
  
    public void setNumber(int number) {  
        this.number = number;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

Based on this JavaBean Inspector will automatically generate an user interface which looks approximately like this:

The fields in the bean can be annotated in order to provide Inspector with hints on how to present them. Also annotations can be used to add validators. If one of the validators fails the user will be presented with an error message and will not be allowed to proceed. This significantly reduces the need for input validation in source code.

An example of the same bean but now with annotations added for presentation and validation.

```
public class AnnotatedSimpleProperties {  
  
    @DisplayName("Expected amount of input")  
    @Unit("bytes")  
    @Presentation(base=16)  
    @Min(0)  
    @Max(1024)  
    private int number;
```

```
@DisplayName("Name")
@ShortDescription("Name of the variable")
@NotNull(message="Name must not be NULL")
@Length(min=1, max=10)
private String name;

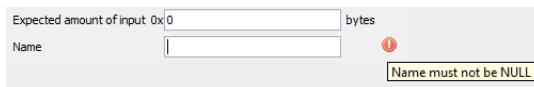
public int getNumber() {
    return number;
}

public void setNumber(int number) {
    this.number = number;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}
}
```

The field 'number' now has a name and a unit, the number will be presented in hex (base 16) and must be between 0 and 1024 (decimal). The 'name' field is simply named "Name", but has an extended help text which will be shown when the user hovers the mouse cursor over the field. The name can not be 'null' and must be between 1 and 10 characters. The generated user interface will look similar to:



The interface displays the field names and units are annotated. The hexadecimal presentation is noted by the 0x in front of the number field. The 'name' field contains a validation error because the field is empty as shown by the red exclamation mark. If the user hovers the mouse over the exclamation mark a tooltip containing the error message will be shown.

Additional annotations can be found in the JavaDoc in package com.riscure.beans.annotation [../javadoc/com/riscure/beans/annotation/package-summary.html]. For more validation options check the JavaBean validation reference implementation Hibernate Validator [<http://www.hibernate.org/subprojects/validator.html>] and check the additional validators provided in com.riscure.beans.constraints [../javadoc/com/riscure/beans/constraints/package-summary.html].

8.5 Developing application protocols

Application protocols are internal plug-ins implementing the Protocol [../javadoc/com/riscure/signalanalysis/acquisition/Protocol.html] interface. Applications protocols are used by the acquisition framework to communicate with a target (e.g. smartcard) over an I/O device (e.g. card reader). Several application protocols are provided including those needed to communicate with the training cards. The user can implement a new application protocol for use with a specific target.

On start of the acquisition the Protocol's init method is called once. After that for every measurement the run method is called. Both methods are called with a reference to ProtocolTarget which allows the protocol to send and receive data to the target.

In the new Acquisition2/Perturbation2 framework it has become easier to implement (or extend) your own application protocol. The module wizard provides a simple means to create boilerplate code that will immediately work. See Section 9.8, “How do I implement an acquisition2/perturbation2 application protocol?” for more details.

8.6 Developing device drivers

Inspector supports a wide range of devices, including oscilloscopes, I/O devices (e.g. card readers), and JTAG devices. If the device drivers provided by Inspector prove insufficient a new driver can be developed to support devices within Inspector. In order to achieve this an external plugin has to be developed which is placed in the Inspector lib directory.

8.6.1 Driver classes

Several driver classes exist depending on how the device can be detected by Inspector.

8.6.1.1 Static

Static drivers always register a device with Inspector regardless of whether it is available or not. This is the easiest type of driver to create, however selecting the device when it is not connected may result in long waiting times while the driver attempts to connect to the device. Also only a single instance of a device can be registered.

Static drivers should register the appropriate device class with Inspector in the *start* method of the driver activator.

8.6.1.2 Plug and play

Plug and play drivers detect the presents of a device and only then register the device with Inspector. This allows Inspector to present a list of currently attached devices to the user and speed up device selection. A plug and play driver can register multiple devices of the same type with Inspector.

At startup plug and play drivers should register a DeviceScanner with Inspector. All registered DeviceScanners will be notified of devices being connected and disconnected as detected by the Windows device management API. Alternatively the driver could implements a custom device detection methodology by spawning a thread in the *start* method of the driver Activator. Once a device is detected the appropriate class instance should be registered with Inspector. If a device is disconnected the class instance should be unregistered.

8.6.1.3 Manual

Devices which can not be automatically detected, such as network attached devices, can be added to Inspector by the user through the hardware manager. The driver should register an instance of Driver in the start method of the Activator. The Hardware Manager will then offer the user the possibility of selecting the driver when clicking Add Device. Configuration necessary to connect to the device should be contained within a JavaBean returned by *createSetup*. The hardware manager will display the bean to the user allowing the user to provide the necessary details. Inspector will then instruct the driver to create the device by calling *createDevice* with the filled in bean. If *isDriverManagedPersistance* returns false Inspector will attempt to reconnect to the device at start up until the user removes the device again.

8.6.2 Device categories

8.6.2.1 I/O devices

These drivers should implement the `IIODevice` interface for devices which are transport protocol aware. An example is an ISO 7816 card reader or a serial port using an embedded protocol. The only method to be implemented is the `command` method, which takes the bytes to be send and returns the bytes received.

8.6.2.2 Oscilloscopes

If a oscilloscope does not provide IVI drivers a custom driver can be created by implementings the `Oscilloscope` interface. This will also allow custom properties to be included in `OscilloscopeProperties`, `TriggerProperties`, and/or `ChannelProperties` in order to configure scope specific settings from within Inspector.

8.6.2.3 Raw I/O devices

Raw I/O devices are devices that do not apply a transport protocol to the data sent and received to the target, for example a serial port. A custom raw I/O device can be created by implementing the `RawIODevice` interface and implementing an `InputStream` and `OutputStream`.

If the desired transport protocol is already available in Inspector, it may be preferable to implement a raw I/O device and use the protocol implementation supplied with Inspector.

8.6.3 Guidelines

Some guidelines and recommendations for implementing device drivers:

- `DeviceInterface` instances should not attempt to open the device for communication, this should only be attempted when the `open` method is called in order to prevent unnecessary wait time in Inspector.
- The `close` method on `Session` should carefully release all claimed resources in order to prevent resource starvation when running Inspector for extended periods.
- Property beans can be extended with custom properties which will be displayed Inspector through an automatically generated user interface. If the generated user interface is insufficient a custom interface can be defined. See Section 8.4.1.1, “Properties”.

8.7 Developing measurement setups

The acquisition2 and perturbation2 frameworks interact with device drivers through *measurement setups*. Measurement setups are in essence building blocks that can be used to compose acquisition2 and perturbation2 modules. More specifically, a measurement setup is an abstraction layer between the acquisition2 and perturbation2 frameworks on the one hand and (measurement) hardware on the other, that hides details about specific devices connected to the PC and how to operate them from the frameworks. Instead it offers the frameworks a method to obtain measurement data. The specifics of the interaction between the framework and measurement setups will be discussed in the remainder of this section.



Note

For an example of an implementation of the measurement setup abstraction layer, please see `acquisition2.measurementsetup.MultiScopeSetup`.

8.7.1 Measurement setup life cycle

Measurement setups are represented in code by the `com.riscure.signalanalysis.acquisition.MeasurementSetup` interface. This interface has several methods, which are called in a particular order by the acquisition2 and perturbation2 frameworks during the life cycle of a measurement setup. This section is subdivided according to the life cycle of a measurement setup.

8.7.1.1 Loading

In the acquisition2 and perturbation2 frameworks, modules indicate which measurement setups should be loaded when the module is started. acquisition2 and perturbation2 modules should override the abstract `createMeasurementSetups()` provided by `acquisition2.GenericAcquisition`, which is the super class of all acquisition2 and perturbation2 modules, for this purpose. perturbation2 modules should also override the abstract `createPerturbationMeasurementSetups()` method, provided by `perturbation2.module.GenericPerturbation`. A measurement setup returned by the latter method is expected to have perturbation properties, whereas a measurement setup returned by the former method should not contain perturbation properties. This is explained in more detail in Section 8.7.1.2, "Display and user configuration".

As a part of the loading process of a measurement setup, the frameworks will attempt to restore persisted user settings for this measurement setup if these settings exist. If persisted settings exist for this measurement setup, its `MeasurementSetup.setSettingsBean(Object)` method is called. The argument passed into this method will always be of the same type as the type of the object returned by an earlier call to `MeasurementSetup.getSettingsBean()` and holds the persisted user settings. The `setSettingsBean(Object)` method is expected to update its internal state according to the settings provided. This means that a subsequent call to `MeasurementSetup.getSettingsBean()` should return an object equivalent to the argument passed into `setSettingsBean(Object)`.

The frameworks persist the settings of a measurement setup along with the ID of that measurement setup. This ID is obtained from the `MeasurementSetup.getId()`, and this method is again used by the frameworks to match persisted settings to a certain measurement setup.

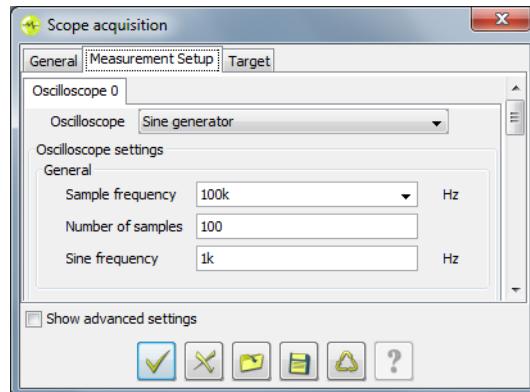
8.7.1.2 Display and user configuration

Every measurement setup indicated in this method will be displayed as 'a tab' in the module. This tab can be used by the user to configure the measurement setup. In order to create a tab for the measurement setup, the acquisition and perturbation2 frameworks call the `MeasurementSetup.getSettingsBean()` method, which should return an AGG compatible JavaBean.

The display name of the tab is obtained from the `MeasurementSetup.getName()` method, which returns a `String`. Figure 8.15, "The MultiScopeSetup implementation of `MeasurementSetup` selected in `ScopeAcquisition`" shows the tab titled 'Measurement setup' selected in the `ScopeAcquisition` module. This tab represents an instance of

the `acquisition2.measurementsetup.MultiScopeSetup` class, which implements `MeasurementSetup`.

Figure 8.15. The MultiScopeSetup implementation of MeasurementSetup selected in ScopeAcquisition



Perturbation properties (modifiable properties)

Some settings, such as the laser power during perturbation, change over the course of a measurement. Such settings are known as perturbation properties and the user configures them in 'Perturbation' tab of a perturbation2 module. The framework recognizes that certain properties are perturbation properties, because the JavaBean returned by `MeasurementSetup.getSettingsBean()` has been annotated with the `@Perturbation` annotation.

To illustrate the usage of perturbation properties we show the implementation of the glitch voltage variable found in all perturbation2 modules. See Example 8.1, "The perturbation parameter "Glitch voltage" in GlitchSetupSettings". This is a fragment of code found in the class file `perturbation2.measurementsetup.GlitchSetupSettings`, which is accessible via the menu option "File" -> "Open System Module" and then selecting `perturbation2/measurementsetup/GlitchSetupSettings.java`. The user is invited to study this class as an example for his own Settings object. This bean property also has a getter (`public BigDecimal getGlitchVoltage()`) and setter (`public void setGlitchVoltage(BigDecimal glitchVoltage)`), but these are not relevant for now.

Example 8.1. The perturbation parameter "Glitch voltage" in GlitchSetupSettings

```

@DisplayName("Glitch voltage")
@Unit("V")
@DecimalMin("-7.4")
@DecimalMax("4.2")
@DecimalStep("0.05")
@Presentation(slider = true)
@NotNull(message = "Glitch voltage not set")
@Perturbation
private BigDecimal glitchVoltage = BigDecimal.ZERO;
    
```

The first important thing to realize is that this variable is not shown on the "Glitch Setup" tab, but instead on the "Perturbation" tab. This is the result of the `@Perturbation` annotation. All bean properties with this annotation are collected on the dedicated "Perturbation" tab.

Secondly, we see that the other annotations describe the on-screen behavior of this variable. The display name will be "glitch voltage", the unit will be "V", it will be rendered as a slider (instead of a text input field) with minimum value -7.4, maximum 4.2 and step size 0.05. There is also a constraint that this variable must be set (i.e. not null), otherwise an error message will be displayed. The default value will be zero, which will be overridden by the consistency framework if there are stored parameters from a previous module run. A full list of these annotations (with formal descriptions) is given in Table 9.5, "An overview of annotations used by the AGG for the purpose of validation". Note that the GUI will also show a "Type" combobox with three different behaviors: Fixed, Random, and Range.

When the module is started it will behave according to the chosen "Type". If the user chose Fixed Inspector will take the specified value. If the user selected Random Inspector will generate a (uniformly) random value within the specified min-max interval. The third option will make Inspector iterate over the given range, starting with the min value and incrementing with the step value. For a more detailed description of the perturbation variables see Section E.11.1, "Perturbation Module".

8.7.1.3 Measurement run



Note

For a deeper understanding of this life cycle, it is recommended to read Section 8.8.1.3, "Measurement run" and study Figure 8.17, "Sequence diagram of the interaction between the framework, target and measurement setup" after having read this section.

When the user has finished configuring the module, and when the user settings have passed validation, the module will be run. At this point each the actual (physical) measurement will commence. Three methods are called in the order described below during this time, namely

- `MeasurementSetup.init()`; This method is called once during the life cycle of the measurement setup. This method can be used to for example connect with any devices (if this has not already happened during the user configuration stage) and applying user settings that will not change during the measurement (such as the clock speed of a smart card).
- `MeasurementSetup.prepare()`; This method is called to prepare the measurement setup for an individual measurement. This method is called as often as there are physical measurements. This method should be used to update settings that change between two measurements. Examples of such settings are the position of an XY-table and the laser power during perturbation.
- `MeasurementSetup.getResult()`; This method is called by the acquisition2 and perturbation2 frameworks to obtain measurement data. This method called as often as and after `prepare()` and is supposed to blocking until all data is available and can be returned. The data is returned of a `List<Data<?>>` object, where `Data<?>` is an interface used to wrap data for logging purposes. Any data deemed relevant should be returned by this method, such as a trace measurement from a scope, communication data with a smart card and verdict information of a perturbation attempt.

Number of measurements

The framework obtains the number of times that `prepare()` and `getResult()` should be called from the measurement setup itself. It does so by calling the `MeasurementSetup.getNumber()` method. This method is called once after `init()` and

before `prepare()` and should return the number of measurements that result from the user's configuration of the measurement setup excluding the configuration of perturbation properties (which is handled by the framework). If for example a measurement setup includes an XY table, and the user has configured it to scan a 10×10 grid, then `getNumber()` should return 100.

Arming and triggering

The `prepare()` method should not be used to arm or trigger a measurement device. If arming and/or triggering is required, the `MeasurementSetup` implementation should also implement the `com.riscure.hardware.Armmable` and `com.riscure.hardware.Triggerable` interfaces, which contain an `arm()` and `trigger()` method respectively. The abstraction layer for the Target under Evaluation is responsible for determining the right moment for the framework to call the `arm()` and `trigger()` methods. Details regarding this are discussed in Section 8.8.1.3, "Measurement run".

Aborting

A user has the option to abort a measurement. When a user aborts a measurement, the framework takes the necessary steps to stop its internal workflow. During that time, each measurement setup will be asked to abort as well, via a call to `MeasurementSetup.abort()`. For most measurement setups, this method needn't do anything, but it can for example be used to interrupt a busy-wait loop.

8.7.1.4 Closing

When all measurements have been performed by, the framework will ask all measurement setups to free its claimed resources, to this end it will call `MeasurementSetup.close()`. This method is mostly used to close any open connections with devices. This method is also called as a part of the clean-up process whenever any errors occur during the measurement.

8.7.2 Advanced measurement setups

Advanced measurement setups are subclass of measurement setups that give the implementer of such a measurement setup more freedom to control the UI for it, at the cost of having to implement certain features manually. This section is dedicated to the implementation of these features.

Advanced measurement setups are represented by the `com.riscure.signalanalysis.acquisition.AdvancedMeasurementSetup` interface and have essentially the same life cycle as regular measurement setups.

8.7.2.1 Custom/complex UI and validation

Advanced measurement setups are still displayed as a tab in an `acquisition2` or `perturbation2` module, but the contents of that tab are not derived from the `getSettingsBean()` method. Instead the `acquisition2` and `perturbation2` frameworks ask the advanced measurement setup to produce a UI by calling the `AdvancedMeasurementSetup.getPanel()` method, which returns a `JPanel`. Apart from any resizing, this `JPanel` is displayed as is in the tab of the measurement setup.

The advanced measurement setup is also responsible for validating the user input. Whenever user input validation is needed, the framework calls the `AdvancedMeasurementSetup.validate()` method. This method should return a

`Collection<String>`, which contains an error message for every invalid user input in the panel provided by the advanced measurement setup.

8.7.2.2 Persistence

Advanced measurement setups are asked to persist the current user settings, or restore previously persisted user settings at the same moment as is done for regular measurement setups, but via a different mechanism. When asked to persist user settings, the framework calls the `AdvancedMeasurementSetup.save(OutputStream)` method. The argument can be used by the advanced measurement setup to (indirectly) write the user settings to the persistent storage. When asked to restore user settings, the `AdvancedMeasurementSetup.load(InputStream)` method is called. This argument is used to read the persisted user settings and apply them. These two methods are the counterparts of the `getSettingsBean()` and `setSettingsBean(Object)` methods.

8.7.2.3 Modifiable Properties

The perturbation2 framework is designed to automatically iterate over all perturbation variables, like "glitch voltage", "glitch offset", etc.. It does this in a backtracking fashion, as explained in Section E.11.1, "Perturbation Module". The standard "types" of variable modification are "Fixed", "Range", and "Random". The user can write his own modifier and we will describe this in the following sections. First we'll give a conceptual introduction to the Modifiable Properties framework. This is followed by a technical overview in Section 8.7.2.4, "Overview of Modifiable Properties Framework". Then we'll dive into the implementation details in Section 8.7.2.5, "Perturbation 2 framework specifics". Finally, we take a tour through Inspector code to see a real Modifiable Property in action.

When the modifiable property framework is presented with a 'property', a term that should be understood according to the JavaBean definition of it, it places that property in one of two categories:

1. Properties of which the value can be varied
2. Properties of which the value cannot be varied.

The properties in the first category are known as 'modifiable properties'.

Then, the modifiable property framework attempts to find all ways in which the span of values that that property can assume can be defined. The most well-known examples of these 'ways' are 'Fixed', 'Range' and 'Random'.

It should be noted that the modifiable property framework itself has no knowledge of perturbation, any specific hardware device or even any concept other than 'property'. Since the modifiable property framework lacks this knowledge, it needs the help of 'authorities' outside itself for both (1) determining whether a property is modifiable and (2) collecting the ways in which a property can be modified. In principle it means that the framework is extended by implementing and registering factories to the framework.

These authorities are known by the Modifiable Properties framework as *modifiable property factories*. The modifiable properties framework consults all factories that have registered themselves with the framework.

The modifiable property framework asks all factories independently whether or not a certain property can be modified or not and will categorise a property as modifiable if at least one of the factories determines the property to be modifiable.

Then the modifiable property framework asks all factories who categorised the property as modifiable to produce all the ways in which the span of property values can be modified that it knows of, and aggregates them.

A module in the perturbation 2 framework gathers all properties that it has available to itself, and presents these properties to the modifiable property framework. The output of the modifiable property framework is used to present the user with a list of modifiable properties as well as a method to define the span of property values for every modifiable property.

8.7.2.4 Overview of Modifiable Properties Framework

The term framework is used to describe the set of classes implementing the logic conceptualised in the previous section. This framework is generally accessed through a single class known as `com.riscure.properties.modifiable.beans.ModifiablePropertiesContainerUtils`. The framework is put to action by either giving it a bean or a property as an input.

As mentioned in the previous section, the framework is oblivious to any concept other than properties and modifiable properties. More specifically, the Modifiable Properties framework only has knowledge of the BeanProperty interface, defined by and used in Inspector. This interface unifies and simplifies several concepts regarding reflection and annotations, and also allow for the implementation of programmatic annotations.

Practical considerations

For the sake of simplicity, the conceptual description given in Section 8.7.2.3, “Modifiable Properties” ignores the fact that a property itself might be a container of other properties (known as a *complex property*, for example a property of a JavaBean or collection type). To account for this fact, the modifiable property framework needs to interact with two kinds of factories (‘authorities’):

1. A factory that, given a modifiable property as an input, will produce a list of modification methods that belong to that property.

The interface representing this kind of factory is known in the code as `com.riscure.properties.modifiable.modifiers.ModifiablePropertyFactory`.

2. A factory that, given a modifiable property, will produce, per sub-property of the input property, a list of modification methods belonging to that sub-property.

The interface representing this kind of factory is known as `com.riscure.properties.modifiable.beans.ModifiablePropertyBeanFactory`.

In order to implement recursion over JavaBean properties, the framework uses an implementation of `ModifiablePropertyBeanFactory` (known as `DefaultModifiablePropertyBeanFactory`).

Another issue that was left unmentioned in Section 8.7.2.3, “Modifiable Properties” is that when a *change event* is fired for a modifiable property, the number of sub-properties that that property might have can change and moreover the modification methods belonging to that property (or sub-properties of that property) might change as well.

The framework accounts for this by attempting to register itself as a property change listener for an input (modifiable) property and updating the output for that property when a property change notification is received.

The output provided by the Modifiable Properties framework is immediately displayable by Inspector's *automatic user interface generator*. It does so by providing its output in the form of `com.riscure.properties.modifiable.beans.ModifiablePropertiesBean` object, which is itself a JavaBean.

Backtracking

Iteration over the multi-dimensional space defined by the set of modification methods associated to distinct modifiable properties is best implemented as a backtracking algorithm. The framework provides an implementation of that backtracking algorithm in the form of an iterator. This iterator can be obtained via the `com.riscure.properties.modifiable.modifiers.ModifiablePropertyUtils` class.

8.7.2.5 Perturbation 2 framework specifics

Since the Modifiable Properties framework has no knowledge of concepts like hardware or perturbation, all entities involved with perturbation need to define among themselves a common interface that defines properties as modifiable or not. In this section, several elements of that interface are highlighted.

Core implementation

All modules in the perturbation 2 framework derive from `GenericPerturbation`. This class implements all details related to perturbation, the most important of which to this discussion is allowing the user to configure the perturbation search space (i.e. configuration of the modifiable properties).

`GenericPerturbation` is implemented in such a way that subclasses only need to specify the measurement setups (`com.riscure.signalanalysis.acquisition.MeasurementSetup`) and type of target (`com.riscure.signalanalysis.acquisition.Target`) that the module will use. Both interfaces work according to the settings bean paradigm. `GenericPerturbation` attempts to find modifiable properties these settings beans.

`GenericPerturbation` expects that the JavaBean obtained from the `getSettings()` method of either `Target` or `MeasurementSetup` marks the modifiable properties with the `@Perturbation` annotation. Moreover, if a nested property should be considered a perturbation property, the 'property path' to that nested property should be annotated with `@Perturbation` as well.

`GenericPerturbation` aggregates all properties marked with `@Perturbation` and presents these properties to the Modifiable Properties framework.

Most modifiable properties are can be modified according to all of the 'Fixed', 'Range' and 'Random' modifiers. A generic implementation of these modifiers is provided in the fault injection packages of Inspector. This is done by implementing the `ModifiablePropertyFactory` interface and allowing those implementations to be registered at the Modifiable Properties framework by Inspector's service scanner.

These implementations, `perturbation2.property.numbers.factories.{IntegerModifiablePropertyFactory, DecimalModifiablePropertyFactory}` assume that properties that are modifiable and can be modified according to the 'Fixed', 'Range' and 'Random' modifiers are annotated according to Table 8.8, "Required annotations".

Table 8.8. Annotations required by 'Fixed', 'Range' and 'Random' generic implementations

| Integer number properties | Decimal number properties |
|---------------------------|---------------------------|
| @Perturbation | @Perturbation |
| @DisplayName | @DisplayName |
| @Min | @DecimalMin |
| @Step | @DecimalStep |
| @Max | @DecimalMax |

Further extensions

Several use-case of the perturbation 2 framework demand more complex implementations of modifiable properties. The most common of these scenarios is the situation in which a list property contains several objects (of the same type), each object in turn containing (potentially) modifiable properties.

The framework implementation of handling complex properties, described in the paragraph on complex properties [194], does not recurse over objects present in a list property. Therefore a custom ModifiablePropertyBeanFactory must be implemented to deal with these properties.

A good example implementation of the above, is `GlitchPatternModifiableBeanPropertyFactory`. This factory handles properties of type `List<GlitchPatternSettings>`. A `GlitchPatternSettings` object has two properties that are modifiable within the context of perturbation 2, namely `glitchOffset` and `glitchLength`. The factory will therefore find twice as much properties as there are elements in the list provided to it.

`GlitchPatternModifiableBeanPropertyFactory` does no difficult work in essence. For every object present in the list, it obtains the `BeanProperty` representations of the modifiable properties, and then feeds these `BeanProperty` representations back into the Modifiable Properties framework, aggregating the results along the way. It does however, needs to perform one trick, described below.

Continuing from the previous section, Inspector's automatic user interface generator (indirectly) uses the value of the `@DisplayName` annotation of a modifiable property in order to name the property. However, since annotations are compile-time constants, the `DisplayName` annotation of all "glitchLength" properties is the same. When this is presented to the user, this will be confusing to him.

It is therefore necessary to override the value of the `@DisplayName` annotation so that the name will include the index of the `GlitchPatternSettings` object in the list from which it was obtained. In order to do that, the `perturbation2.property.BeanPropertyExtender` class is used. This class, which implements the `BeanProperty` interface, is constructed from a `BeanProperty` and then provides methods to override annotations or set annotations that were not yet set.

8.7.2.6 Inspector's implementation of modifiable properties

Up to this point we have provided a rather theoretical description of the Modifiable Properties framework. It is important to note that the user has access to (most of) the classes described

above. Indeed, it is worthwhile to study the Inspector implementation and then experiment a bit by changing the existing code.

We will walk through the implementation of the "Range" modifier to explain the concepts concretely. The user is invited to see the Inspector implementation himself with the "File" -> "Open System Module" menu option. We start in the `perturbation2.Perturbation` class. This module has three `MeasurementSetup` components:

```
public class Perturbation extends GenericPerturbation implements ModuleInterface {
    ...
    @Override
    protected List<MeasurementSetup> createMeasurementSetups() throws IOException {
        List<MeasurementSetup> result = new ArrayList<MeasurementSetup>();
        result.add(new MultiScopeSetup(1));
        result.add(new GlitchSetup(false));
        result.add(new ICWavesSetup(false));
        return result;
    }
    ...
}
```

In class `perturbation2.measurementsetup.GlitchSetup` the properties are stored in a `GlitchSetupSettings` object:

```
public class GlitchSetup extends AbstractDefaultGlitchSetup implements
    MeasurementSetup {
    ...
    private GlitchSetupSettings settings;

    public GlitchSetup(boolean raw) throws IOException {
        super(raw);
        GlitchSetupSettings vcgss = new GlitchSetupSettings();
        init(vcgss);
    }
    ...
}
```

In the class `perturbation2.measurementsetup.GlitchSetupSettings` the actual perturbation parameters are listed. See Example 8.1, "The perturbation parameter "Glitch voltage" in `GlitchSetupSettings`" for a snippet of this class. The user can add variables in this file and save it as e.g. "MyGlitchSetupSettings.java". If he then also adjusts `GlitchSetup` to use this new class and saves it as e.g. "MyGlitchSetup.java", and then replaces the use of `GlitchSetup` in `Perturbation` with `MyGlitchSetup` (save as "MyPerturbation.java") he will have his own custom module.¹

The framework automatically scans all the bean properties and when it finds a property with the `@Perturbation` annotation and (in this case) the `@DecimalMin`, `@DecimalMax`, `@DecimalStep`, and `@DisplayName` annotations (these five were also mentioned in Table 8.8, "Required annotations") it will generate three `ModifiablePropertyStrategy` objects for Fixed, Random, and Range. See the `perturbation2.property.numbers.factories.DecimalModifiablePropertyFactory` class for the actual mechanism. The `match()` method tests whether a `BeanProperty` can

¹Compiling a non-module class might trigger the warning "Target is neither a module nor a service compilation completed, file cannot be loaded as Inspector module". This message can be safely ignored.

be handled by this factory. This is the case when the five annotations mentioned above are present:

```
public class DecimalModifiablePropertyFactory implements ModifiablePropertyFactory {
    ...
    public Match match(BeanProperty beanProperty) {
        Match match = Match.NONE;
        if(beanProperty != null) {
            boolean hasRequiredAnnotations = (
                (beanProperty.getAnnotation(Perturbation.class) != null) &&
                (beanProperty.getAnnotation(DecimalMin.class) != null) &&
                (beanProperty.getAnnotation(DecimalMax.class) != null) &&
                (beanProperty.getAnnotation(DecimalStep.class) != null) &&
                (beanProperty.getAnnotation(DisplayName.class) != null)
            );
            boolean typeIsDecimal = TypeUtils.isDecimal(beanProperty.getType());
            if(hasRequiredAnnotations && typeIsDecimal) {
                match = Match.DEFAULT;
            }
        }
        return match;
    }
}
```

The method `getModifiableProperties()` first does a doublecheck on the matching status of a property and this factory. In the following line it creates a list with the three strategy implementations: fixed, random, and range.

```
public class DecimalModifiablePropertyFactory implements ModifiablePropertyFactory {
    ...
    public ModifiablePropertyContainer getModifiableProperties(BeanProperty
        beanProperty) {
        if(match(beanProperty) != Match.DEFAULT) {
            throw new IllegalArgumentException(String.format("Property '%s' cannot be used
by this factory", beanProperty));
        }

        List<ModifiableProperty> modifiableProperties = Arrays.asList(
            (ModifiableProperty) new ModifiablePropertyImpl(new
SimpleDecimalModifiablePropertyStrategy(beanProperty)),
            (ModifiableProperty) new ModifiablePropertyImpl(new
XkcdRandomDecimalModifiablePropertyStrategy(beanProperty)),
            (ModifiableProperty) new ModifiablePropertyImpl(new
RangeDecimalModifiablePropertyStrategy(beanProperty))
        );
        ...
        return new ModifiablePropertyContainer(modifiableProperties,
            persistanceProperties);
    }
}
```

The standard implementations of the `ModifiablePropertyStrategy` interface can be found in the `perturbation2\property\numbers\properties\strategies` directory. One of these standard strategies is the `RangeDecimalModifiablePropertyStrategy` class. It's `modifier()` method will produce a `PropertyModifier` object:

```
public class RangeDecimalModifiablePropertyStrategy extends
XkcdRandomDecimalModifiablePropertyStrategy {
    ...
}
```

```

public PropertyModifier modifier() {
    BigDecimal min = ((LowHigh<BigDecimal>)this.getValue()).getLow();
    BigDecimal max = ((LowHigh<BigDecimal>)this.getValue()).getHigh();
    BigDecimal step = (BigDecimal) this.getStep();
    return super.getPropertyModifier(min, step, min, max);
}
...
}

```

This implementation of the 'range' strategy relies on the more generic `BigDecimalNumberStrategy`, which is handled by the superclass `XkcdRandomDecimalModifiablePropertyStrategy`. The superclass method `getPropertyModifier(min, step, min, max)` will return a new `GenericPropertyModifier<BigDecimal>` instance. This object's `modify()` method will be called by the framework. When a user decides to implement his own `PropertyModifier` he can insert his code here. For example, the following modification of the `modifier()` method will result in the same list of range values, but as a Van der Corput sequence [http://en.wikipedia.org/wiki/Van_der_Corput_sequence].

```

public PropertyModifier modifier() {
    final BigDecimal min = ((LowHigh<BigDecimal>)this.getValue()).getLow();
    final BigDecimal max = ((LowHigh<BigDecimal>)this.getValue()).getHigh();
    final BigDecimal step = (BigDecimal) this.getStep();

    return new PropertyModifier() {
        BigDecimal currentValue = min;
        BigDecimal stepValue = max.subtract(min).multiply(new BigDecimal(2));
        @Override
        public boolean canModify() {
            return stepValue.compareTo(step) >= 0;
        }

        @Override
        public BigDecimal modify() {
            if (canModify()) {
                currentValue = currentValue.add(stepValue);
                if (currentValue.compareTo(max) > 0) {
                    stepValue = stepValue.divide(new BigDecimal(2));
                    currentValue = min.add(stepValue.divide(new BigDecimal(2)));
                }
            }
            getBeanProperty().set(currentValue);
            return currentValue;
        }
    };
}

```

8.8 Developing targets

In Section 8.7, “Developing measurement setups” it was mentioned that *measurement setups* are an abstraction layer between the `acquisition2` and `perturbation2` frameworks on the one hand, and device drivers on the other hand. In similar fashion, *targets* are an abstraction layer between the frameworks and the device communicating with the *target of evaluation* (ToE). Details about what device to use to communicate with the ToE and what messages to send to the ToE are hidden from the frameworks. The frameworks are offered methods to start communicating with the ToE and to obtain the log of that communication instead. The interaction between the frameworks and the target are the subject of this section.



Note

This section does not describe the development of protocols, since protocols are specific to certain implementations of the target abstraction layer. For more information on protocols, refer to Section 9.8, “How do I implement an acquisition2/perturbation2 application protocol?” and Section 9.9, “How do I implement an acquisition2/perturbation2 raw protocol?”.

For an example of an implementation of the target abstraction layer, please see `acquisition2.target.RawTarget`.

8.8.1 Target life cycle

The life cycle of a target closely follows the life cycle of a measurement setup and this section is subdivided according to those life cycles. Targets are represented by the `com.riscure.signalanalysis.acquisition.Target` interface. Each section discusses the methods called on that interface during that particular life cycle.

8.8.1.1 Loading

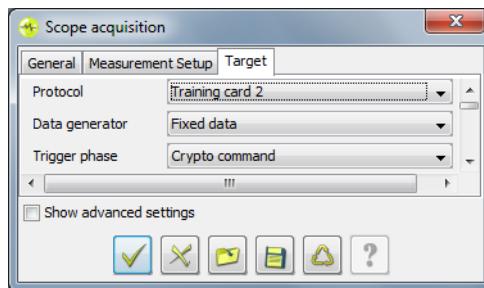
In the acquisition2 and perturbation2 frameworks, modules indicate which specific target should be loaded when the module is started. `acquisition2` and `perturbation2` modules should override the abstract `createTarget()` method provided by `acquisition2.GenericAcquisition`.

The loading process is practically the same as for measurement setups: the framework will attempt to restore persisted user settings by calling `Target.setSettingsBean(Object)`. The only difference between targets and measurement setups in this life cycle is that targets do not have an ID because there is always only one target in the `acquisition2` and `perturbation2` frameworks. For more details, see Section 8.7.1.1, “Loading”.

8.8.1.2 Display and user configuration

The only difference during this life cycle between targets and measurement setups is that targets are not named in the `acquisition2` and `perturbation2` frameworks, and therefore the `Target` interface does not have a `getName()` method. Targets are displayed under the tab entitled ‘Target’ in the module that loads them, as can be seen in Figure 8.16, “The `TriggeredProtocolTarget` implementation of the `Target` interface selected in `ScopeAcquisition`”.

Figure 8.16. The `TriggeredProtocolTarget` implementation of the `Target` interface selected in `ScopeAcquisition`



For more details, see Section 8.7.1.2, “Display and user configuration”.

8.8.1.3 Measurement run



Note

For a better understanding of this section, it is recommended to first read Section 8.7.1.3, “Measurement run”.

This life cycle is divided into two distinct stages, namely *setup* and *loop* for clarity. The contents of this section are summarized in Figure 8.17, “Sequence diagram of the interaction between the framework, target and measurement setup”.

Setup

When the user has finished configuring the module, and when the user settings have passed validation, the module will be run. At this time, the acquisition2 and perturbation2 frameworks will set the (physical) measurement up.

The framework's first action is to attach an *I/O listener* to the target. It does so by calling the `Target.setIOListener(IOListener)` method. The target is expected to pass all data sent to and received from the ToE to the frameworks via the `IOListener.{sending(CommandData), received(CommandResponse)}` methods respectively. The framework uses this information to give the user insight in and access to the actual communication with the ToE.

The framework will first initialize all measurement setups by calling `MeasurementSetup.init()`. After having done this, it is able to create a reference to all measurement setups and it passes this reference to the target. The primary reason for passing a reference to the measurement setup to the target is that this enables the target to arm and trigger all measurement devices that were configured by the user. The framework performs this step by creating an object of type `com.riscure.signalanalysis.MeasurementReference` and then calling `Target.init(MeasurementReference)`. The `MeasurementReference` object has `arm()` and `trigger()` methods for arming and, if necessary, triggering the measurement setup respectively, as well as methods to obtain the delay between the trigger and the start of the measurement and the length of the measurement.

The call to `init(MeasurementReference)` concludes the setup stage of this life cycle.

Loop

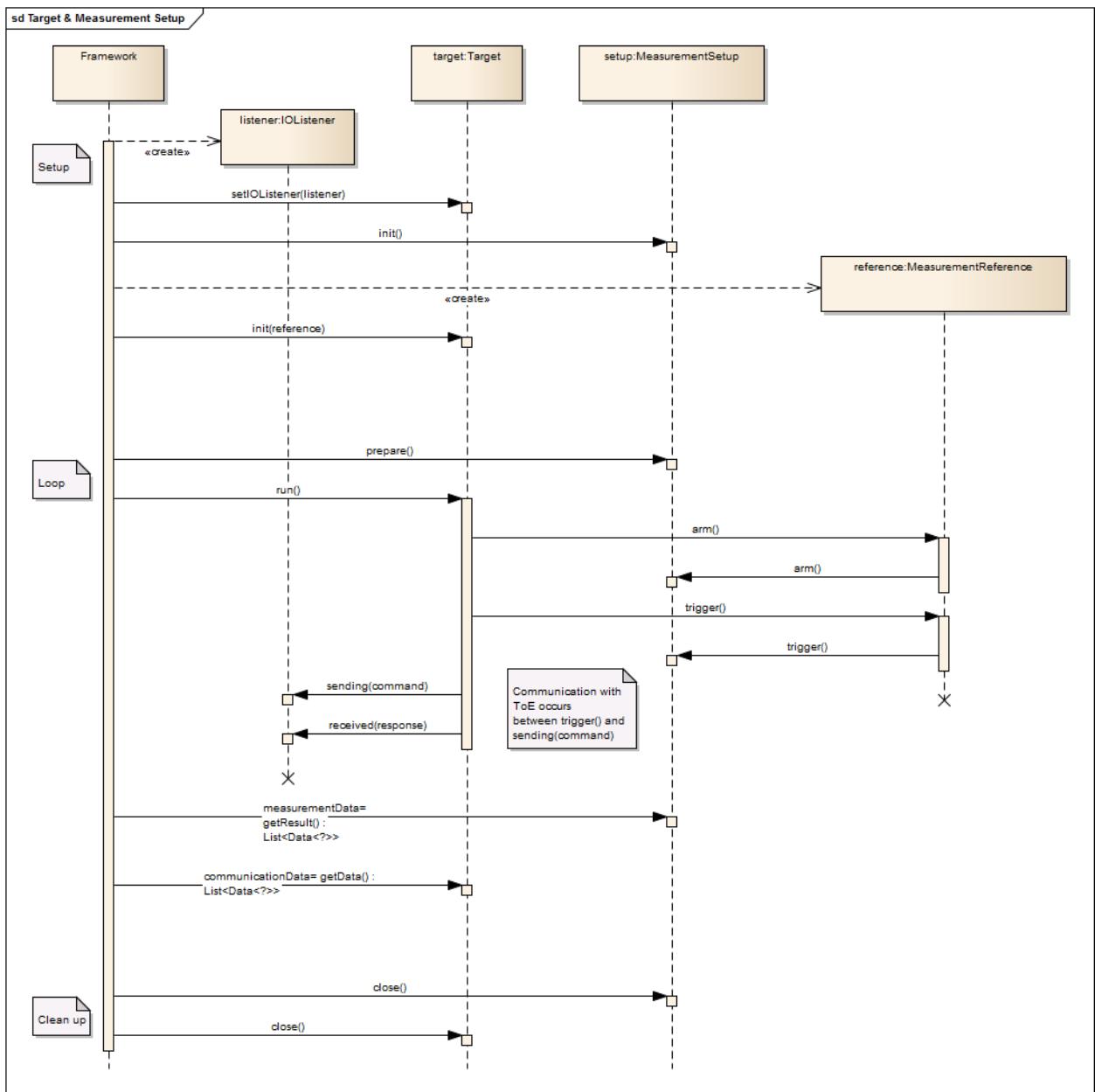
In the loop stage, the physical measurement begins. The framework first asks all measurement setups to prepare for measurement, which in practice means that new settings should be pushed to the measurement devices. The framework does this by calling `MeasurementSetup.prepare()`. The framework then asks the target to start communicating with the ToE, which it does by calling `Target.run()`.

It is the target's responsibility to make sure that all measurement devices are armed, and if necessary triggered, before it starts communicating with the ToE. Furthermore, communication to and from the ToE should be reported to the I/O listener received during the setup stage. This is detailed in Figure 8.17, “Sequence diagram of the interaction between the framework, target and measurement setup”.

When the target has finished its communication with the ToE, the framework requests measurement data from the measurement setups and communication data from the target, by calling `MeasurementSetup.getResult()` and `Target.getData()` respectively. Both methods should return their data in the form of a `List<Data<?>>` object. One final consideration is that

Target.getData() should also include a verdict (the outcome of a perturbation attempt) if the user was performing perturbation instead of acquisition.

Figure 8.17. Sequence diagram of the interaction between the framework, target and measurement setup



8.8.1.4 Closing

The only difference between targets and measurement setups during this life cycle is that Target.close() is called for targets and that MeasurementSetup.close() is called for measurement setups. See Section 8.7.1.4, “Closing”.

8.9 Developing target managers

A target manager is an external plugin implementing the TargetManagerFactory [../javadoc/com/riscure/signalanalysis/targetmanager/TargetManagerFactory.html] interface. This interface will in turn create a TargetManager [../javadoc/com/riscure/signalanalysis/targetmanager/TargetManager.html], which will be used by the user interface to allow the user to manipulate the specified target.

The target manager is provided with a reference to a device through which it is able to communicate with the ToE.

8.10 Developing transport protocols

Transport protocols represent the lower lowels of the ISO protocol stack. A transport protocol is provided with a raw I/O device in order to create a full I/O device using the Protocol device.

Transport protocols are external plug-ins implementing the TransportProtocolFactory [../javadoc/com/riscure/protocol/TransportProtocolFactory.html] interface. This factory will create a new TransportProtocol [../javadoc/com/riscure/protocol/TransportProtocol.html] based on a user provided source, destination, and protocol properties. A transport protocol can be opened resulting in a TransportSession [../javadoc/com/riscure/protocol/TransportSession.html]. The session only exposes the single method command, which is identifcal to the command method used by IODevice [../javadoc/com/riscure/hardware/io/IODevice.html].

8.11 Developing custom user interfaces

Section 8.4.1.1, “Properties” describes the automatic generation of an user interface based on the properties in a JavaBean™. The automatic user generator may not be flexible enough to facilitate all possible ways of displaying properties. Therefore it is possible to overwrite the automatic generator with a custom developed user interface for a specific bean.

In order to implement a custom panel extend the class AbstractBeanPanelFactory and implements the *match* and *createPanel* method. The factory class must have a public default constructor. The resulting class should be placed in a Java Archive with a resource file called *org.osgi.framework.BundleActivator* in *META-INF/services* within the JAR file containing a single line with the class name of the factory class.

The *match* method will be called by Inspector for every JavaBean. The factory must return an indication of how well it can render the provided bean class. *Annotation* means the factory can specifically render a representation for the class and its annotation, *None* indicates the factory can not render the bean class. Inspector will sort custom factories by match order and request the highest matching factory to render the bean. If the *createPanel* method throws an exception Inspector will attempt to use the next custom factory. If no factory is willing or able to render the bean the default factory will be used.

9 The Developer How-Do-It

This section provides Inspector software developers with a number of how-tos on common Inspector software development tasks, like implementing a new Inspector module.



Note

For sake of clarity the following sections are using a male pronoun when referring to a software developer. Please read 'him or her' wherever you read 'him'.

The purpose of each how-to in this section is to:

1. Provide an Inspector software developer with skeleton code to use when implementing his solution. This prevents a software developer from having to duplicate code from other (existing) code,
2. Provide an Inspector software developer with some insights into the architecture / framework within which a solution needs to be implemented. This provides a software developer with a greater insight into the Inspector software architecture which allows him to more easily grasp the impact of design and code changes.

9.1 How do I implement an AGG compatible JavaBean?

The Automatic User Interface Generator (AGG) serves the purpose of generating an (editable) view of a JavaBean, which also guards against constraint violations that the bean developer has defined in the bean by annotations or otherwise.

9.1.1 What is a JavaBean and what is a bean property?

When reading this tutorial please keep in mind that a JavaBean is any class that has the following attributes:

1. The class is a public class or public (static) inner class.
2. The class has a public default (argumentless) constructor.

Also keep in mind that a bean property in a JavaBean is anything that abides by the following rules:

1. A property can have a public accessor (getter) with the following declaration `<propertyType> get<PropertyName>` if the property type is non-boolean, or with the following declaration `boolean is<PropertyName>` if the property type is boolean.

The following table shows a few examples of this naming convention. Please bear in mind that also in JavaBeans, capitalisation is important.

Table 9.1. JavaBean accessor naming conventions

| Property name | Type | Accessor |
|---------------|---------|---|
| example | String | <code>public String getExample()</code> |
| example | boolean | <code>public boolean isExample()</code> |

| | | |
|--------------|------------|------------------------------|
| someProperty | int | public int getSomeProperty() |
| x | double | public double getX() |
| iSO | BigDecimal | public BigDecimal getISO() |

1. A property can have a public mutator (setter) with the following declaration `set<PropertyName>(<.PropertyType> <variableName>)` for any type of property. See the following table for several examples.

Table 9.2. JavaBean mutator naming conventions

| Property name | Type | Mutator |
|---------------|------------|------------------------------------|
| example | String | public void setExample(String s) |
| example | boolean | public void setExample(boolean b) |
| someProperty | int | public void setSomeProperty(int i) |
| x | double | public void setX(double d) |
| iSO | BigDecimal | public BigDecimal getISO() |

1. If a property is implemented using a backing field, which is not a must, then there are two scenarios to consider

In the first case the backing field will be recognised as part of the property. This has the following effects:

1. The order of the backing field declarations determines the order in which fields will be displayed in an AGG generated view of a JavaBean.
2. A property can always be annotated by annotating the accessor, but now it becomes possible to annotate the property by annotating the backing field as well.

Table 9.3. JavaBean backing field naming conventions

| Property name | Type | Backing field |
|---------------|------------|--------------------------|
| example | String | private String example; |
| example | boolean | public boolean example; |
| someProperty | int | public int someProperty; |
| xx | double | public double x; |
| iSO | BigDecimal | public BigDecimal iSO; |

9.1.2 How do I obtain an (editable) user interface view for a JavaBean instance?

Through the following method call: `com.riscure.ui.bean.BeanPanelUtils.getBeanPanel(...)`. This method returns a **JPanel**, with UI views for all properties in the given input bean. All properties of the input bean which are of a JavaBean type are recursively handled by this method, in other words the AGG supports nested bean properties.

The method has 3 definitions as a means to implement default argument passing. The arguments are explained in the next table.

Table 9.4. BeanPanelUtils.getBeanPanel(...) argument overview

| Argument | Type | Meaning | Default |
|----------|------|---------|---------|
|----------|------|---------|---------|

| | | | |
|----------|------------------|--|-------|
| bean | Object | The bean to render | N/A |
| readOnly | Boolean | Whether or not the resulting JPanel should be editable | false |
| selector | PropertySelector | A filter that chooses the particular properties from the input bean which should be shown in the resulting JPanel . | null |

9.1.3 How do I implement the MVC pattern with the AGG?

The AGG implementation takes care of the View and Controller parts of the Model-View-Controller (MVC) pattern. The Model part of the MVC pattern is left up to the programmer of the bean. How this is done is explained in this section through the following example problem:

We want an end-user to provide us with the area of a square, a decimal number. We want the user interface to be flexible enough to provide either the length of an edge of the square so that we can compute the area ourselves, or the area of the square directly.

Since two fields need to be edited, we create a bean containing the properties **edgeLength** and **area**, shown in the next Code Listing.

Example 9.1. The initial bean class that does not yet implement the Model part of the MVC pattern

```
public class AreaMvcExample {  
    public AreaMvcExample() {  
        this.setArea(4.0d);  
    }  
  
    @Min(0)  
    private double edgeLength;  
    @Min(0)  
    private double area;  
  
    public double getEdgeLength() {  
        return this.edgeLength;  
    }  
    public void setEdgeLength(double edgeLength) {  
        this.edgeLength = edgeLength;  
    }  
    public double getArea() {  
        return this.area;  
    }  
    public void setArea(double area) {  
        this.area = area;  
    }  
}
```

The following step is to convert **AreaMvcExample** into an observable model.

9.1.3.1 Property change notifications

In order to convert **AreaMvcExample** into an observable model, it needs to be able to fire property change notifications when the value of a property changes.

In Java, there is no interface that can be implemented by a class that will indicate that the implementing class fires property change notifications. Instead, it is a convention to implement

a number of methods which property change event handlers need to call by reflection to register themselves.

the next code shows the additions that need to be made to **AreaMvcExample** in order to enable firing property change notifications.

Example 9.2. Enabling property change notifications as part of the Model implementation of the MVC pattern

```
import java.beans.PropertyChangeSupport;
import java.beans.PropertyChangeListener;
public class AreaMvcExample {
    public AreaMvcExample() {
        this.setArea(4.0d);
    }

    @Min(0) private double edgeLength;
    @Min(0) private double area;

    public double getEdgeLength() {
        return this.edgeLength;
    }
    public void setEdgeLength(double edgeLength) {
        this.edgeLength = edgeLength;
    }
    public double getArea() {
        return this.area;
    }
    public void setArea(double area) {
        this.area = area;
    }
    /*
     * Property change support */
    private PropertyChangeSupport pcs = new PropertyChangeSupport(this);

    public void addPropertyChangeListener(PropertyChangeListener listener) {
        this.pcs.addPropertyChangeListener(listener);
    }
    public void addPropertyChangeListener( String propertyName, PropertyChangeListener listener ) {
        this.pcs.addPropertyChangeListener(propertyName, listener);
    }
    public void removePropertyChangeListener(PropertyChangeListener listener) {
        this.pcs.removePropertyChangeListener(listener);
    }
    public void removePropertyChangeListener( String propertyName,
    PropertyChangeListener listener ) {
        this.pcs.removePropertyChangeListener(propertyName, listener);
    }
}
```

The final step is to fire property change events to notify the View and Controller of changes occurring in the model.

9.1.3.2 Firing property change events

In order to update View and Controller of the changes in our model, we need to fire property change events as soon as a property changes its value. In our example, changing edgeLength also changes area and vice versa. In order to centralise this behaviour, a helper method

`setInternal()` is introduced which is called by both `setEdgeLength()` and `setArea()`. This helper method not only sets the new values of `edgeLength` and `area`, but also fires the property change events. Notice that firing a property change event in general requires both the old and the new value as event arguments. This completes the Model part of the MVC pattern :

```
public class AreaMvcExample {  
    public AreaMvcExample() {  
        this.setArea(4.0d);  
    }  
  
    @Min(0) private double edgeLength;  
    @Min(0) private double area;  
  
    public double getEdgeLength() {  
        return this.edgeLength;  
    }  
    public void setEdgeLength(double edgeLength) {  
        this.setInternal( this.getEdgeLength(), edgeLength, this.getArea(),  
        Math.pow(edgeLength, 2.0d) );  
    }  
    public double getArea() {  
        return this.area;  
    }  
    public void setArea(double area) {  
        this.setInternal( this.getEdgeLength(), Math.sqrt(area), this.getArea(),  
        area );  
    }  
    private void setInternal( double oldEdgeLength, double newEdgeLength, double  
    oldArea, double newArea ) {  
        this.edgeLength = newEdgeLength;  
        this.area = newArea;  
        pcs.firePropertyChange("edgeLength", oldEdgeLength, newEdgeLength);  
        pcs.firePropertyChange("area", oldArea, newArea);  
    }  
  
    /*  
     * Property change support */  
    private PropertyChangeSupport pcs = new PropertyChangeSupport(this);  
  
    public void addPropertyChangeListener(PropertyChangeListener listener) {  
        this.pcs.addPropertyChangeListener(listener);  
    }  
    public void addPropertyChangeListener( String propertyName, PropertyChangeListener  
    listener ) {  
        this.pcs.addPropertyChangeListener(propertyName, listener);  
    }  
    public void removePropertyChangeListener(PropertyChangeListener listener) {  
        this.pcs.removePropertyChangeListener(listener);  
    }  
    public void removePropertyChangeListener( String propertyName,  
    PropertyChangeListener listener ) {  
        this.pcs.removePropertyChangeListener(propertyName, listener);  
    }  
}
```

9.1.3.3 Some notes on firing property change events.

The Code Listings in this section assume that property change notifications are implemented as shown in the code of Section 9.1.3.2, “Firing property change events”

That example is actually a non-standard way of firing a property change event. The next code example shows the standard pattern of firing property change events for a property with the placeholders <propertyType> for the type of the property, <propertyName> for property name and <PropertyName> for the property name in the accessor and mutator method definitions.

Example 9.3. The standard fire property change event pattern

```
private <propertyType> <propertyName>;
public <propertyType> get<PropertyName>() {
    return this.<propertyName>;
}
public void set<PropertyName>(<propertyType> <propertyName>) {
    <propertyType> old = this.<propertyName>;
    this.<propertyName> = <propertyName>;
    pcs.firePropertyChange("<PropertyName>", old, this.<propertyName>);
}
```

There are some situations where a property change event needs to be fired but the old and new values of a property are unknown or hard to retrieve. Examples of these situations will appear in upcoming sections. In those events, null can be passed as event arguments for both the old and new value:

Example 9.4. Firing a property change event for a property when the old and new values are unknown

```
pcs.firePropertyChange("<PropertyName>", null, null);
```

The next line of code will suffice to notify property change event handlers when an arbitrary set of properties change in an object. Unfortunately, this is at present not supported by the AGG.

Example 9.5. Firing a property change event when multiple properties changed in the containing object

```
pcs.firePropertyChange(null, null, null);
```

9.1.4 How do I set constraints on a (single) bean property?

There are two methods of doing this:

1. By annotating either the backing field or accessor of the property. This approach is the easier and more preferred approach amongst the two methods.
2. Through implementing your own validation logic in the mutator of a property and throwing an exception when an invalid value was given as input to the mutator. This approach is more complex but is required when for example the constraints of the property are dynamic.

9.1.4.1 Annotations

Note



This approach is based on JSR 303.

There are different annotations that are understood by the AGG when used to annotate a bean property. Of these, only the annotations shown and explained in Table 9.5, “An overview of annotations used by the AGG for the purpose of validation” are used for the purpose of defining constraints. The annotations in this table shows most but not all annotations from JSR 303, only the most commonly used ones. All annotations from JSR 303 however, can be used to define constraints. Some of the constraints from Table are custom constraints defined in the Inspector API. The annotations from JSR 303 all have a message attribute that can be used to pass a custom error message to the end-user.

The AGG takes care never to set a value on a property that is outside the range of values defined by the annotations used on that property.. When an end-user tries to input a value that lies outside of this range, a notification will be shown to that the inputted value is invalid and, if possible, what the valid range of values is.

A final consideration is the default value of a property. It is assumed that the default property is any of the valid values for that property. The AGG does not perform any check to see if this is indeed the case.

Table 9.5. An overview of annotations used by the AGG for the purpose of validation

| Annotation | Applicable types | Purpose | Remarks |
|---------------------------|---|--|--|
| <code>@Min</code> | All integral number types (<code>byte</code> , <code>short</code> , <code>int</code> , <code>long</code> , <code>BigInteger</code>) | Defines a minimum value | <code>@Min</code> takes as argument a <code>long</code> and is in this sense inconsistent with <code>@DecimalMin</code> . |
| <code>@Max</code> | Same as <code>@Min</code> | Defines a maximum value | Same as <code>@Min</code> |
| <code>@Step</code> | Same as <code>@Min</code> | Defines a step size | Usually used in conjunction with <code>@Min</code> and <code>@Max</code> and implicitly defines a range of possible values for the property. For example, a property annotated with <code>@Min(3)</code> and <code>@Step(2)</code> cannot take any value below 3 and not any value that is even (4, 6, etc.). |
| <code>@DecimalMin</code> | All decimal number types (<code>float</code> , <code>double</code> , <code>BigDecimal</code>) | Same as <code>@Min</code> , but for decimal number types. | <code>@DecimalMin</code> takes as argument the <code>BigDecimal</code> representation of a floating point number and is in this sense inconsistent with <code>@Min</code> . |
| <code>@DecimalMax</code> | Same as <code>@DecimalMin</code> | Same as <code>@Max</code> , but for decimal number types. | Same as <code>@DecimalMin</code> |
| <code>@DecimalStep</code> | Same as <code>@DecimalMin</code> | Same as <code>@Step</code> , but for decimal number types. | Same as <code>@Step</code> , but for decimal number types. |
| <code>@LowMin</code> | <code>LowHigh<></code> where the parametrized type is an integral number type, for example <code>LowHigh<Short></code> | Same as <code>@Min</code> , but for <code>LowHigh<></code> of integral number types. | As <code>@Min</code> , takes as argument a <code>long</code> . Is used to define the boundary of a <code>LowHigh</code> , which is rendered as a range slider. |
| <code>@HighMax</code> | Same as <code>@LowMin</code> | Same as <code>@Max</code> , but for <code>LowHigh<></code> of integral number types. | Same as <code>@LowMin</code> |

| | | | |
|------------------------------|---|---|---|
| <code>@DecimalLowMin</code> | <code>LowHigh<></code> where the parametrized type is a decimal number type, for example <code>LowHigh<BigDecimal></code> | Same as <code>@Min</code> , but for <code>LowHigh<></code> of decimal number types. | As <code>@DecimalMin</code> , takes as argument a String. Is used to define the boundary of a <code>LowHigh</code> , which is rendered as a range slider. |
| <code>@DecimalHighMax</code> | Same as <code>@DecimalLowMin</code> | Same as <code>@Max</code> , but for <code>LowHigh<></code> of decimal number types. | Same as <code>@DecimalLowMin</code> . |
| <code>@Length</code> | <code>String</code> | Defines the length of a <code>String</code> . | Two values need to be set in this annotation: min and max. The type of min and max is int. |
| <code>@Regex</code> | <code>String</code> | Defines a regular expression that the property value should match. | |
| <code>@Email</code> | <code>String</code> | Defines that only well-formed email addresses are accepted. | |
| <code>@AssertFalse</code> | <code>boolean</code> | Defines that the property should be false. | |
| <code>@AssertTrue</code> | Same as <code>@AssertFalse</code> | Defines that the property should be true. | |
| <code>@Past</code> | <code>java.util.{Date,Calendar}</code> | Defines that a date should be in the past. | |
| <code>@Future</code> | Same as <code>@Past</code> | Defines that a date should be in the future. | . |

The next example shows the most common way of defining a property named `annotationExample` of type `int`, which can take the values 3, 5, 7 and 9.

Example 9.6. Bean property constraints through annotations, with a custom error message

```

@Min(value=3, message="A custom message: Value cannot be less than 3")
@Max(9)
@Step(2)
private int annotationExample = 5;

public int getAnnotationExample() {
    return this.annotationExample;
}
public void setAnnotationExample(int annotationExample) {
    this.annotationExample = annotationExample;
}

```

9.1.4.2 Custom mutator logic

In this method approach you determine in the mutator logic whether or not a value is valid. When the input is invalid, an exception should be thrown. In this method it is up to you to determine whether you allow the bean property to be put in an invalid state by an invalid value or not.

The natural consequence of this approach is that the AGG cannot determine beforehand what the valid range of inputs is. However, the AGG will display to the end-user the message of the exception that was thrown by the mutator logic.

An example that mimics the behaviour of `@AssertTrue` is given in the next example.

Example 9.7. Bean property constraints through custom validation logic

```
private boolean invalidExample = true;

public boolean getInvalidExample() {
    return this.invalidExample;
}

public void setInvalidExample(boolean b) {
    if(!b) {
        throw new RuntimeException("Only true is allowed for property invalidExample");
    }
    this.invalidExample = b;
}
```

9.1.5 How do I constrain a relationship amongst bean properties?

It should be noted that the AGG never sets an invalid value on a single bean property if a range of valid values was defined for that property. However, this is not sufficient to prevent an entire bean from being in an invalid state, since the AGG does not know beforehand whether setting a single value on a property will violate a relationship constraint.

In practice it means that if your bean has relational constraints, you cannot assume that your bean is in a valid state after an end-user has edited your bean through the AGG. How to collect all constraint violations in a JavaBean instance is explained in Section 9.1.6, “How do I find all constraint violations on a JavaBean instance?”.

There are two methods of implementing a relational constraint

1. By (ab)using read-only **boolean** properties. This is the easiest, fastest but least reusable method amongst the two.
2. By implementing a custom constraint and associated constraint validator.

The sections Section 9.1.5.1, “(Ab)using read-only boolean properties” and Section 9.1.5.2, “Custom constraints and constraint validators” show possible solutions to the following question: The bean in the following code example has two **int** properties: **start** and **end**. How can the relationship between these properties be constrained to **(end – start) > 5**?

Example 9.8. A JavaBean with unconstrained properties

```
public class RelationConstraintExample {  
    @Min(2)  
    @Step(2)  
    private int start = 2;  
    @Min(4)  
    @Step(2)  
    private int end = 8;  
  
    public int getStart() {  
        return this.start;  
    }  
    public void setStart(int start) {  
        this.start = start;  
    }  
    public int getEnd() {  
        return this.end;  
    }  
    public void setEnd(int end) {  
        this.end = end;  
    }  
}
```

9.1.5.1 (Ab)using read-only boolean properties

Define a read-only **boolean** property, annotated with **@AssertTrue** as shown in the next example:

```
public class RelationConstraintExample {  
    @Min(2)  
    @Step(2)  
    private int start = 2;  
    @Min(4)  
    @Step(2)  
    private int end = 8;  
  
    public int getStart() {  
        return this.start;  
    }  
    public void setStart(int start) {  
        this.start = start;  
    }  
    public int getEnd() {  
        return this.end;  
    }  
    public void setEnd(int end) {  
        this.end = end;  
    }  
  
    @AssertTrue(message="End should be at least 5 units greater than start")  
    public boolean isValidRange() {  
        if (this.getEnd() < this.getStart()) {  
            return false;  
        }  
        int actualRange = this.getEnd() - this.getStart();  
        if (actualRange < 5) {  
            return false;  
        }  
    }  
}
```

```
        return true;
    }
}
```

9.1.5.2 Custom constraints and constraint validators

This approach is based on JSR 303.

Annotations at the property level can only be used to set constraints on a single property. In order to constrain a relationship between bean properties using annotations, the bean class holding the constrained properties needs to be annotated.

In this solution, the JavaBean is annotated with a newly defined annotation `@MinimumRangeConstraint`. The definition of this annotation contains a reference to a validator `MinimumRangeValidator`. When the bean is validated through the methods shown in Section 9.1.6, “How do I find all constraint violations on a JavaBean instance?”, this validator will be called and its result will be incorporated in the validation result.

The resulting code, including some clarifying comments, is shown next:

```
// The newly defined constraint
@MinimumRangeConstraint( minimumRange=5, message="End should be at least 5 units
greater than start")
public class RelationConstraintExample {
    @Min(2)
    @Step(2)
    private int start = 2;
    @Min(4)
    @Step(2)
    private int end = 8;

    public int getStart() {
        return this.start;
    }
    public void setStart(int start) {
        this.start = start;
    }
    public int getEnd() {
        return this.end;
    }
    public void setEnd(int end) {
        this.end = end;
    }
}

@Target( { FIELD, METHOD, TYPE })
@Retention(RUNTIME)

// @Constraint indicates that this attribute is a constraint.
// validatedBy refers to the validator class
@Constraint(validatedBy = MinimumRangeValidator.class)
public @interface MinimumRangeConstraint {

    // This constraint can be configured
    int minimumRange() default 0;
    // The following 3 attributes are 'enforced' by JSR 303
    String message() default "{com.riscure.demo.constraint.example}";
    Class<?>[] groups() default {};
}
```

```
        Class<? extends Payload>[] payload() default {};
    }

    // The validator class

    public class MinimumRangeValidator implements
        ConstraintValidator< MinimumRangeConstraint, RelationConstraintExample > {

        private MinimumRangeConstraint constraint;

        // A hook to obtain the annotation instance, which contains relevant settings

        @Override public void initialize(MinimumRangeConstraint constraintAnnotation) {
            this.constraint = constraintAnnotation;
        }

        // The actual validation method
        @Override
        public boolean isValid( RelationConstraintExample value, ConstraintValidatorContext
            context ) {
            if(value == null) {
                return true;
            }

            if(value.getEnd() < value.getStart()) {
                return false;
            }

            int actualRange = value.getEnd() - value.getStart();
            if(actualRange < this.constraint.minimumRange()) {
                return false;
            }
            return true;
        }
    }
}
```

9.1.6 How do I find all constraint violations on a JavaBean instance?

The following call `com.riscure.beans.ValidationsUtils.validate(T beanInstance)` which returns a `Set<ConstraintViolation<T>>` aggregates all (simple property and relational) constraint violations that the inputted bean instance has at that moment. If the resulting Set is empty, it means that the bean is in a valid state.

However, when a bean instance is used in conjunction with the AGG, individual properties are not put in an invalid state. It is still possible that an end-user has entered some invalid inputs in the AGG generated view of the bean instance. the next code example shows how to obtain a list of all error messages currently being displayed to the end-user.

Example 9.9. Obtaining a list of all invalid end-user input

```
Object bean = // some bean instance;  
  
JPanel beanPanel = BeanPanelUtils.getBeanPanel(bean);  
  
// Collection to hold all current error messages  
Collection<String> errorMessages;  
  
if( beanPanel instanceof FullPanel ) {  
    errorMessages = ( ( FullPanel ) beanPanel ).getCurrentErrorMessages();  
} else {  
    errorMessages = Collections.emptyList();  
}
```

9.2 How do I control the way properties are displayed by the AGG?

9.2.1 How do I change the display name of a property?

By annotating the property with the **@DisplayName** annotation. The use of this annotation is demonstrated in Section 9.2.4, “How do I control the way a number editor is being displayed?”.

9.2.2 How do I add a tag or description to a property?

By annotating the property with the **@ShortDescription** annotation. The use of this annotation is demonstrated in Section 9.2.4, “How do I control the way a number editor is being displayed?”.

9.2.3 How do I add a unit to a property?

By annotating the property with the **@Unit** annotation. The use of this annotation is demonstrated in Section 9.2.4, “How do I control the way a number editor is being displayed?”.

9.2.4 How do I control the way a number editor is being displayed?

There are various ways of displaying a field to edit a number. It can be a text field, a spinner or a slider. Each of these fields may have additional annotations that affect the behaviour of the field.

9.2.4.1 A simple text field

Declaring a property that is of a number type (**byte**, **short**, **int**, **long**, **BigInteger**, **float**, **double**, **BigDecimal**) will display the property as a simple text field. To avoid rounding problems when using decimal number properties, **BigDecimal** should be preferred over **float** and **double**.

Example 9.10. The bean displayed by the AGG in the figures below

```
public class NumberAsTextFieldExample {  
    @DisplayName("Integer text field example")  
    @ShortDescription("This integer appears as a simple text field")  
    @Unit("Hz")  
    private short integerNumber = 5;  
    @DisplayName("Decimal text field example")  
    @ShortDescription("This decimal appears as a simple text field")  
    @Unit("V")  
    private BigDecimal decimalProperty = BigDecimal.valueOf(5d);  
  
    public short getIntegerNumber() {  
        return this.integerNumber;  
    }  
    public void setIntegerNumber(short integerNumber) {  
        this.integerNumber = integerNumber;  
    }  
    public BigDecimal getDecimalProperty() {  
        return this.decimalProperty;  
    }  
    public void setDecimalProperty(BigDecimal decimalProperty) {  
        this.decimalProperty = decimalProperty;  
    }  
}
```

This code example will generate the following dialogs:

Figure 9.1. Displaying numbers as text

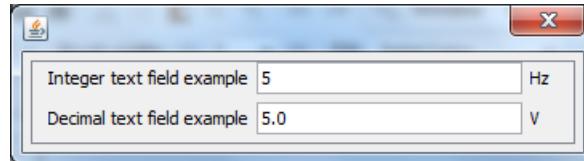
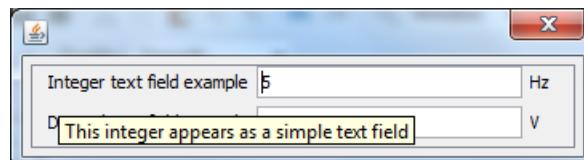


Figure 9.2. Example of a tool tip



9.2.4.2 A spinner

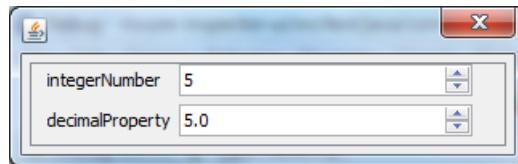
To display an integer number property as a spinner, annotate the property with **@Step**. Decimal number properties can be annotated either with **@Step** or **@DecimalStep**. The following code example shows the how to display a spinner field generated by the AGG.

The constraints **@Min**, **@Max**, **@DecimalMin** and **@DecimalMax** will be respected by the generated spinners.

Example 9.11. bean code demonstrating spinners

```
public class NumberAsSpinnerExample {  
    @Step(2)  
    private short integerNumber = 5;  
    @DecimalStep("2.1")  
    private BigDecimal decimalProperty = BigDecimal.valueOf(5d);  
  
    public short getIntegerNumber() {  
        return this.integerNumber;  
    }  
    public void setIntegerNumber(short integerNumber) {  
        this.integerNumber = integerNumber;  
    }  
    public BigDecimal getDecimalProperty() {  
        return this.decimalProperty;  
    }  
    public void setDecimalProperty(BigDecimal decimalProperty) {  
        this.decimalProperty = decimalProperty;  
    }  
}
```

Figure 9.3. Displaying numbers as spinners



9.2.4.3 A slider

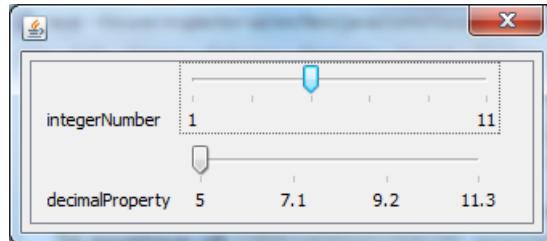
Number properties should be annotated with `@Presentation(slider=true)`. To display an integer number property, the property should be further annotated with `@Min`, `@Step` and `@Max`. Decimal number properties should be further annotated with `@DecimalMin`, `@DecimalStep` and `@DecimalMax`

The slider will by default show labels for the minimum and maximum allows values of the property. Additional labels can be added to the slider by using the `labels()` attribute of `@Presentation`. These labels are placed at the appropriate location on the slider. The following code shows how to create a slider field.

Example 9.12. Bean code demonstrating sliders

```
public class NumberAsSliderExample {  
    @Min(1)  
    @Step(2)  
    @Max(11)  
    @Presentation(slider=true)  
    private short integerNumber = 5;  
  
    @DecimalMin("5")  
    @DecimalStep("2.1")  
    @DecimalMax("11.3")  
    @Presentation(slider=true, labels={"7.1", "9.2"})  
    private BigDecimal decimalProperty = BigDecimal.valueOf(5d);  
  
    public short getIntegerNumber() {  
        return this.integerNumber;  
    }  
    public void setIntegerNumber(short integerNumber) {  
        this.integerNumber = integerNumber;  
    }  
    public BigDecimal getDecimalProperty() {  
        return this.decimalProperty;  
    }  
    public void setDecimalProperty(BigDecimal decimalProperty) {  
        this.decimalProperty = decimalProperty;  
    }  
}
```

Figure 9.4. Displaying numbers as sliders



9.2.4.4 Base 2, 8 or 16 notations

Integer number properties can be displayed to and edited by the end-user in base 2, base 8 and base 16 notations by annotating the properties with `@Presentation(base=2)`, `@Presentation(base=8)` and `@Presentation(base=16)` respectively. Not annotating the property with `@Presentation` or not setting the `base()` attribute of `@Presentation` will display the property in the default mode of base 10.

The code example in the next section describes how to apply these annotations `base2`, `base8`, `base10`, `base10Si` and `base16` to present 5 views of the single backing field `number`. The `base10Si` property is explained in the next section.

9.2.4.5 SI notation

Integer and decimal number properties are by default displayed to and editable by the end-user in SI notation. In order to disable SI notation, annotate a number property with `@Presentation(si=false)`:

```
public class NumberBaseAndSiExample {
    private int number = 3882;

    @Presentation(base=2)
    public int getBase2() {
        return this.number;
    }
    public void setBase2(int base2) {
        this.number = base2;
        this.fireChangeEvent();
    }

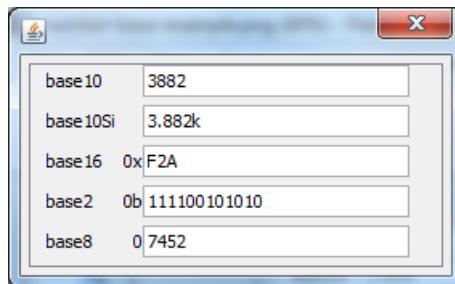
    @Presentation(base=8)
    public int getBase8() {
        return this.number;
    }
    public void setBase8(int base8) {
        this.number = base8;
        this.fireChangeEvent();
    }

    @Presentation(si=false)
    public int getBase10() {
        return this.number;
    }
    public void setBase10(int base10) {
        this.number = base10;
        this.fireChangeEvent();
    }
    public int getBase10Si() {
        return this.number;
    }
    public void setBase10Si(int base10Si) {
        this.number = base10Si;
        this.fireChangeEvent();
    }

    @Presentation(base=16)
    public int getBase16() {
        return this.number;
    }
    public void setBase16(int base16) {
        this.number = base16; this.fireChangeEvent();
    }

    private void fireChangeEvent() {
        pcs.firePropertyChange("base2", null, null);
        pcs.firePropertyChange("base8", null, null);
        pcs.firePropertyChange("base10", null, null);
        pcs.firePropertyChange("base10Si", null, null);
        pcs.firePropertyChange("base16", null, null);
        pcs.firePropertyChange("base16", null, null);
    }

    /*
     * Property change support <code omitted>
     */
}
```

Figure 9.5. Displaying numbers in different base notations

9.2.4.6 Truncated numbers

Sometimes it is useful to present numbers with a fixed precision. For instance when a module prints a series of numbers it might not be necessary to maintain the maximum precision; typically users will be interested in the most significant part of the numbers. When the **truncated** property is set to true Inspector will present the number with two decimals precision. The default value is false.

Note that this property only affects the way the values are displayed. Internally, the original precision is maintained. Also note that the truncate operation always rounds *down*.

Example 9.13. Three examples of the truncated number property

Here are three numbers displayed with different presentation properties. First we see the default SI formatting without truncating. Then the effect of adding **truncated=true** is shown. Finally, the use of the truncated non-SI presentation is demonstrated.

@Presentation():

```
123.43012k  
1.288002M  
83.2089k
```

@Presentation(truncated=true):

```
123.43k  
1.28M  
83.20k
```

@Presentation(si=false, truncated=true):

```
123430.12  
1288002.00  
83208.90
```

9.2.5 How do I display a range slider?

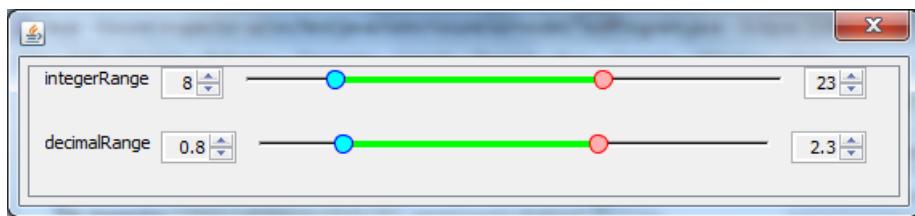
Define a **LowHigh** property. If the parameterised type is an integer number type, the property should be annotated with **@LowMin**, **@HighMax** and **@Step**. If it is annotated with a decimal

number type, it should be annotated with `@DecimalLowMin`, `@DecimalHighMax` and `@DecimalStep`.

Example 9.14. Bean code demonstrating range sliders

```
public class RangeSliderExample {  
    @LowMin(3)  
    @HighMax(33)  
    @Step(5)  
    private LowHigh<Byte> integerRange = new LowHigh<Byte>( Byte.class, (byte) 8,  
    (byte) 23 );  
  
    @DecimalLowMin("0.3")  
    @DecimalHighMax("3.3")  
    @DecimalStep("0.5")  
    private LowHigh<BigDecimal> decimalRange = new LowHigh<BigDecimal>(  
        BigDecimal.class, BigDecimal.valueOf(0.8), BigDecimal.valueOf(2.3) );  
  
    public LowHigh<Byte> getIntegerRange() {  
        return this.integerRange;  
    }  
    public void setIntegerRange(LowHigh<Byte> integerRange) {  
        this.integerRange = integerRange;  
    }  
    public LowHigh<BigDecimal> getDecimalRange() {  
        return this.decimalRange;  
    }  
    public void setDecimalRange(LowHigh<BigDecimal> decimalRange) {  
        this.decimalRange = decimalRange;  
    }  
}
```

Figure 9.6. Displaying ranges as range sliders



9.2.6 How do I display a drop down list to allow an end-user to select a value?

There are three methods of displaying drop down lists. In this section, examples of all three methods are shown and discussed.

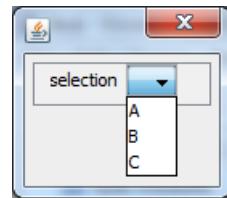
9.2.6.1 Enum

Enum properties are shown as drop down lists by default, as demonstrated in the following code. There is no way to filter what members of the `Enum` type should be included in the drop down list.

Example 9.15. Bean code demonstrating enum as drop down list

```
public class ComboBoxWithEnumExample {  
    private ComboBoxItem selection;  
  
    public ComboBoxItem getSelection() {  
        return this.selection;  
    }  
    public void setSelection(ComboBoxItem selection) {  
        this.selection = selection;  
    }  
  
    public static enum ComboBoxItem {  
        A,  
        B,  
        C  
    }  
}
```

Figure 9.7. Displaying an Enum as a drop down list.



9.2.6.2 Using the @Options annotation

An approach that the AGG provides to display a drop down list in which an arbitrary type is displayed is the usage of the **@Options** annotation. In the example that follows, the class **ComboBoxItem** will be displayed.

The **@Options** annotation is used to annotate the property that will be set by the generated drop down list. The **@Options** annotation contains the name of the property that holds all the options that the end-user can select from.

The property named **optionsSelection** is of type **ComboBoxItem**. This property is annotated with **@Options("optionsList")**. **optionsList** is a property of type **List<ComboBoxItem>**. This property is initialised to hold three items.

Whenever the user selects a value from the drop down list, a new value is added to the drop down list. The mutator of **optionsSelection**, **setOptionsSelection()**, is used to implement this dynamic behaviour. Property change events must be fired if the contents of the backing list change, otherwise the generated drop down list will transition to an erroneous state. The result of this dynamic behaviour is shown in Figure .

Notice that the **optionsList** property is annotated with **@Hidden**. This **@Hidden** annotation is discussed in section How do I always hide a property? The **@Hidden** annotation. The **@Hidden** annotation is used here because a property of **Lists** of complex types cannot be displayed by the AGG without further annotating the property. This topic is further explored in Section 9.2.9, “How do I display an arbitrary amount of nested beans?”.

Example 9.16. ComboBoxItem code used in next example

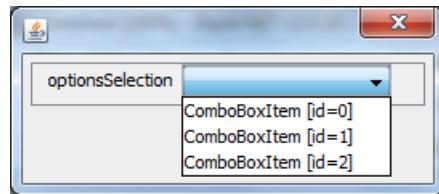
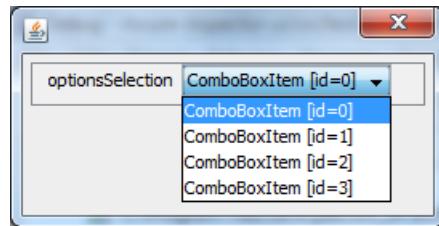
```
public class ComboBoxItem {  
  
    public final int id;  
  
    public ComboBoxItem(int id) {  
        this.id = id;  
    }  
  
    @Override public String toString() {  
        return String.format("ComboBoxItem [id=%s]", id);  
    }  
}
```

Bean example using the ComboBoxItem class:

Example 9.17. Bean code demonstrating combobox with complex objects

```
public class ComboBoxWithOptionsExample {  
    @Options("optionsList")  
    private ComboBoxItem optionsSelection;  
  
    // List of complex type cannot be displayed out of the box  
    // Therefore we hide it  
  
    @Hidden  
    private List<ComboBoxItem> optionsList = Lists.newArrayList(  
        new ComboBoxItem(0), new ComboBoxItem(1), new ComboBoxItem(2));  
  
    public ComboBoxItem getOptionsSelection() {  
        return this.optionsSelection;  
    }  
    public void setOptionsSelection(ComboBoxItem optionsSelection) {  
        this.optionsSelection = optionsSelection;  
        // Add a new item to the list  
        int currentLastItemIndex = this.getOptionsList().size() - 1;  
        ComboBoxItem currentLastItem = this.getOptionsList().get(currentLastItemIndex);  
        ComboBoxItem newLastItem = new ComboBoxItem(currentLastItem.id + 1);  
        this.getOptionsList().add(newLastItem);  
        pcs.firePropertyChange("optionsList", null, null);  
    }  
    public List<ComboBoxItem> getOptionsList() {  
        return this.optionsList;  
    }  
    public void setOptionsList(List<ComboBoxItem> optionsList) {  
        this.optionsList = optionsList;  
    }  
  
    /*  
     * Property change support <code omitted>  
     */  
  
    // Property change support code here  
}
```

Two properties, linked through the @Options annotation, are jointly displayed as a drop down list by the AGG.

Figure 9.8. Displaying a drop down list in the AGG**Figure 9.9. A dynamic update of the drop down list in the AGG when compared to Figure 9.8, “Displaying a drop down list in the AGG”**

9.2.6.3 DynamicEnum

Properties of type **DynamicEnum<T>** are displayed by the AGG as a drop down list. The **DynamicEnum** interface was defined to approximate the behaviour of a regular Java **Enum**, in the sense that an instance of **DynamicEnum** defines both a collection of set values and a (possible **null**) selection from that set.

The **DynamicEnum<T>** interface allows a programmer to register event handlers for the events in which the selected item changes or the contents of the list of possible values changes. An easy to use implementation of this interface is **com.riscure.util.enums.dynamic.impl.DynamicEnumImpl<T>** which implements **DynamicEnum<T>** and **Collection<T>**.

Example 9.18. Bean code demonstrating dynamically updated drop-down list

```
public class ComboBoxWithDynamicEnumExample {  
  
    private DynamicEnumImpl<ComboBoxItem> dynamicEnum;  
  
    public ComboBoxWithDynamicEnumExample() {  
        this.dynamicEnum = new DynamicEnumImpl<ComboBoxItem>();  
        this.dynamicEnum.addAll(Arrays.asList( new ComboBoxItem(0), new  
        ComboBoxItem(1), new ComboBoxItem(2) ));  
  
        this.dynamicEnum.addValueListener(new DynamicEnumListener<ComboBoxItem>() {  
  
            @Override  
            public void valueChanged(DynamicEnumEventArgs<ComboBoxItem> e) {  
                List<ComboBoxItem> optionsList = dynamicEnum.items();  
                ComboBoxItem currentLastItem = optionsList.get(optionsList.size() - 1);  
                ComboBoxItem newLastItem = new ComboBoxItem(currentLastItem.id + 1);  
                dynamicEnum.add(newLastItem);  
            }  
  
            @Override  
            public void itemsChanged(ItemsEventArgs<ComboBoxItem> e) { }  
        });  
    }  
  
    public DynamicEnumImpl<ComboBoxItem> getDynamicEnum() {  
        return this.dynamicEnum;  
    }  
  
    public void setDynamicEnum(DynamicEnumImpl<ComboBoxItem> dynamicEnum) {  
        this.dynamicEnum = dynamicEnum;  
    }  
}
```

Figure 9.10. Displaying an Enum as a drop down list.

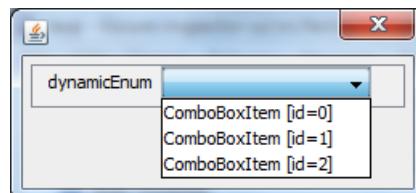
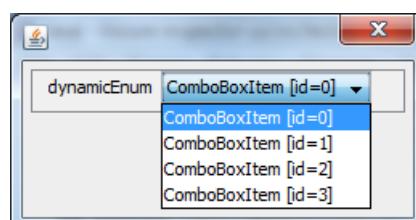


Figure 9.11. Using the DynamicEnum interface, equivalent to Figure 9.7, “Displaying an Enum as a drop down list.”



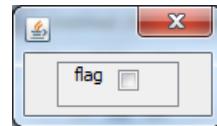
9.2.7 How do I display a check box?

boolean properties are displayed by the AGG as checkboxes, as can be seen in the following code example.

Example 9.19. Bean code demonstrating a check box

```
public class CheckBoxExample {  
    private boolean flag;  
  
    public boolean getFlag() {  
        return this.flag;  
    }  
    public void setFlag(boolean flag) {  
        this.flag = flag;  
    }  
}
```

Figure 9.12. An example of an AGG generated checkbox

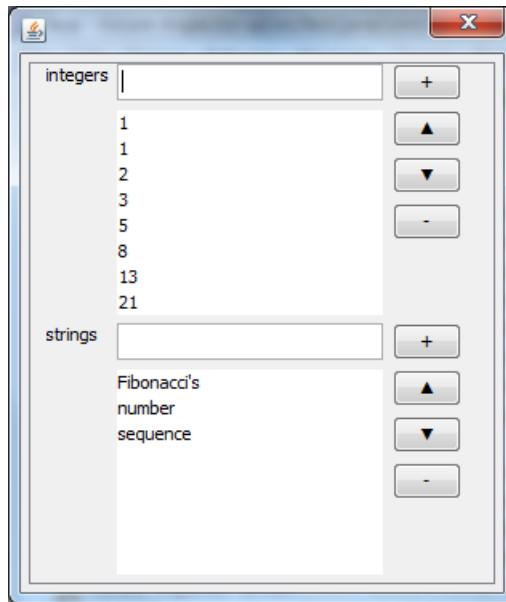


9.2.8 How do I display an arbitrary amount of primitives?

Properties of type **List** of a primitive type, **byte**, **short**, **int**, **long**, **BigInteger**, **float**, **double**, **BigDecimal** and **String** are displayed by the AGG as a reorderable CRUD-view. The C and D operations are of course only supported if the backing list supports these operations.

Example 9.20. Bean code demonstrating a reorderable arbitrary number of CRUD views

```
public class ListOfPrimitivesExample {  
  
    private List<Integer> integers;  
    private List<String> strings;  
  
    public ListOfPrimitivesExample() {  
        this.integers = Lists.newArrayList(1,1,2,3,5,8,13,21);  
        this.strings = Lists.newArrayList("Fibonacci's", "number", "sequence");  
    }  
  
    public List<Integer> getIntegers() {  
        return this.integers;  
    }  
    public void setIntegers(List<Integer> integers) {  
        this.integers = integers;  
    }  
    public List<String> getStrings() {  
        return this.strings;  
    }  
    public void setStrings(List<String> strings) {  
        this.strings = strings;  
    }  
}
```

Figure 9.13. Examples of AGG generated reorderable CRUD views

9.2.9 How do I display an arbitrary amount of nested beans?

There are two types of properties that can be used to make the AGG display an arbitrary amount of nested beans:

1. A **Map<String,T>** property.
2. A **List<T>** property annotated with the **@Reorderable** annotation.

The first type of property can be used to generate a tabbed view of the values. The second type of property is used to generate a reorderable view of the values. Both methods will be demonstrated in this section.

Example 9.21. The class that will be shown as a nested bean by the examples in this section

```
public class SubBean {  
    private int integerProperty;  
    private double decimalProperty;  
  
    public SubBean() {  
        this(1, 2.5);  
    }  
  
    public SubBean(int integerProperty, double decimalProperty) {  
        this.integerProperty = integerProperty;  
        this.decimalProperty = decimalProperty;  
    }  
  
    public int getIntegerProperty() {  
        return this.integerProperty;  
    }  
    public void setIntegerProperty(int integerProperty) {  
        this.integerProperty = integerProperty;  
    }  
    public double getDecimalProperty() {  
        return this.decimalProperty;  
    }  
    public void setDecimalProperty(double decimalProperty) {  
        this.decimalProperty = decimalProperty;  
    }  
}
```

9.2.9.1 Map<String, Object>

In this method, the key of the Map property is used to identify and name the generated subpanels.

The following method is demonstrated in Code Listing . A backing field is used to implement two properties, **listView** and **tabbedView**. Notice that the only difference between this properties is the presence or absence of the @Presentation(compact=true) as annotation. The result is shown in Figure .

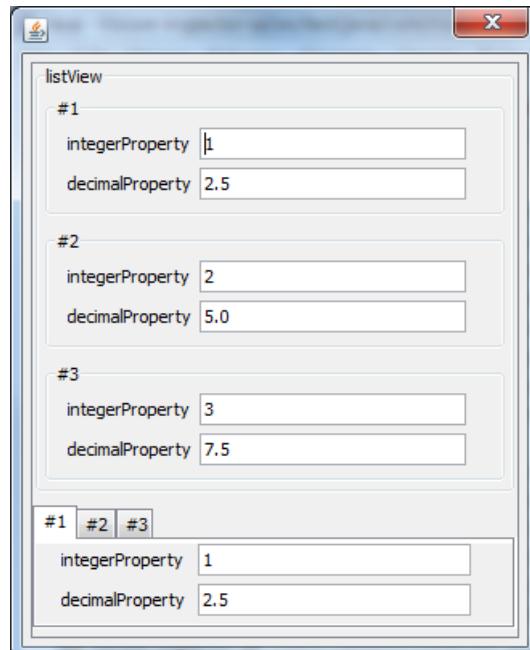
It should be noted that the order of the displayed tabs is the same as the order from which elements are retrieved from the Map. In cases where the order of tabs matters, LinkedHashMap would be the preferred Map implementation to use.

Example 9.22. Bean code demonstrating stacked list and tabbed interface of same property objects

```
public class ArbitraryWithMapExample {  
    private Map<String, Object> backingField;  
    public TestBean() {  
        this.backingField = Maps.newLinkedHashMap();  
        this.backingField.put("#1", new SubBean());  
        this.backingField.put("#2", new SubBean(2, 5));  
        this.backingField.put("#3", new SubBean(3, 7.5));  
    }  
  
    public Map<String, Object> getListView() {  
        return this.backingField;  
    }  
  
    public void setListView(Map<String, Object> backingField) {  
        this.backingField = backingField;  
    }  
  
    @Presentation(compact=true)  
    public Map<String, Object> getTabbedView() {  
        return this.backingField;  
    }  
  
    public void setTabbedView(Map<String, Object> backingField) {  
        this.backingField = backingField;  
    }  
}
```

results in this image:

Figure 9.14. An example of a stacked list and a tabbed view of the same property



9.2.9.2 @Reorderable

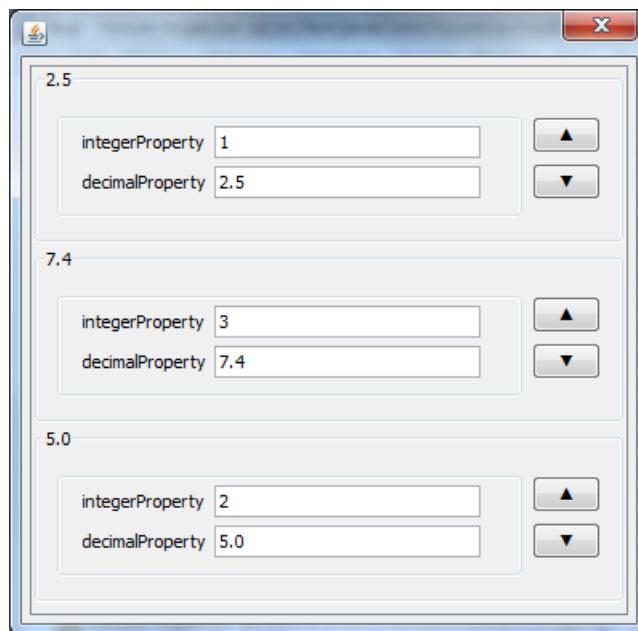
In this method the **@Reorderable** annotation is used on a List property to signal to the AGG that a reorderable view of the contents should be generated. The **@Reorderable** annotation has an attribute called **methodName()** that can be used to indicate what method to call on the objects in the list to obtain an end-user-friendly name for that value. If the **methodName()** attribute is not used, no friendly name will be displayed.

The next example demonstrates the use of **@Reorderable**. The **getDecimalPropertyMethod()** of the SubBean class will be used to obtain an end-user friendly display name for the values in the list.

Example 9.23. Bean code demonstrating a reorderable stacked list.

```
public class ArbitraryWithReorderableExample {  
  
    @Reorderable(methodName="getDecimalProperty")  
    private List<SubBean> reorderableList = Lists.newArrayList(  
        new SubBean(), new SubBean(2, 5), new SubBean(3, 7.5) );  
    public List<SubBean> getReorderableList() {  
        return reorderableList;  
    }  
    public void setReorderableList(List<SubBean> reorderableList) {  
        this.reorderableList = reorderableList;  
    }  
}
```

Figure 9.15. An example of a reorderable stacked list



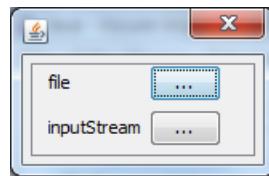
9.2.10 How do I display an open file dialog?

File and **InputStream** properties are displayed by the AGG as a button that opens an open file dialog after clicking on it.

Example 9.24. Bean code demonstrating buttons that open a file dialog

```
public class FilePickerExample {  
  
    private File file;  
    private InputStream inputStream;  
  
    public File getFile() {  
        return this.file;  
    }  
    public void setFile(File file) {  
        this.file = file;  
    }  
  
    public InputStream getInputStream() {  
        return this.inputStream;  
    }  
    public void setInputStream(InputStream inputStream) {  
        this.inputStream = inputStream;  
    }  
}
```

Figure 9.16. Displaying a File and InputStream property



9.3 How do I show/hide certain properties in the AGG?

The following sections describe methods of showing and hiding properties in the AGG that are best fit to specific scenarios.

9.3.1 How do I always hide a property?

Annotate a bean property with **@Hidden** annotation on a property and the AGG will never display it.

This annotation can also be used on nested bean properties.

9.3.2 How do I show or hide advanced/expert properties?

Property annotated with the **@Expert** annotation can be made visible or hidden in the view resulting from **BeanPanelUtils.getBeanPanel()**.

This annotation can also be used on nested bean properties.

Example 9.25. Toggling expert properties on an AGG generated view

```
Object bean = // some instance;
final JPanel beanPanel = BeanPanelUtils.getBeanPanel(bean);
final JCheckBox expertToggle = new JCheckBox("Show expert properties");

expertToggle.addChangeListener(new ChangeListener() {
    @Override
    public void stateChanged(ChangeEvent e) {
        if(beanPanel instanceof FullPanel) {
            ((FullPanel) beanPanel).setShowExpert(expertToggle.isSelected());
        }
    }
});

JPanel container = new JPanel(new BorderLayout());
container.add(beanPanel, BorderLayout.CENTER);
container.add(expertToggle, BorderLayout.SOUTH);
```

9.3.3 How do I show or hide properties based on a certain condition?

This can be achieved by using the `@Applicable` annotation or with property change support

9.3.3.1 `@Applicable`

The `@Applicable` annotation can be used to show or hide a property based on the value of another boolean property. To do this, the property of which the visibility needs to be toggled needs to be annotated with `@Applicable("<propertyName>")` where `propertyName` is the name of the `boolean` property.

The following example shows two properties, `applicableUser` and `anotherApplicableUser`, which will be shown when `applicableFlag` is true and will be hidden when `applicableFlag` is false.

Note that the method `FullPanel.getCurrentErrorMessages()`, described in Section 9.1.6, “How do I find all constraint violations on a JavaBean instance?”, only includes error messages of properties annotated with `@Applicable` when these properties are being displayed at the time of calling the method.

Also note that the `@Applicable` annotation cannot be used by a nested property to refer to a property in the parent bean class.

Example 9.26. An example of two properties making use of @Applicable

```
public class ApplicableExample {  
    private boolean applicableFlag;  
  
    @Applicable("applicableFlag")  
    private int applicableUser;  
  
    @Applicable("applicableFlag")  
    private double anotherApplicableUser;  
  
    public boolean isApplicableFlag() {  
        return this.applicableFlag;  
    }  
  
    public void setApplicableFlag(boolean applicableFlag) {  
        boolean old = this.applicableFlag;  
        this.applicableFlag = applicableFlag;  
        pcs.firePropertyChange("applicableFlag", old, this.applicableFlag);  
    }  
  
    public int getApplicableUser() {  
        return this.applicableUser;  
    }  
  
    public void setApplicableUser(int applicableUser) {  
        this.applicableUser = applicableUser;  
    }  
  
    public double getAnotherApplicableUser() {  
        return this.anotherApplicableUser;  
    }  
  
    public void setAnotherApplicableUser(double anotherApplicableUser) {  
        this.anotherApplicableUser = anotherApplicableUser;  
    }  
  
    /*  
     * Property change support <code omitted>  
     */  
  
    // Property change support code here  
}
```

9.3.3.2 Property change support and custom logic

In this method the fact that the AGG does not (visibly) render a JavaBean when null is passed into it is used to show or hide certain properties based on the value of other properties.

The following example demonstrates a property (**dispatchedProperty**) when another property (**dispatchingProperty**) has the value **10**, and hiding it when the other property has any value other than **10**. Notice how **setDispatchingProperty()** performs this check and then passes an instance or **null** to **setDispatchedProperty()**. **setDispatchedProperty()** will fire a property change event, which will trigger the AGG to re-render that property.

Example 9.27. Showing and hiding properties using property change support and custom logic

```
public class CustomShowHideExample {

    private int dispatchingProperty;
    private CustomShowHideExample dispatchedProperty;

    public int getDispatchingProperty() {
        return this.dispatchingProperty;
    }

    public void setDispatchingProperty(int dispatchingProperty) {
        this.dispatchingProperty = dispatchingProperty;
        if(this.dispatchingProperty == 10) {
            // Show dispatchedProperty
            this.setDispatchedProperty(new CustomShowHideExample());
        } else {
            // Hide dispatchedProperty
            this.setDispatchedProperty(null);
        }
    }

    public CustomShowHideExample getDispatchedProperty() {
        return this.dispatchedProperty;
    }

    public void setDispatchedProperty(CustomShowHideExample dispatchedProperty) {
        CustomShowHideExample old = this.dispatchedProperty;
        this.dispatchedProperty = dispatchedProperty;
        pcs.firePropertyChange("dispatchedProperty", old, this.dispatchedProperty);
    }

    /*
     * Property change support <code omitted>
     */

    // Property change support code here
}
```

9.3.3.3 @Applicable in combination with property change support and a read-only boolean property

It is possible to combine the previously described approaches as demonstrated in this section.

A read-only boolean property (**dispatchedPropertyApplicable**) is introduced and **dispatchedProperty** is annotated with `@Applicable("dispatchedPropertyApplicable")`. Whenever the mutator of **dispatchingProperty** is called, it simply fires a property change event for **dispatchedPropertyApplicable**. The AGG takes care of the rest.

This approach is also more reusable in the sense that multiple properties can be annotated with `@Applicable("dispatchedPropertyApplicable")` to make them dependent on **dispatchingProperty**.

Example 9.28. An alternative way of using @Applicable with a read-only boolean property

```
public class ApplicableWithReadOnlyPropertyExample {
    private int dispatchingProperty;

    @Applicable("dispatchedPropertyApplicable")
    private String dispatchedProperty = "I will be shown if dispatchingProperty == 10";

    public int getDispatchingProperty() {
        return this.dispatchingProperty;
    }

    public void setDispatchingProperty(int dispatchingProperty) {
        this.dispatchingProperty = dispatchingProperty;
        pcs.firePropertyChange("dispatchedPropertyApplicable", null, null);
    }

    @Hidden
    public boolean isDispatchedPropertyApplicable() {
        return this.dispatchingProperty == 10;
    }

    public String getDispatchedProperty() {
        return this.dispatchedProperty;
    }

    public void setDispatchedProperty(String dispatchedProperty) {
        this.dispatchedProperty = dispatchedProperty;
    }

    /*
     * Property change support <code omitted>
     */
    // Property change support code here
}
```

9.4 How do I annotate properties at runtime rather than at compile-time?

Runtime annotations are not part of the Java language. They are however available in Inspector by annotating a property with the **@Programmatic** annotation. Details are described in the following sections.

9.4.1 Instantiating annotations at runtime

In conventional Java programming, annotations are instantiated by the programmer at compile-time rather than at runtime. The Java runtime then takes care that an annotation instance is available at runtime.

Since an annotation, like **@Min**, is in essence a Java interface, a programmer can implement the annotation and instantiate an instance at runtime. There are however some specific rules for implementing the **equals()** and **hashCode()** methods for annotations . JSR 299 defines the **AnnotationLiteral** class that implements **equals()** and **hashCode()** for annotations in a generic way.

The usage of this class is demonstrated by the next code, which is implementing the @Max annotation. The @Max annotation has 4 attributes which need to be implemented. The implementation allows the value() and message() attributes to be set through the constructor and returns the default values for the other attributes.

Please note that an interface cannot be implemented as an anonymous inner class if you want to use **AnnotationLiteral** to take care of **equals()** and **hashCode()**.

The **AnnotationLiteral** class can be found in Maven dependency **javax.enterprise:cdi-api**.

Example 9.29. Implementing an annotation interface with the aid of **AnnotationLiteral**

```
public class MaxQualifier extends AnnotationLiteral<Max> implements Max {  
    private long value;  
    private String message;  
  
    public MaxQualifier(long value) {  
        this(value, "Value cannot exceed '" + value + "'");  
    }  
  
    public MaxQualifier(long value, String message) {  
        this.value = value; this.message = message;  
    }  
  
    @Override  
    public String message() {  
        return this.message;  
    }  
  
    @Override  
    public Class<?>[] groups() {  
        return new Class<?>[0];  
    }  
  
    @SuppressWarnings("unchecked")  
    @Override  
    public Class<? extends Payload>[] payload() {  
        return new Class[0];  
    }  
    @Override  
    public long value() {  
        return this.value;  
    }  
}
```

9.4.2 The @Programmatic annotation

When a property is annotated with **@Programmatic**, it is assumed that the JavaBean that declaring the property also declares a method named **<propertyName>Annotations** returning a **Collection<Annotation>**. The **@Programmatic** annotation also defines a **methodName()** attribute that overrides the default method name to look for.

The next table shows some examples of these naming conventions. The code example shows a JavaBean that defines two properties, **x** and **y**, which are both compile-time annotated with **Max(5)**. It makes uses of the **MaxQualifier** class defined in the previous section.

Table 9.6. @Programmatic method naming conventions

| Property name | @Programmatic declaration | Expected method declaration |
|---------------|--|--|
| example | @Programmatic | public Collection<Annotation> exampleAnnotations() |
| example | @Programmatic(methodName="customName") | public Collection<Annotation> customName() |
| someProperty | @Programmatic | public Collection<Annotation> somePropertyAnnotations() |
| x | @Programmatic(methodName="customName") | public Collection<Annotation> customName() |

Example 9.30. Usages of the @Programmatic annotation

```
public class ProgrammaticAnnotationExample {

    @Programmatic
    private int x;

    @Programmatic(methodName="xAnnotations")
    private int y;

    public int getX() {
        return this.x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public Collection<Annotation> xAnnotations() {
        return Arrays.asList( (Annotation) new MaxQualifier(5) {} );
    }

    public int getY() {
        return this.y;
    }

    public void setY(int y) {
        this.y = y;
    }
}
```

9.4.3 The thin wrapper that makes @Programmatic work

The Inspector code base contains a package **com.riscure.beans**. In this package the standard Java reflection package **java.lang.reflect**, the standard Java beans package **java.beans**, and the Apache Commons beans package **org.apache.commons.beanutils** are wrapped to provide a single and simple entry point to these related functionalities.

The AGG works directly with the **com.riscure.beans** package and in particular, it makes heavy use of the **BeanProperty** interface therein. This interface defines two methods related to obtaining annotations. Based on the presence of the **@Programmatic** annotation on a property, these methods retrieve annotations via the standard packages or via the custom logic defined in section The **@Programmatic** annotation.

9.5 How do I expose properties for tuning during module execution time?

Runtime-tuning of properties during module execution can be achieved by using the **@Tunable** annotation and some additional programming effort. We will discuss each step in the following sections.

9.5.1 The @Tunable annotation

The **@Tunable** annotation is designed to be a flag which is used to annotate all properties a user wants to expose during the module executions.

The following example demonstrates how to use the **@Tunable** to mark the properties to be exposed during module execution.

Example 9.31. Annotate properties with @Tunable

```
public class DummySubPropertyOne {

    @DisplayName("1.2: Should show")
    @Tunable
    public String dsptwo;

    @DisplayName("1.3: Should not show")
    private String dspthree;

    public DummySubPropertyOne() {
    }
    public String getDspthree() {
        return this.dspthree;
    }

    public void setDspthree(String dspthree) {
        String old = this.dspthree;
        this.dspthree = dspthree;
        pcs.firePropertyChange("dsptwo", old, this.dspthree);
    }

    public String getDsptwo() {
        return this.dsptwo;
    }

    public void setDsptwo(String dsptwo) {
        String old = this.dsptwo;
        this.dsptwo = dsptwo;
        pcs.firePropertyChange("dsptwo", old, this.dsptwo);
    }

    private PropertyChangeSupport pcs = new PropertyChangeSupport(this);

    public void addPropertyChangeListener(PropertyChangeListener listener) {
        this.pcs.addPropertyChangeListener(listener);
    }
}
```

9.5.2 Acquire a JPanel filled with @Tunable annotated properties

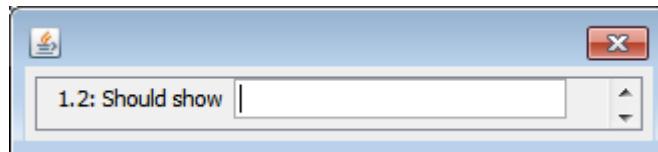
Below code snippet demonstrates how to filter on the `@Tunable` annotation from java bean properties.

Example 9.32. GUI panel with @Tunable properties

```
dp = new DummySubPropertyOne();
JPanel tunablePanel = BeanPanelUtils.getBeanPanel(dp, false, new
PropertySelector() {
    @Override
    public boolean selectProperty(BeanProperty beanProperty) {
        boolean hasTunableAnnotation =
        beanProperty.getAnnotation(Tunable.class) != null;
        return hasTunableAnnotation;
    }
});
```

The Java JPanel **tunablePanel** (see Figure 9.17, “JPanel filtered on @Tunable”) will then only contain the propertie(s) being annotated with `@Tunable`.

Figure 9.17. JPanel filtered on @Tunable



9.5.3 How to popup the tunable panel during module execution?

A typical approach is to set a flag from within the GUI, e.g. a checkbox, to let the user enable/disable the runtime tuning function. In the `init()` method of the custom-made **Target** or **MeasurementSetup**, some additional code needs to be developed to pop up the tunable panel.

9.6 How do I extend the AGG?

It is currently not possible to extend the Automatic GUI Generator.

9.7 How do I implement an acquisition2/perturbation2 compatible hardware device driver?

The acquisition2/perturbation2 frameworks make use of hardware devices registered in Inspector's Hardware Manager (HWM). The various methods of implementing a new hardware device that is compatible with the HWM will be discussed in this section.

9.7.1 How do I register and unregister a device in the HWM?

The HWM is based on the OSGi Service Platform. The HWM registry is implemented using OSGi's service registry. The interface from which all HWM compatible interfaces and implementations of hardware devices derive is **DeviceInterface<? extends Session>**. The HWM simplifies registering objects of this type by providing the method call **com.riscure.hardware.HardwareUtils.registerDevice(...)**. This method returns a **ServiceRegistration** which can be used to unregister the device at a later point in time by calling its **unregister()** method.

This method has several definitions as a means to implement default argument passing. The arguments are explained in this table:

Table 9.7. HardwareUtils.registerDevice(...) argument overview

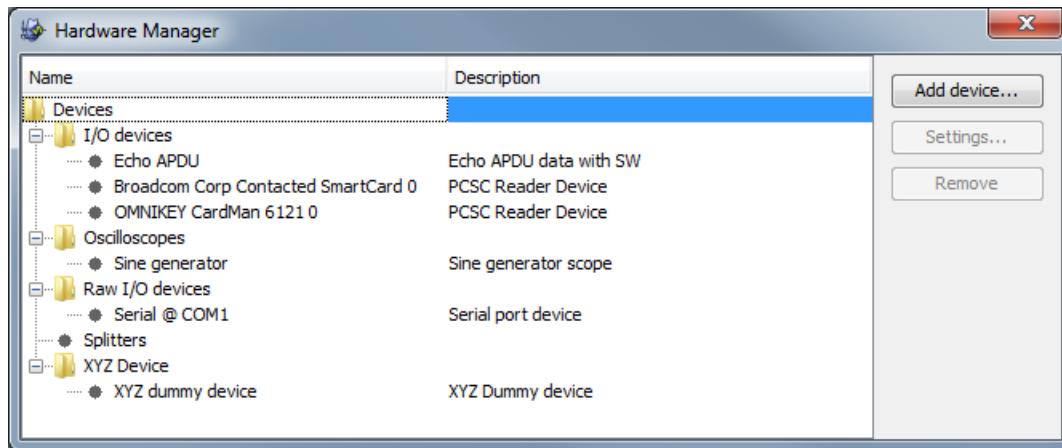
| Argument | Type | Meaning | Default |
|----------------------------------|--|---|-------------------------|
| context | BundleContext | The BundleContext through which the OSGi service registry is approached. | N/A |
| deviceClass/deviceClasses | Class<?>/Class<?>[] | The classes (in other words, device types) under which the device should be registered. | N/A |
| device | DeviceInterface<?> | The device to register. | N/A |
| properties | Properties | The properties associated with this registration. | new Properties() |

Devices registered in the HWM will appear in the HWM dialog shown in Figure . The nodes "I/O device", "Oscilloscopes", "Raw I/O devices" and "XYZ device" correspond to the **deviceClass/deviceClasses** argument of **HardwareUtils.registerDevice(...)**. Actual device entries like "Echo APDU" correspond to the **device** argument.

There are three categories of devices that are registered by the **HardwareUtils.registerDevice(...)** call, namely

1. Static: Devices which are always present in the HWM. This is the case for 'dummy' devices such as the Echo APDU device and the Sine generator scope. These are ordinarily not removed.
2. PnP: Devices that need to be registered and unregistered in the HWM and on the basis of some form of Plug and Play (PnP), such as the PowerTracer.
3. Manual: A user manually registers/adds a device by configuring it through the "Add device..." button in the HWM dialog. This is for example necessary with networked devices for which an IP address has to be configured, such as the LeCroy oscilloscope. These devices are manually unregistered/removed by the user as well.

All situations are explained in sections How do I register a 'dummy' device?, How do I register and unregister devices on the basis of PnP events? and How do I allow an end-user to manually register and unregister a device in the HWM? respectively. Prior to that, the method of obtaining the context argument of type **BundleContext**, required by **HardwareUtils.registerDevice(...)**, will be shown in Section 9.7.2, "How do I make Inspector detect and call my device driver code?"

Figure 9.18. The Hardware Manager dialog

Further documentation can be found on the JavaDoc page of the `DeviceInterface` [..`/javadoc/com/riscure/hardware/DeviceInterface.html`] class. The reader is encouraged to browse through the Inspector API [..`/javadoc/index.html`] pages. In this manual we will not provide a JavaDoc link to each specific class.

9.7.2 How do I make Inspector detect and call my device driver code?

Inspector expects hardware device drivers to be implemented as OSGi service bundles. Therefore the entry point of all device driver code is an implementation of **OSGi's BundleActivator** interface.

Inspector does not read any of OSGi's extensions to a JAR's manifest file, located in `META-INF/MANIFEST.MF` relative from the JAR's root. Therefore, if a JAR is an OSGi service bundle, its activator is not recognised through the manifest file either. Moreover, in Inspector a bundle is allowed to have multiple bundle activators. All activators that should be called are to be listed by their fully qualified name in a file called `org.osgi.framework.BundleActivator` located in the folder `META-INF/services/` relative to the JAR's root. In this file, every activator is placed on a new line.

Example 9.33. An example of the `org.osgi.framework.BundleActivator` file, indicating that two activators are present in the JAR

```
com.riscure.demo.hardware.scope.SomeScopeActivator
com.riscure.demo.hardware.powertracer.SomePowerTracerVersionActivator
```

The activator will receive a reference to a `BundleContext` in its `start(...)` method, which can and should be used to make the call to `HardwareUtils.registerDevice(...)`. This will be demonstrated in some detail in the following sections.

9.7.3 How do I register a 'dummy' device?

Simply call `HardwareUtils.registerDevice(...)` in the `start(...)`-method of the activator, as shown in the next example. The highlighted code registers a device under the "I/O devices" node. Naturally, other nodes can be used as well. An oscilloscope for example, would be registered by using `Oscilloscope.class` as value of the `deviceClass` parameter of `HardwareUtils.registerDevice(...)`. The effect of this registration is shown in Figure ,

where the static device is selected in the HWM dialog. The values in the name and description columns are derived from the `getName()` and `getDescription()` methods of the `DeviceInterface` interface respectively.

Example 9.34. Registering a static device in Inspector's HWM

```
package com.riscure.demos.hardware.staticdevice;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;
import com.riscure.hardware.HardwareUtils;
import com.riscure.hardware.io.IODevice;

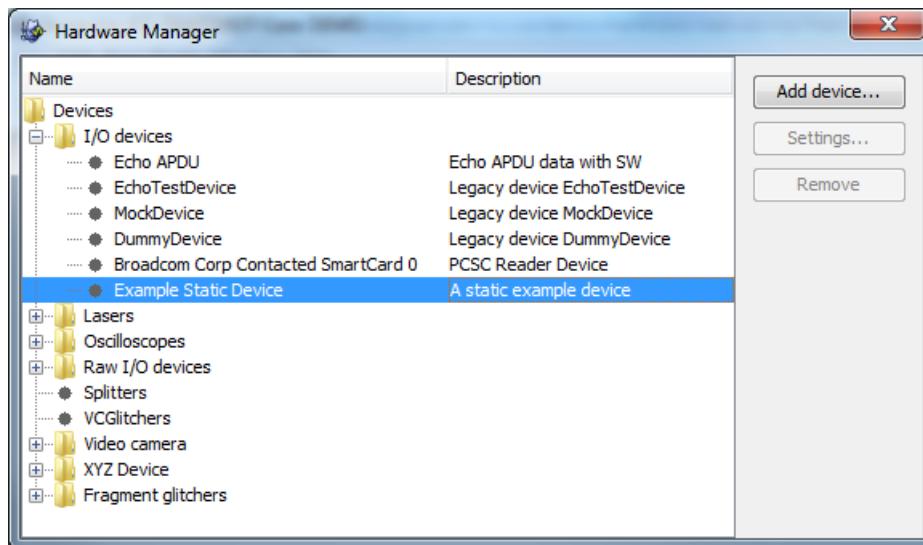
public class StaticDeviceExampleActivator implements BundleActivator {

    private ServiceRegistration deviceRegistration;

    @Override
    public void start(BundleContext context) throws Exception {
        // Register the device under the "I/O devices" node
        this.deviceRegistration = HardwareUtils.registerDevice(
            context, IODevice.class, new StaticDeviceExampleDevice());
    }

    @Override
    public void stop(BundleContext context) throws Exception {
        // Unregister the device
        this.deviceRegistration.unregister();
    }
}
```

Figure 9.19. The newly added static device in the hardware manager dialog



As can be seen from Figure , the "Settings..." button is not enabled for the static device. In order to enable this button, the method `getSettings()` of `DeviceInterface` should return a non-null instance of a JavaBean that can be displayed by Inspector's AGG. Device settings are settings that should apply for all future sessions created on the device. The proper use of these settings will be discussed in section How do I implement a session on a device?. The following

example demonstrates how settings can be used to allow a user to change the display name and description of a device.

```
package com.riscure.demos.hardware.staticdevice;

import com.riscure.hardware.DeviceInterface;

public class StaticDeviceExampleDevice
    implements DeviceInterface<StaticDeviceExampleSession> {

    private StaticDeviceExampleSettings settings = new StaticDeviceExampleSettings();

    @Override
    public String getDeviceId() {
        return StaticDeviceExampleDevice.class.getName();
    }

    @Override
    public String getDriverId() {
        // No driver used
        return this.getDeviceId();
    }

    @Override
    public String getName() {
        return this.settings.getName();
    }

    @Override
    public String getDescription() {
        return this.settings.getDescription();
    }

    @Override
    public boolean isUserCreated() {
        // Static devices are not user created
        return false;
    }

    @Override
    public boolean isDeviceManagedSettings() {
        // Let the HWM manage the persistence of the settings of this device
        // by returning false
        return false;
    }

    @Override
    public Object getSettings() {
        // An AGG compatible JavaBean is expected by the HWM
        return this.settings;
    }

    @Override
    public void setSettings(Object settings) {
        this.settings = (StaticDeviceExampleSettings) settings;
    }

    public static class StaticDeviceExampleSettings {
        private String name = "Example Static Device";
        private String description = "A static example device";
    }
}
```

```
public String getName() {
    return this.name;
}

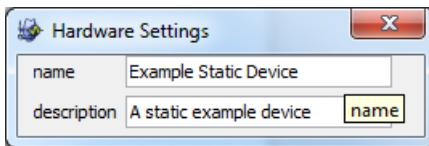
public void setName(String name) {
    this.name = name;
}

public String getDescription() {
    return this.description;
}

public void setDescription(String description) {
    this.description = description;
}
}

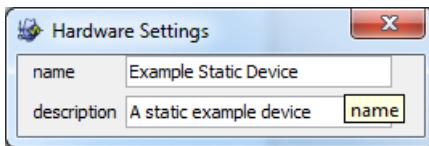
// <Further code omitted>
```

Figure 9.20. Settings presented to the end-user to allow him to change the name and description of the static device



These settings are immediately visible in the hardware manager dialog.

Figure 9.21. Settings presented to the end-user to allow him to change the name and description of the static device



9.7.4 How do I register and unregister devices on the basis of PnP events?

The technique for registering and unregistering devices in the HWM based on PnP events is an extension to the technique used to registering static devices in the HWM. In the case of static devices, the service bundle activator directly registers a device. In the case PnP devices, the activator needs to (1) instantiate a listener that is able to (2) detect PnP events for the specific type of device that needs to be registered and unregistered and, on the basis of that listener, (3) register a device when it is plugged and remove a device when it is plugged out. Finally it needs to, (4) perform the bootstrapping operation of registering all devices already plugged in before the activator/listener was started.

The next example demonstrates the skeleton of an activator that follows the above four steps. The four steps are highlighted in green when they appear. This Code Listing only gives a very rough sketch of how to implement the listener class. The remainder of this section is dedicated to the details of implementing the listener class.

```
package com.riscure.demos.hardware.pnpdevice;

import java.util.List;
import java.util.Map;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;
import com.google.common.collect.Maps;
import com.google.common.collect.ImmutableList;
import com.riscure.hardware.HardwareUtils;
import com.riscure.hardware.io.IODevice;

public class PnPDeviceExampleActivator
    implements BundleActivator, PnPDeviceExampleRegistrationProxy {

    private BundleContext context;
    private Map<PnPDeviceExampleDevice, ServiceRegistration> registrations =
        Maps.newHashMap();

    @Override
    public void start(BundleContext context) throws Exception {
        this.context = context;
        // (1) Create a Listener
        PnPDeviceExampleListener listener = new PnPDeviceExampleListener(this);
        // ... <code to hook the listener>

        // (4) Register all devices already plugged before start() was called
        List<PnPDeviceExampleDevice> devices; // ... <code to obtain connected devices>

        for (PnPDeviceExampleDevice device : devices) {
            register(device);
        }
    }

    @Override
    public void stop(BundleContext context) throws Exception {
        List<PnPDeviceExampleDevice> devices =
            ImmutableList.copyOf(this.registrations.keySet());

        for (PnPDeviceExampleDevice device : devices) {
            unregister(device);
        }
    }

    // (3) Register a device

    @Override
    public void register(PnPDeviceExampleDevice device) {
        if(!registrations.containsKey(device)) {
            registrations.put( device, HardwareUtils.registerDevice(context, IODevice.class,
device));
        }
    }

    // (3) Unregister a device
    @Override
    public void unregister(PnPDeviceExampleDevice device) {
        if(this.registrations.containsKey(device)) {
            this.registrations.get(device).unregister();
        }
    }
}
```

```
}

}

package com.riscure.demos.hardware.pnpdevice;

public interface PnPDeviceExampleRegistrationProxy {

    void register(PnPDeviceExampleDevice device);

    void unregister(PnPDeviceExampleDevice device);
}

package com.riscure.demos.hardware.pnpdevice;

public class PnPDeviceExampleListener implements SomeListener {

    private PnPDeviceExampleRegistrationProxy proxy;

    public PnPDeviceExampleListener(PnPDeviceExampleRegistrationProxy proxy) {
        this.proxy = proxy;
    }

    // (2) Detect plug of target device type
    @Override
    public void handleAdd(SomeInfo info) {
        if(this.infoIsOfTargetDeviceType(info)) {
            PnPDeviceExampleDevice device;
            // ... <code to obtain device based on info here>
            // (3) Register device
            proxy.register(device);
        }
    }

    // (2) Detect unplug of target device type
    @Override
    public void handleRemove(SomeInfo info) {
        if(this.infoIsOfTargetDeviceType(info)) {
            PnPDeviceExampleDevice device;

            // ... <code to obtain device based on info here>

            // (3) Unregister device
            proxy.unregister(device);
        }
    }

    private boolean infoIsOfTargetDeviceType(SomeInfo info) {
        boolean infoIsOfTargetDeviceType;
        // ... <code to determine condition here>
        return infoIsOfTargetDeviceType;
    }
}
```

9.7.4.1 Windows hardware events

The HWM allows implementations of the **DeviceScanner** interface registered in Inspector's registry to listen to Windows's hardware events. The listener (**DeviceScanner**) can determine whether the device that was plugged in is interesting to it by, for example, checking the

device's vendor and product ID. This is for example useful when a device is connected to the PC via USB. The next example shows how to make use of the **DeviceScanner** interface.

```
package com.riscure.demos.hardware.pnpdevice;
import java.util.List;
import java.util.Map;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;
import com.google.common.collect.Maps;
import com.google.common.collect.ImmutableList;
import com.riscure.hardware.DeviceScanner;
import com.riscure.hardware.HardwareUtils;
import com.riscure.hardware.io.IODevice;

public class PnPDeviceExampleActivator
    implements BundleActivator, PnPDeviceExampleRegistrationProxy {

    private BundleContext context;
    private ServiceRegistration scannerRegistration;
    private Map<PnPDeviceExampleDevice, ServiceRegistration> registrations =
        Maps.newHashMap();

    @Override
    public void start(BundleContext context) throws Exception {
        this.context = context;
        // (1) Create a Listener
        PnPDeviceExampleListener listener = new PnPDeviceExampleListener(this);
        scannerRegistration = context.registerService( DeviceScanner.class.getName(),
            listener, null );
    }

    @Override
    public List<PnPDeviceExampleListener> getRegisteredDevices() {
        return ImmutableList.copyOf(this.registrations.keySet());
    }
    // <code omitted>
}

package com.riscure.demos.hardware.pnpdevice;
import java.util.List;

public interface PnPDeviceExampleRegistrationProxy {
    List<PnPDeviceExampleDevice> getRegisteredDevices();
    void register(PnPDeviceExampleDevice device);
    void unregister(PnPDeviceExampleDevice device);
}

package com.riscure.demos.hardware.pnpdevice;
import com.riscure.hardwareDeviceInfo;
import com.riscure.hardware.DeviceScanner;
import java.util.List;

public class PnPDeviceExampleListener implements DeviceScanner {
    private static final String VENDOR_PRODUCT_ID = "VID_0000&PID_0000";
    private PnPDeviceExampleRegistrationProxy proxy;

    public PnPDeviceExampleListener(PnPDeviceExampleRegistrationProxy proxy) {
        this.proxy = proxy;
        // (4) Register all devices already plugged before start() was called
    }
}
```

```
    this.updateRegistrations();
}

// (2) Detect plug of target device type
@Override
public void deviceAdded(DeviceInfo info) {
    handleDeviceChange(info);
}

// (2) Detect unplug of target device type
@Override
public void handleRemove(DeviceInfo info) {
    handleDeviceChange(info);
}

private void handleDeviceChange(DeviceInfo info) {
    if(this.infoIsOfTargetDeviceType(info)) {
        this.updateRegistrations();
    }
}

// (3) Register and unregister devices based on the actual situation
private void updateRegistrations() {
    List<PnPDeviceExampleDevice> connectedDevices;

    // ... <code to obtain connected devices>
    List<PnPDeviceExampleDevice> registeredDevices =
        this.proxy.getRegisteredDevices();
    for (PnPDeviceExampleDevice connectedDevice : connectedDevices) {

        // if the reference list already contains the device,
        // remove it from the reference list
        if(registeredDevices.contains(connectedDevice)) {
            registeredDevices.remove(connectedDevice);
        // otherwise, register it
        } else {
            this.proxy.register(connectedDevice);
        }
    }

    // the reference list contains the devices that are not present anymore
    // unregister all the targets that are not present anymore
    for(PnPDeviceExampleDevice registeredDevice : registeredDevices) {
        this.proxy.unregister(registeredDevice);
    }
}

private boolean infoIsOfTargetDeviceType(DeviceInfo info) {
    return info.getPath().contains(VENDOR_PRODUCT_ID);
}
}
```

9.7.4.2 COM port events

A **PortStatusListener** registered to Inspector's **com.riscure.port.PortManager** class is able to listen to COM port events. The **PortStatusListener** receives as event argument to its methods a **PortStatus** object, which can be opened to obtain an instance of the **CommPort** interface. This instance in turn can be used to obtain an **InputStream** and an **OutputStream**.

In the remainder of this section only the registration of a listener to **PortManager** is shown in the following code.

```
package com.riscure.demos.hardware.pnpdevice;
import java.util.List;
import java.util.Map;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;
import com.google.common.collect.Maps;
import com.google.common.collect.ImmutableList;
import com.riscure.hardware.HardwareUtils;
import com.riscure.hardware.io.IODevice;
import com.riscure.port.PortManager;

public class PnPDeviceExampleActivator implements BundleActivator,
    PnPDeviceExampleRegistrationProxy {
    private BundleContext context;
    private Map<PnPDeviceExampleDevice, ServiceRegistration> registrations =
        Maps.newHashMap();

    @Override
    public void start(BundleContext context) throws Exception {
        this.context = context;
        // (1) Create a listener
        PortManager manager = PortManager.getInstance();
        PnPDeviceExampleListener listener = new PnPDeviceExampleListener(manager, this);
        manager.addPortStatusListener(listener);
    }

    // <code omitted>
}
```

The next code snippet shows the changes to **PnPDeviceExampleActivator** that are needed to make use of **PortStatusListener**.

```
package com.riscure.demos.hardware.pnpdevice;

public interface PnPDeviceExampleRegistrationProxy {
    // <code omitted>
}

package com.riscure.demos.hardware.pnpdevice;

import com.riscure.port.PortStatus;
import com.riscure.port.PortStatusListener;

public class PnPDeviceExampleListener implements PortStatusListener {
    private static final String DEVICE_DESCRIPTION = "PNP EXAMPLE DEVICE";
    private PortManager manager;
    private PnPDeviceExampleRegistrationProxy proxy;
    public PnPDeviceExampleListener( PortManager manager,
        PnPDeviceExampleRegistrationProxy proxy ) {
        this.manager = manager;
        this.proxy = proxy;
    }
    // (2) Detect plug of target device type
    @Override
```

```
public void portAdded(PortStatus status) {
    if(this.statusIsOfType(status)) {
        PnPDeviceExampleDevice device;
        // ... <code to obtain device based on status here>
        // (3) Register device
        proxy.register(device);
    }
}

// (2) Detect unplug of target device type
@Override
public void handleRemove(PortStatus status) {
    if(this.statusIsOfType(status)) {
        PnPDeviceExampleDevice device;
        // ... <code to obtain device based on status here>
        // (3) Unregister device
        proxy.unregister(device);
    }
}

@Override
public void portStatusChanged(PortStatus status) {
    // Ignore changes to ownership
}
private boolean infoIsOfType(PortStatus status) {
    return DEVICE_DESCRIPTION.equals(status.getDescription());
}
}
```

9.7.4.3 Other

The Windows hardware events and COM port events are of course not the only methods to implement PnP-compatible devices. IVI scopes for example, can be made PnP-like by listening to changes to the file system. However, since Inspector does not provide a unified interface for these scenarios as it does with Windows hardware events and COM port events, further details are left unexplored.

9.7.5 How do I allow an end-user to manually register and unregister a device in the HWM?

To this end the HWM employs the concept of a driver, which is represented by the com.riscure.hardware.Driver interface. The basic HWM mechanism of allowing a user to add and remove a device and its interaction with the Driver interface will be demonstrated by example.

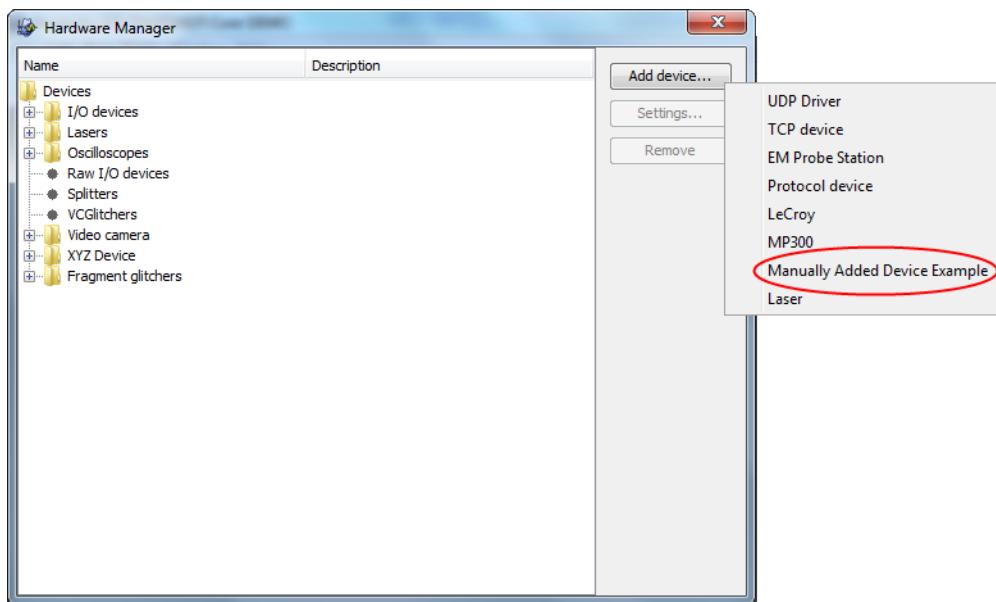
When the "Add device..." button is pressed in the HWM dialog, a list of registered drivers will appear. A driver can be added to this list by registering one in the start method of the service bundle's activator. This is demonstrated by the following code and the effect of that code is encircled in red in the figure below.

```
package com.riscure.demos.hardware.manualdevice;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;
import com.riscure.hardware.HardwareUtils;
```

```
public class ManualDeviceExampleActivator implements BundleActivator {  
  
    private ServiceRegistration driverRegistration;  
  
    @Override  
    public void start(BundleContext context) throws Exception {  
        // Register the driver  
        // Pass the context to the driver, so that devices can be registered later on.  
        this.driverRegistration = HardwareUtils.registerDriver( context, new  
ManualDeviceExampleDriver(context) );  
    }  
  
    @Override  
    public void stop(BundleContext context) throws Exception {  
        // Unregister the driver  
        this.driverRegistration.unregister();  
    }  
}
```

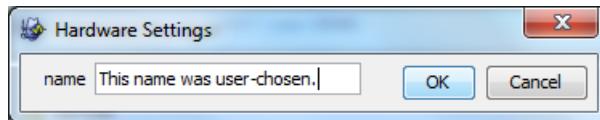
Figure 9.22. The driver listing following a press of the "Add device..."-button in the Hardware Manager dialog



When an end-user selects a driver from the list presented to him, the driver's `createSettings(...)` method is called. This method should create and return a new JavaBean that can be displayed by Inspector's AGG. The properties from the driver setup can be duplicated in the settings of the device in order to allow the user to change these properties after device creation without losing settings which are based on the unique device id.

For the purpose of this example, only the name of the device can be configured. In practical cases one might allow a user to configure an IP address and port at which the device is located, etc. The configuration dialog is shown in the next figure.

Figure 9.23. The settings bean displayed to the user after making a selection .



After the user presses "OK" in this dialog the driver's **createDevice(...)** method is called. An ID, created by the HWM and used by it to uniquely identify the device, and the bean as configured by the end-user are passed to this method. This method has the following responsibilities:

1. Create a device according to the settings provided by the end-user. The ID passed into the method should be returned by the **getDeviceId()** of the returned **DeviceInterface**.
2. Register the device in the HWM.

In the following three code examples, both tasks are left up to the constructor of the device. the last code example shows that for manual devices, **isUserCreated()** should return true and therefore **remove()** should be implemented as well.

```
package com.riscure.demos.hardware.manualdevice;

import java.io.IOException;
import org.osgi.framework.BundleContext;
import com.riscure.hardware.DeviceInterface;
import com.riscure.hardware.Driver;

public class ManualDeviceExampleDriver implements Driver {
    // The context argument needed when registering devices in the HWM
    private BundleContext context;
    public ManualDeviceExampleDriver(BundleContext context) {
        this.context = context;
    }

    @Override
    public String getDriverId() {
        return ManualDeviceExampleDriver.class.getName();
    }

    @Override
    public String getName() {
        // The name that will be displayed in the "Add device..." list
        return "Manually Added Device Example";
    }

    @Override
    public String getDescription() {
        return "An example driver that shows how to allow a user to add a device";
    }

    @Override
    public boolean isDriverManagedPersistance() {
        return false;
    }

    @Override public Object createSetup() {
        // The bean that will be displayed when the end-user wishes to create this type of
        device
```

```
    return new ManualDeviceDriverExampleSettings();
}

@Override
public DeviceInterface<?> createDevice(String id, Object setup) throws IOException {
// Create the device and return it.
return new ManualDeviceExampleDevice( id, getDriverId(),
(ManualDeviceDriverExampleSettings) setup, context );
}
}
```

Code snippet B: Settings class used by the Manual Driver code

```
public class ManualDeviceDriverExampleSettings {
private String name = "The device created by the user will have the this name";
public String getName() { return this.name; }
public void setName(String name) { this.name = name; }
}
```

Code Snippet C:Constructor class used by the Manual Driver code

```
package com.riscure.demos.hardware.manualdevice;

import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;
import com.riscure.hardware.DeviceInterface;
import com.riscure.hardware.HardwareUtils;
import com.riscure.hardware.io.IODevice;

public class ManualDeviceExampleDevice implements
DeviceInterface<ManualDeviceExampleSession> {

private String id;
private String driverId;
private String name;
private ServiceRegistration registration;

public ManualDeviceExampleDevice( String id, String driverId,
ManualDeviceDriverExampleSettings setup, BundleContext context ) {
this.id = id;
this.driverId = driverId;
this.name = setup.getName();
this.registration = HardwareUtils.registerDevice( context, IODevice.class, this
);
}

@Override
public boolean isUserCreated() {
// Returns true, required by the HWM
return true;
}

@Override
public void remove() {
// This method can be called by the HWM, because isUserCreated() returns true
this.registration.unregister();
}
// <Code omitted>
}
```

9.7.6 How do I implement a session on a device?

The **DeviceInterface** interface has an **open(...)** method which takes an **Object** argument. This method should return a **Session** object, which, as the name suggests, represents a session on a device. This session is the medium through which **acquisition2** and **perturbation2** frameworks communicate with the (physical) device. If the **DeviceInterface** object represents for example a TCP device, its **open(...)** method would be responsible for creating a TCP connection to that device and the resultant **Session** object would wrap that connection by providing methods to transmit data over that connection.

The methods that need to be implemented when implementing a specific specialisations of the **Session** interface are different for each class/type of device. The reason for this is that different classes of device have their own specific functions and as a consequence of that, have to be approached in a different way. For example, configuring a device's current position has no meaning when the device is a PC/SC-reader or an oscilloscope, but is of fundamental importance when the device is an XYZ stage. Table gives an overview of the current **DeviceInterface** specialisations and their corresponding session types. Details on how to implement the session specialisations are not given in this section, but can be found in section How do I implement specific specialisations of the Session interface?.

Table 9.8. An overview of device classes and their corresponding session type.

| DeviceInterface specialisations | Session specialisations |
|---------------------------------|---------------------------------|
| CameraVideoCamera | CameraSessionVideoCameraSession |
| ContactlessDevice | ContactlessSession |
| Dlc | DlcSession |
| FragmentGlitcher | FragmentGlitcherSession |
| Laser | Session |
| PowerSupplyDevice | PowerSupplySession |
| RawIODevice | RawIODeviceSession |
| Oscilloscope | OscilloscopeSession |
| VCGlitcherDevice | VCGlitcherSession |
| XYZDevice | XYZSession |
| IODevice | IODeviceSession |

The **Session** specialisations shown in this table can be roughly categorised in two groups:

1. Sessions that are used primarily to configure a device. A device that has this type of session is for example an XYZ stage.
2. Sessions that are used to configure a device before starting an active process on that device, such as retrieving a measurement or executing a communication protocol. Devices that have this type of session are for example oscilloscopes and I/O-devices.

The sessions of the second group introduce an additional method to retrieve something usually termed a connection. It should be noted that this additional layer has no standardised naming within Inspector.

Before **open(...)** is called on an instance of **DeviceInterface**, the **acquisition2** and **perturbation2** frameworks will first call the instance's **createPreferences()** method. The

open(...) implementation can assume that the input parameter to this method will be of the same type as returned by **createPreferences()**. The output of this method at present not used by the acquisition and perturbation2 frameworks, but is intended for future use. The object returned by this method should be a JavaBean compatible with Insecptor's AGG or **null**. This bean can be used by the frameworks to allow a user to configure settings that are specific to the session that he is about to open on the device. This is also where the difference lies between the JavaBean returned by the **getSettings()** method and the JavaBean returned by the **createPreferences()** method of **DeviceInterface**: the former, used in the HWM dialog, is used to configure settings for all future sessions on the device while the latter is used to configure settings specific to the next session on the device.

The **open()** method should finally obey the following rules:

1. If multiple sessions are supported on the device, as is possible for example for TCP devices, a new instance of **Session** should be returned. The implementation itself is responsible for concurrent calls to different active sessions.
2. If multiple sessions are not supported, an exception should be thrown when **open(...)** is called when there is already an active session on the device.

This leads to a basic skeleton of the **open(...)** method implementation shown in the next example.

Example 9.35. Skeleton implementation of DeviceInterface.open(Object prefs)

```
package com.riscure.demos.hardware.misc;
import java.io.IOException;
import com.riscure.hardware.Session;

// <code omitted>

public class SomeExampleDevice implements SomeDeviceInterface<SomeSession> {
// <code omitted>
private Session session;
@Override
public Session open(Object preferences) throws IOException {
boolean deviceHasActiveSession = this.session != null && this.session.isOpen();

if(deviceHasActiveSession) {
throw new IOException("This device does not support simultaneous sessions");
} else {
this.session = new SomeExampleSession( (SomeSettings) this.getSettings(),
(SomePreferences) preferences );
}
return this.session;
}
}
```

And this

Example 9.36. Skeleton implementation of Session

```
package com.riscure.demos.hardware.misc;

import java.io.IOException;

// <code omitted>

public class SomeExampleSession implements SomeSession {
    public boolean open;
    public SomeExampleSession( SomeSettings settings, SomePreferences
        preferences ) throws IOException {
        this.open(settings, preferences);
    }

    private void open( SomeSettings settings, SomePreferences preferences ) throws
    IOException {
        // ... <code to open connection to physical device and claim resources>
        this.open = true;
    }
    @Override
    public boolean isOpen() {
        return this.open;
    }
    @Override
    public void close() throws IOException {
        // ... <code to close connection to physical device and clean up resources>
        this.open = false;
    }
    // <code omitted>
}
```

9.7.7 How do I implement specific specialisations of the Session interface?

Since there are a wide variety of **Session** specialisations available in the HWM API, this section will focus on the implementation of **OscilloscopeSession** and **IODeviceSession**, because these **Session** specialisations cover the majority of points of interests relevant when implementing any of the Session specialisations.

9.7.8 How do I implement the OscilloscopeSession interface?

This section picks up from Section 9.7.6, “How do I implement a session on a device?” in which the opening of a session on a device was sketched, but it is assumed this method returns an instance of an implementation of **OscilloscopeSession**, called **ScopeExampleSession**.

At a first glance the implementation of **OscilloscopeSession** seems daunting, because the implementation of **OscilloscopeSession** requires the implementation of a number of different interfaces, namely

1. **OscilloscopeProperties** because of the **getProperties()** method
2. **Channel** and **ChannelProperties** because of the **getChannels()** method

3. **Trigger** and **TriggerProperties** because of the **getTrigger()** method

4. **Acquisition** and **ChannelAcquisition** through the **acquire()** method.

These interfaces and methods can be categorised in two groups:

1. Preparing the scope for performing an acquisition. All interfaces and method except the **Acquisition** and **ChannelAcquisition** the interfaces and the **acquire()** method belong to this group.
2. Performing an acquisition on the scope. This group contains only the the **Acquisition** and **ChannelAcquisition** the interfaces and the **acquire()** method.

Configuring an acquisition

Different (physical) oscilloscopes have different features and, even though they have a small common feature set, it is likely that for example on one (physical) oscilloscope its data channel's resolution can be configured whereas on another this property cannot be configured. It can further be the case that, even though two oscilloscopes both allow an end-user to configure the range to be measured on a data channel, one oscilloscope supports different data channel ranges than the other. One oscilloscope might even allow an end-user to configure a range freely (freeform) instead of requiring the end-user to choose from certain ranges only.

The **acquisition2** and **perturbation2** frameworks deal with the aforementioned variations in oscilloscope features sets by

1. enabling an implementer to indicate for certain properties whether or not freeform input is allowed by the scope for that property. Properties for which this can be indicated will have an **isFreeform<PropertyName>Supported()** method returning a boolean. An example of this type of property is the **timePerSample** property in Table .
2. enabling an implementer to indicate for certain properties what input values are allowed by the scope for that property. Properties for which this can be indicated will have an **getSupported<PropertyNames>()** method returning a List containing the values allowed by the scope for that property. An example of this is the resolution property in Table .
3. defining certain scope properties as required and other properties as optional to implement by an implementer. Properties that an implementer has to implement will not have a **is<PropertyName>Supported()** method returning a boolean, whereas optional properties will have this type of method. An implementer can indicate for optional properties that a scope supports it by returning true in the **is<PropertyName>Supported()** method. Table shows for example that **triggerTimeout** is a required property, whereas **triggerDelay** is not.
4. custom (non-standard) properties can be defined by the programmer, which will be displayed to and can be configured by the end-user, in cases where required and optional properties defined by the acquisition2 and perturbation2 frameworks do not suffice.

The method **getProperties()** of **OscilloscopeSession** returns an instance of **OscilloscopeProperties**. This interface represents and defines general acquisition properties that the acquisition2 and perturbation2 frameworks can configure. The properties and their related methods are shown in Table .

Table 9.9. The properties that can be configured on a channel via the OscilloscopeProperties interface

| Property | Type | Associated methods |
|----------|------|--------------------|
|----------|------|--------------------|

| | | |
|-----------------|--------|--|
| numberOfSamples | int | long getMaxNumberOfSamples() int getNumberOfSamples() void setNumberOfSamples(int samples) |
| timePerSample | double | boolean isFreeformTimebaseSupported() List<Double> getSupportedTimesPerSample() double getTimePerSample() void setTimePerSample(double s) |

The OscilloscopeSession implementation should also indicate which channels are present on the physical scope. The method `getChannels()` returning a `List<Channel>` is used for this purpose. The Channel interface has a `getChannelProperties()` method which should return an instance of `ChannelProperties`. Through this instance, the acquisition2 and perturbation2 frameworks can configure the properties of a channel shown in this table.

Table 9.10. The properties that can be read and/or configured on a channel via the ChannelProperties interface.

| Property | Type | Associated methods |
|-------------------|----------|--|
| enabled | boolean | boolean isEnabled() void setEnabled(boolean enabled) |
| triggerChannel | boolean | boolean isTriggerChannel() |
| dataChannel | boolean | boolean isDataChannel() |
| coupling | Coupling | boolean isCouplingSupported() List<Coupling> getSupportedCoupings() Coupling getCoupling() void setCoupling(Coupling coupling) |
| inputImpedance | double | boolean isInputImpedanceSupported() boolean isFreeformInputImpedanceSupported() List<Double> getSupportedInputImpedances() double getInputImpedance() void setInputImpedance(double impedance) |
| maxInputFrequency | double | boolean isMaxFrequencySupported() double getMaxInputFrequency() void setMaxInputFrequency(double frequency) |
| offset | double | boolean isOffsetSupported() double getOffset() void setOffset(double offset) |
| probeAttenuation | double | boolean isProbeAttenuationSupported() double getProbeAttenuation() void setProbeAttenuation(double attenuation) |
| range | double | boolean isRangeSupported() boolean isFreeformRangeSupported() List<Double> getSupportedRanges() double getRange() void setRange(double range) |

| | | |
|------------|-----|---|
| resolution | int | <code>boolean isResolutionSupported() List<Integer> getSupportedResolutions() int getResolution() void setResolution(int resolution)</code> |
|------------|-----|---|

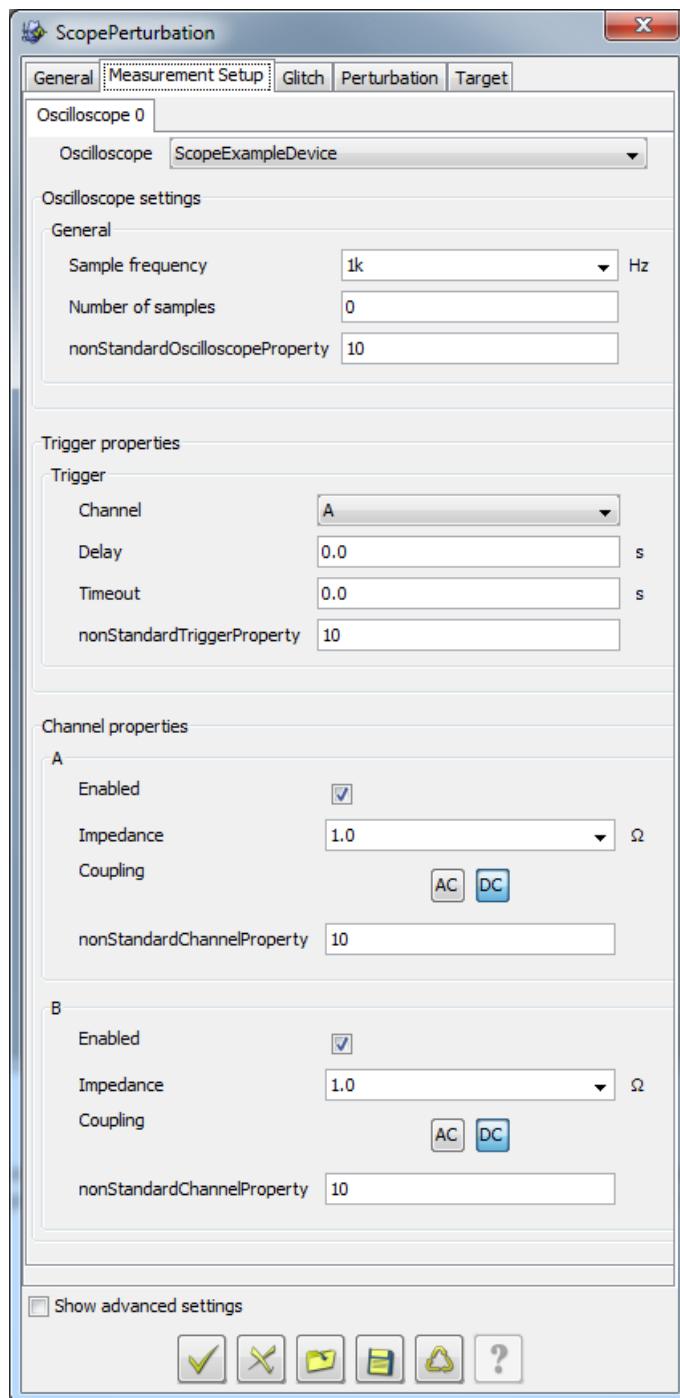
The **OscilloscopeSession** implementation should finally also indicate whether or not the oscilloscope supports triggered measurement. If the scope supports triggering, the **getTrigger()** method of **OscilloscopeSession** should return an instance of **TriggerProperties** and otherwise **null**. The trigger is configured through the **TriggerProperties** instance returned by **Trigger's getTriggerProperties()** method. The properties that can be configured through this interface are shown in Table .

Table 9.11. The properties that can be read and/or configured to trigger a measurement commencement via the TriggerProperties interface

| Property | Type | Associated methods |
|--------------------|--------------|---|
| triggerChannelName | String | <code>List<String> getSupportedTriggerChannelNames() String getTriggerChannelName() void setTriggerChannelName(String channelName)</code> |
| triggerTimeout | double | <code>double getTriggerTimeout() void setTriggerTimeout(double timeout)</code> |
| triggerDelay | double | <code>boolean isTriggerDelaySupported() double getTriggerDelay() void setTriggerDelay(double delay)</code> |
| triggerLevel | double | <code>boolean isTriggerLevelSupported() double getTriggerLevel() void setTriggerLevel(double level)</code> |
| triggerSlope | TriggerSlope | <code>boolean isTriggerEdgeSupported()Trigger Slope getTriggerSlope() void setTriggerSlope(TriggerSlope slope)</code> |

The instances of **OscilloscopeProperties**, **TriggerProperties** and **ChannelProperties** are also allowed to contain non-standard properties. This is useful when the properties listed in Table , Table and Table do not cover the feature set of the physical oscilloscope. Custom properties should be implemented in the aforementioned instances as bean properties. Section What is a JavaBean and what is a bean property? describes how a bean property is defined.

Figure shows how the **acquisition2** framework displays the instances of **OscilloscopeProperties**, **TriggerProperties** and **ChannelProperties** respectively. All the instances that are being displayed contain one non-standard property. In this example the method **getChannels()** returns a **List** containing two **Channels** named A and B.

Figure 9.24. Configuring a scope in the acquisition2 framework

Instances of **OscilloscopeProperties**, **TriggerProperties** and **ChannelProperties** are shown from top to bottom. These instances all contain one non-standard property.

The acquisition2 and perturbation2 frameworks will manage the persistence of the object instances returned by the methods **getProperties()**, **getChannelProperties()** and **getTriggerProperties()** of the **OscilloscopeSession**, **Channel** and **Trigger** interfaces respectively. This means that the objects returned by these methods should be implemented as one would implement a JavaBean (see section What is a JavaBean and what is a

bean property?), in the sense that the class should have a public default constructor. In practical terms it means that an implementation of for example `setTimePerSample(...)` of the **OscilloscopeProperties** interface should avoid configuring the scope directly in this call and that for example the method `getSupportedTimesPerSample()` in the same interface should avoid obtaining relevant information directly from the scope, because in order to do so it would need a reference to an object that can probably not be persisted (like an **OutputStream**) and that can therefore not be passed to the object instance during its construction.

Therefore it is recommended to implement the `get*Properties()` methods as demonstrated for `getProperties()` of the **OscilloscopeSession** interface.

Example 9.37. The recommend pattern to follow when implementing `get*Properties()` methods

```
package com.riscure.demos.hardware.scope;

import com.riscure.hardware.scope.OscilloscopeProperties;
import com.riscure.hardware.scope.OscilloscopeSession;

// <code omitted>

public class ScopeExampleSession implements OscilloscopeSession {
    // <code omitted>
    private ScopeExampleProperties properties;
    // <code omitted>
    @Override
    public OscilloscopeProperties getProperties() {
        boolean firstCall = this.properties == null;
        if(firstCall) {
            this.properties = new ScopeExampleProperties();
        }

        List<Double> supportedTimesPerSample;

        // ... <optional code to obtain information from physical device>
        // Call a private setter after obtaining information from physical device.

        this.properties.supportedTimesPerSample(supportedTimesPerSample);
    }
    return this.properties;
}

@Override
public void setProperties(Object props) {
    this.properties = (ScopeExampleProperties) props;
}

// <code omitted>
}
```

Performing an acquisition

When the acquisition2 and perturbation2 frameworks are about to perform an acquisition, the method `acquire()` in the **OscilloscopeSession** interface will be called. This method returns an instance of **Acquisition**, which represents an (on-going) acquisition. In essence relates to **OscilloscopeSession** in the same way as `open(...)` in the DeviceInterface interface relates to DeviceInterface: in both instances an action has to be undertaken according to the settings

defined by the user. Therefore the skeleton implementations of these methods are very similar. This is shown in the next example which should be compared with the previous. The differences between these Code Listings stem from the fact that an **Acquisition** represents an on-going action whereas a **Session** does not. This means that the implementation should satisfy the following conditions:

1. Return an instance of **Acquisition** as soon as possible.
2. As long as the action has not yet been completed, **isReady()** in the **Acquisition** instance should return **false**. Combined with condition 1 return an instance of Acquisition as soon as possible. It means that the constructor of the **Acquisition** instance should not wait until the (physical) acquisition has been completed.
3. The method **getChannelAcquisition(...)** in the **Acquisition** instance can be called by the acquisition2 and perturbation2 frameworks before **isReady()** returns **true** and therefore the implementation of this method should contain a busy-wait loop.

The method **getChannelAcquisition()** in the **Acquisition** interface will be with Channel instance returned in **getChannels()** that the user enabled through the instance's **ChannelProperties**. This method returns an instance of **ChannelAcquisition**. This interface represents the measurement done on one of the channels of the physical scope. The implementation of this interface has no special considerations and is easily implemented using the JavaDoc documentation of **ChannelAcquisition**.

```
package com.riscure.demos.hardware.scope;
import com.riscure.hardware.scope.Acquisition;
import com.riscure.hardware.scope.OscilloscopeSession;

// <code omitted>

public class ScopeExampleSession implements OscilloscopeSession {
    // <code omitted>
    private ScopeExampleAcquisition latestAcquisition;
    // <code omitted>
    @Override
    public Acquisition acquire() throws IOException {
        boolean scopeHasOngoingAcquisition = ( this.latestAcquisition != null &&
        !this.latestAcquisition.isReady(); );
        if(scopeHasOngoingAcquisition) {
            throw new IOException("This scope already has an on-going acquisition");
        } else {
            // (1) Return as soon as possible
            this.latestAcquisition = new ScopeExampleAcquisition(
                // ... <object(s) required to communicate with device>
                , this.getProperties(), this.getChannels(), this.getTrigger() );
        }
        return this.latestAcquisition;
    }
}

package com.riscure.demos.hardware.scope;
import java.io.IOException;
import java.util.List;
import com.riscure.hardware.scope.Acquisition;
import com.riscure.hardware.scope.Channel;
import com.riscure.hardware.scope.ChannelAcquisition;
import com.riscure.hardware.scope.OscilloscopeProperties;
import com.riscure.hardware.scope.Trigger;
```

```
public class ScopeExampleAcquisition implements Acquisition {

    private static final int DEFAULT_TIMEOUT_MS = 5000;
    private static final int WAIT_TIME_MS = 1000;
    private boolean ready;
    private int timeout_ms = DEFAULT_TIMEOUT_MS;
    public ScopeExampleAcquisition( // ... <object(s) required to communicate with
        device>
        , OscilloscopeProperties properties, List<Channel> channels, Trigger trigger ) {
        // ... <code to configure the scope according to user configuration>
        // (1) Return as soon as possible
        new Thread(new Runnable() {
            @Override
            public void run() {
                // ... <code that blocks or busy-waits until the scope has acquired>
                // (2) ready can now be set to true
                ready = true;
            }
        }).start();
    }

    @Override
    public boolean isReady() {
        return this.ready;
    }

    @Override
    public void abort() throws IOException {
        // ... <code to abort the acquisition>
    }

    @Override
    public int getTimeout() {
        return this.timeout_ms;
    }

    @Override
    public void setTimeout(int timeout) {
        this.timeout_ms = timeout;
    }

    @Override
    public ChannelAcquisition getChannelAcquisition(Channel channel) throws IOException
    {
        // (3) busy-wait until isReady() returns true or timeout limit has been reached
        long systemEndTime_ns = System.nanoTime() + (this.getTimeout() * 1000 * 1000);
        while (!this.isReady()) {
            try {
                boolean timeoutExceeded = systemEndTime_ns < System.nanoTime();
                if (timeoutExceeded) {
                    throw new InterruptedIOException( "Oscilloscope failed to complete acquisition
in time." );
                }
                Thread.sleep(WAIT_TIME_MS);
            } catch (InterruptedException e) {
                throw new IOException(e);
            }
        }
    }
}
```

```
    return new ScopeExampleChannelAcquisition( // ... <object(s) required to
        communicate with device>
        , channel );
}
```

9.7.9 How do I implement the **IODeviceSession** interface?

This section picks up from Section 9.7.6, “How do I implement a session on a device?” assuming that the `open(...)` method from that Code Listing returns an instance of an implementation of **IODeviceSession** called **IODeviceExampleSession**.

The **IODeviceSession** demands far fewer methods to be implemented than the **OscilloscopeSession** interface. Despite this, it has essentially the same structure, because the interface also contains two similar sections, namely a:

1. preparation section: The **getProperties()** and **setProperties(...)** methods of the **IODeviceSession** interface are used by the **acquisition2** and **perturbation2** frameworks to allow a user to prepare the target device for communication. The **getProperties()** method should return an instance of JavaBean that is compatible with Inspector's AGG. The **setProperties(...)** method is used to restore user settings for a target device from persistence.
2. communication section: The **connect()** method should return an instance of **IODeviceConnection** that is configured according to the settings that can be obtained from the object returned by **getProperties()**. It is analogous to the **acquire()** method in **OscilloscopeSession**. The instance returned by this method is used by the acquisition2 and perturbation2 frameworks to communicate with the target device / target of interest through an application protocol (like EMV, OTA, USIM, etc.).

For most devices this method should cause a 'power up' of/to the actual target, while calling `close()` on the connection will cause it to power down again.

This instance of **IODeviceConnection** should in some way implement the communication protocol related to the physical device / medium. In other words, the payload that is sent through the **command(...)** method of the **IODeviceConnection** should deliver that payload as the communication protocol of the physical medium demands it. For example the PowerTracer implementation of **IODeviceConnection**, called **PowerTracerConnection** communicates with the target of interest (a contact card) through the ISO 7816 communication protocol (using Riscure's communication protocol library) and the MP300 implementation of **IODeviceConnection**, called **MP300Connection**, communicates with the target of interest (a contactless card) through the ISO 14443 communication protocol (also using Riscure's communication protocol library).

```
package com.riscure.demos.hardware.iodevice;

import java.io.IOException;
import com.riscure.hardware.io.IIODeviceConnection;
import com.riscure.hardware.io.IIODeviceSession;

public class IIODeviceExampleSession implements IIODeviceSession {
    // <code omitted>
```

```
// (1) Preparation / configuration
private IODeviceExampleProperties properties = new IODeviceExampleProperties();

// (2) Communication
private IODeviceExampleConnection latestConnection;

// <code omitted>

// (1) Preparation / configuration
@Override
public Object getProperties() {
    return this.properties;
}

// (1) Preparation / configuration
@Override
public void setProperties(Object props) {
    this.properties = (IODeviceExampleProperties) props;
}

// (2) Communication
@Override
public IODeviceConnection connect() throws IOException {
    boolean hasActiveConnection = ( this.latestConnection != null
&& this.latestConnection.isConnected() );
    if(hasActiveConnection) {
        throw new IOException( "This device does not support simultaneous connections" );
    } else {
        this.latestConnection = new IODeviceExampleConnection( // ... <object(s) required
to communicate with device>
            , (IODeviceExampleProperties) this.getProperties() );
    }
    return this.latestConnection;
}
}

package com.riscure.demos.hardware.iodevice;
import java.io.IOException;
import com.riscure.hardware.io.IODeviceConnection;
import com.riscure.protocol.CommandData;
import com.riscure.protocol.CommandResponse;

public class IODeviceExampleConnection implements IODeviceConnection {
    private boolean connected;
    private IODeviceExampleProperties properties;
    public IODeviceExampleConnection(
        // ... <object(s) required to communicate with device>
        , IODeviceExampleProperties properties ) throws IOException {
        this.properties = properties;
        this.reset();
    }

    @Override
    public boolean isConnected() {
        return this.connected;
    }

    @Override
    public void close() throws IOException {
        // ... <code to close connection on device>
```

```
this.connected = false;
}

@Override
public void reset() throws IOException {
// ... <code to open connection on device if not open>
this.connected = true;
}

// (2) Communication
@Override
public CommandResponse command(CommandData command) throws IOException {
CommandResponse response;
// ... <code that :
// 1. Wraps payload in CommandData according to communication protocol,
// 2. Sends wrapped payload to device and
// 3. Unwraps the data returned by the device >
return response;
}
}
```

Skeleton implementation of **IODeviceSession** and **IODeviceConnection**.

9.8 How do I implement an acquisition2/perturbation2 application protocol?

The **acquisition2** and **perturbation2** frameworks allow a user to select an application protocol for a measurement or fault injection attempt. These frameworks assume that application protocols are two-party protocols, where the target device represents one party of the protocol and a protocol implementation represents the second party. One party is represented by the **ProtocolTarget** interface whilst the other party is represented by the **Protocol** interface. The **ProtocolTarget** interface also serves as an abstraction layer over the physical device that is being communicated with during a measurement or fault injection attempt, and as such implementations of this interface take care of the transport protocol belonging to the physical medium that it represents. The **Protocol** interface is the initiating party in the protocol. In order to define a new application protocol, only the **Protocol** interface needs to be implemented and this section is dedicated to explaining how to do that.

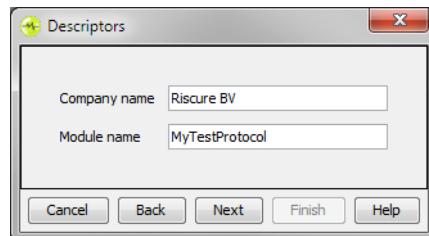
How do I register an application protocol in the acquisition2 and perturbation2 frameworks?

1. Implement a class that implements the **com.riscure.signalanalysis.acquisition.Protocol** interface and make sure its class file and any related dependencies are in Inspector's system or user modules folder.
2. Use the Module Wizard to create a simple Protocol class. If necessary, you can then add code to this class. This approach is explained next.

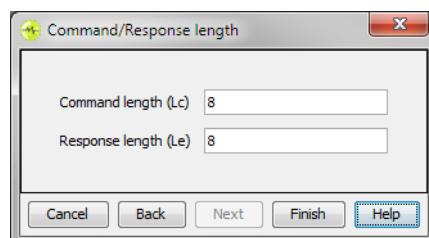
9.8.1 The Application Protocol Wizard

In the "File" menu of Inspector there is the "New Module Wizard..." option. This wizard offers the possibility to design a "Protocol". Basically two steps have to be performed:

1. First the "Descriptors" have to be entered:

Figure 9.25. Entering the descriptors in the Protocol Wizard

2. After clicking the "Next" button you have to enter the command length and the (expected) response length in bytes:

Figure 9.26. Entering the command and response lengths in the Protocol Wizard

This will produce the following boilerplate code:

```
package acquisition2.protocol.trainingcard;

import java.io.IOException;

import static com.riscure.signalanalysis.data.SimpleVerdict.*;
import com.riscure.io.Direction;
import com.riscure.osgi.legacy.Service;
import acquisition2.protocol.BasicProtocol;

@Service("MyTestProtocol")
public class MyTestProtocol extends BasicProtocol {

    /**
     * Creates a protocol with input length 8 bytes and expected output
     * length 8 bytes.
     */
    public MyTestProtocol() {
        super(8, 8);
        addPhase("Select command");
        addPhase("Crypto command");
    }

    @Override
    protected void init() throws IOException {
        phase("Select command");
        //Change the byte command below:
        command("00 a4 04 00 07 A0 00 00 FF F0 01");
    }
}
```

```
@Override
protected void run() throws IOException {
    phase("Crypto command");
    byte[] cmd = randomize(
        //Change the byte command below:
        "A0 04 03 00 10 C7 39 D7 EA FA E4 ED A3 C7 39 D7 EA FA E4 ED A3 00",
        5, this.getInputLength());
    byte[] response = command(cmd);

    // Store the incoming data (command) and the outgoing response
    addDataIn(cmd, 5, this.getInputLength());
    addDataOut(response, 0, this.getOutputLength());

    this.verdict(NORMAL);
}
}
```

The **@Service()** annotation declares the user-friendly name of the protocol; it's this name that will be displayed in the protocol dropdown list in the module. You can change the name to something like "Test Protocol", if you prefer.

As you see, this class extends the **BasicProtocol** class. It is this super class which does most of the work. The most important functions of the utility class **BasicProtocol** are the following:

- **void phase(String id)**: Set the next trigger phase to the specified phase name. The phases are compared by "id", which per default is equal to their "name".
- **byte[] command(String cmd)**: Send the specified command string tot the target. This method returns the response as a byte array.
- **byte[] command(byte[] cmd)**: The same as above, but with the command represented as a byte array.
- **void addDataIn(byte[] data, int offset, int length)**: Add the given command bytes to the list of incoming data. Note that "In" indicates the direction of communication as seen from the target. So this method should be used to store a command string in the trace. Note that the last two parameters are optional when you want to store the complete byte array. Note that the length should not exceed the input length that was specified in the constructor (**super(8, 8)** in this example).
- **void addDataOut(byte[] data, int offset, int length)**: Add the given response data bytes to the list of data. This is outgoing data for the target. The last two parameters are optional. Note that the length should not exceed the expected output length that was specified in the constructor.
- **byte[] randomize(String cmd, int offset, int length)**: Fill part of the byte command with values produced by the data generator, which will probably be either random or "fixed value". The **offset** indicates the start of the data section, while **length** refers to the number of bytes that will be copied into the command byte array.
- **byte[] randomize(byte[] cmd, int offset, int length)**: The same as above, but with the command represented as a byte array.
- **List<Phase> getPhases()**: Give a list of the phases that are registered in this protocol.
- **void verdict(Verdict verdict)**: Set the verdict to a specific value. Typical options are **SimpleVerdict.INCONCLUSIVE**, **SimpleVerdict.NORMAL**, and **SimpleVerdict.SUCCESSFUL**.

In the constructor of **MyTestProtocol** we see that the super class constructor is called, setting the input length and the output length to 8. You can use the methods **getInputLength()** and **getOutputLength()** from now on, as is shown in the **run()** method. The calls to **addPhase(...)** populate the list of phases that will be returned by **getPhases()**.

The **init()** method basically sets the phase to "Select" and sends out the command string. If your card requires a different select command then you can change it here.

In the **run()** method some very basic work is done, but quite possibly this is enough for most simple protocols. In that case you only have to change the byte command string. On the first line the "Crypto" phase is set. Then the **randomize()** function is called in order to fill the Data part with appropriate values. Make sure that you replace this string with your command. The command string is then sent to the target via the **command()** method. The response is a byte array. Then we add the (outgoing) command data to the list of communication data. Note that you have to specify the start of the data section (5) and the length of the data (8). The same goes for the (incoming) response data that starts at index 0 and has 8 bytes (followed by "90 00").

In this example the **verdict(...)** method is called with value **SimpleVerdict.NORMAL**. In a practical scenario there should be a real test to determine the verdict. Such a test could look like the following code:

```
if (HexUtils.endsWith(response, "90 00")) {  
    verdict(NORMAL);  
} else {  
    verdict(SUCCESSFUL);  
}
```

Note that you need an **import com.riscure.util.HexUtils** command in order for the above lines to work.

Press the "Compile" icon at the Inspector toolbar. It will ask for a file location. After saving the file you can start one of the acquisition2 modules and already select your new protocol.

The following sections show how a more sophisticated protocol can be programmed from the ground up. However, java code that the wizard produces (in combination with the utility class **acquisition2.protocol.trainingcard.BasicProtocol**) can easily be extended to achieve the same functionality. For instance, **BasicProtocol** already has **get/setSettingsBean()** functions, so you can easily add your own settings (as described below). You can also override methods (like **addData()** or **getPhases()**) for alternative behavior.

9.8.2 How do I allow an end user to configure the protocol for an acquisition?

The **acquisition2** and **perturbation2** frameworks allow a user to configure two aspects of a protocol:

1. General protocol settings: If the protocol has general settings that need to be configured for one 'run' of the protocol, the **getSettingsBean()** method of **Protocol** should return a JavaBean that can be rendered by Inspector's AGG (see chapter How do I implement an AGG compatible JavaBean? for more details). Otherwise, it should return null. The **setSettingsBean(...)** method is used by the **acquisition2** and **perturbation2** frameworks to restore settings from persistence. The implementation of this method can assume that the argument to it is has the same type as the object returned by **getSettingsBean()**.

General protocol settings are for example the AID of an applet that needs to be selected on a smart card or an encryption/decryption key that needs to be used during the protocol.

1. The trigger moment on which to arm measurement and/or fault injection devices: The trigger moments are defined by the method `getPhases()` of the **Protocol** interface, which should return a `List<Phase>`. The Phase interface defines a single trigger moment and this method is closely associated to the `run(...)` method of **Protocol** which will be discussed in section How do I implement the protocol execution?.

An example of an interesting trigger moment could be the moment in which an encryption or decryption operation is about to be performed by the target in the protocol.

the next code demonstrates an example implementation of the `getSettingsBean()`, `setSettingsBean(...)` and `getPhases()` methods described above. The following figure shows how the **acquisition2** framework displays the general protocol settings and trigger moment.

```
package com.riscure.demos.protocol;
import java.util.List;
import com.google.common.collect.ImmutableList;
import com.riscure.signalanalysis.acquisition.Phase;
import com.riscure.signalanalysis.acquisition.Protocol;

// <code omitted>

public class ExampleProtocol implements Protocol {
    // (1) General protocol settings private ExampleSettingsBean settingsBean;
    @Override
    public Object getSettingsBean() {
        return this.settingsBean;
    }

    @Override
    public void setSettingsBean(Object settings) {
        this.settingsBean = (ExampleSettingsBean) settings;
    }

    // (2) Possible trigger moments
    @Override
    public List<Phase> getPhases() {
        return ImmutableList.of( (Phase) new ExamplePhase("ExamplePhase 1"),
                               (Phase) new ExamplePhase("ExamplePhase 2"),
                               (Phase) new ExamplePhase("ExamplePhase 3") );
    }

    // <code omitted>
}

package com.riscure.demos.protocol;
import com.riscure.beans.annotation.DisplayName;
import com.riscure.beans.annotation.Presentation;
// (1) General protocol settings

public class ExampleSettingsBean {
    @DisplayName("Key")
    @Presentation(base=16)
    private byte[] key = new byte[] { (byte)0xCA, (byte)0xFE, (byte)0xBA, (byte)0xBE };
```

```
public byte[] getKey() {
    return this.key;
}

public void setKey(byte[] key) {
    this.key = key;
}
}

package com.riscure.demos.protocol;
import com.riscure.signalanalysis.acquisition.Phase;
// (2) Possible trigger moment

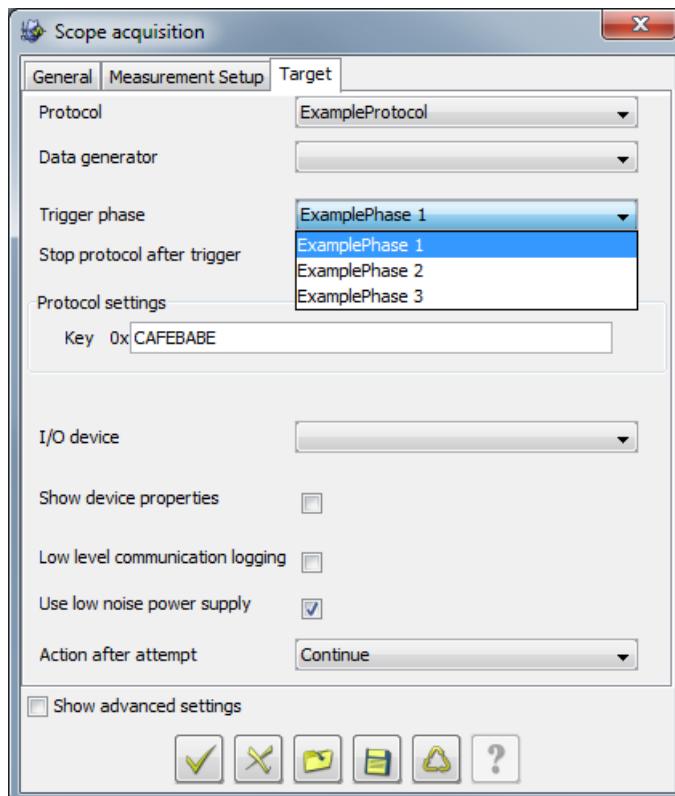
public class ExamplePhase implements Phase {
    private String id;
    public ExamplePhase(String id) { this.id = id; }

    @Override
    public String getName() {
        return this.getPhaseId();
    }

    @Override public String getPhaseId() {
        return this.id;
    }
}
```

An example implementation of `getSettingsBean()`, `setSettingsBean(...)` and `getPhases()` of the **Protocol** interface.

Figure 9.27. The acquisition2 framework displaying protocol settings



9.8.3 How do I implement the protocol execution?

A protocol is expected to run in two stages, namely a preparation stage and an execution stage. The preparation stage can for example be used to select an applet on a smart card while the execution stage communicates, according to the protocol with the applet that was selected in the preparation stage. These stages are represented by the **init(...)** and **run(...)** methods of the **Protocol** interface respectively.

These two methods take as an argument an object of type **ProtocolTarget**. As was mentioned before, the **ProtocolTarget** object represents the non-initiating party in a two-party protocol. Therefore the **ProtocolTarget** interface has the following methods to receive a message from and send a response to the initiating party:

1. **Response send(Command command, Phase phase)**
2. **Response send(Command command, Phase phase, boolean protocolErrorException)**

The first method is a convenience overload that passes true as the third argument to the second method. An overview of the arguments and their meanings is given in Table .

Table 9.12. ProtocolTarget.send(...) argument overview

| Argument | Type | Meaning |
|------------------------|---------|---|
| command | Command | The message/command to send to the other partner in the protocol. |
| phase | Phase | The phase (trigger moment) to which the command belongs. |
| protocolErrorException | boolean | A flag that determines whether or not an exception should be thrown by the send(...) method when a protocol error occurs. |

The **init()** and **run()** methods are expected to call the **send(...)** method of **ProtocolTarget** until the initialisation and execution stages respectively have been completed. This requires an implementation of the **Command** and **Respond** interfaces.

The **Command** interface defines two methods:

1. **getCommand()** which should return a **byte[]**. This array is the message that should be sent to over the physical medium that the **ProtocolTarget** instance represents.
2. **parseResponse(byte[])** which returns an instance of the **Response** interface. This method is supposed to parse the payload of the response that was received by the physical device after sending the result of the **getCommand()** method to it.

The Response interface defines three methods:

1. **getResponse()** which should return a **byte[]**. This array is should be the payload of the response at the application protocol level.
2. **isValid()** which should return whether or not the response is valid within the rules of the application protocol.
3. **assertValid()** which should throw a **ProtocolException** if **isValid()** returns **false**;

The **acquisition2** and **perturbation2** frameworks define the classes **DirectCommand** and **DirectResponse** in the package **acquisition2.protocol.command**. These classes are straightforward implementation of the **Command** and **Response** interfaces respectively, that simply wrap a **byte[]**.

It is important to realise that the **Phase** argument is used to arm all devices involved in the acquisition / glitch attempt as follows: the end-user selects a trigger for an attempt by choosing one of the **Phases** provided by **getPhases()**. **ProtocolTarget.send(...)** will arm all involved devices when the phase argument that matches the trigger moment selected by the end-user is passed into it for the first time (and not for any of the subsequent times) during an acquisition / glitch attempt. This means that

1. **Phases** should not be used to build a state machine that can move into one state multiple times during protocol execution, as the second and subsequent times this 'state' transition occurs cannot be used by the end-user to trigger a measurement / glitch.
2. **Protocol.init(...)** should not call **ProtocolTarget.send(...)** with a **Phase** that has the same ID as a **Phase** that can be obtained from **Protocol.getPhases()**, as this will cause an arm of all related measurement and glitch devices.

One final consideration is that

1. **Protocol.init(...)** and **Protocol.run(...)** should not catch **Throwable** or **Error**, as the **acquisition2** and **perturbation2** frameworks use a subclass of **Error** to force a break out of **Protocol.run(...)** when the user wants the protocol to abort after the trigger moment.

The next code is an example of **Protocol.init(...)** and **Protocol.run(...)** that obeys to the three rules given above.

```
package com.riscure.demos.protocol;
import java.io.IOException;
import java.util.List;
import com.google.common.collect.ImmutableList;
import com.riscure.signalanalysis.acquisition.Protocol;
import com.riscure.signalanalysis.acquisition.ProtocolTarget;
import com.riscure.signalanalysis.data.CommunicationData;

public class ExampleProtocol implements Protocol {
    // <code omitted>
    @Override
    public void init(ProtocolTarget target) throws IOException {
        target.send( new ExampleCommand( new byte[] { (byte) 0xCA } ),
                    new ExamplePhase("ExamplePhase 0") );
    }

    @Override
    public void run(ProtocolTarget target) throws IOException {
        target.send( new ExampleCommand( new byte[] { (byte) 0xFE } ),
                    new ExamplePhase("ExamplePhase 1") );
        target.send( new ExampleCommand( new byte[] { (byte) 0xBA } ),
                    new ExamplePhase("ExamplePhase 2") );
        target.send( new ExampleCommand( new byte[] { (byte) 0xBE } ),
                    new ExamplePhase("ExamplePhase 3") );
    }
    // <code omitted>
}
```

9.8.4 How do I implement the communication log according to the expectation of the acquisition2 and perturbation2 frameworks?

The method `getData()` of the `Protocol` interface should return a list of `CommunicationData`. It is expected that this list follows the protocol execution. This is demonstrated in the next code which enables a straightforward implementation of `getData()`. This code makes use of `acquisition2.lib.BinaryCommunicationData`, a straightforward implementation of the `CommunicationData` interface

```
package com.riscure.demos.protocol;

import java.io.IOException;
import java.util.List;
import acquisition2.lib.BinaryCommunicationData;
import com.google.common.collect.ImmutableList;
import com.google.common.collect.Lists;
import com.riscure.io.Direction;
import com.riscure.signalanalysis.acquisition.Phase;
import com.riscure.signalanalysis.acquisition.Protocol;
import com.riscure.signalanalysis.acquisition.ProtocolTarget;
import com.riscure.signalanalysis.data.CommunicationData;

public class ExampleProtocol implements Protocol {
    // <code omitted>
    private List<CommunicationData> communicationData = Lists.newArrayList();
    // <code omitted>
    @Override
    public void init(ProtocolTarget target) throws IOException {
        this.sendData( target, new ExampleCommand( new byte[] { (byte) 0xCA } ),
                      new ExamplePhase("ExamplePhase 0") );
    }

    @Override
    public void run(ProtocolTarget target) throws IOException {
        this.sendData( target, new ExampleCommand( new byte[] { (byte) 0xFE } ),
                      new ExamplePhase("ExamplePhase 1") );
        this.sendData( target, new ExampleCommand( new byte[] { (byte) 0xBA } ),
                      new ExamplePhase("ExamplePhase 2") );
        this.sendData( target, new ExampleCommand( new byte[] { (byte) 0xBE } ),
                      new ExamplePhase("ExamplePhase 3") );
    }

    @Override
    public List<CommunicationData> getData() {
        List<CommunicationData> commData = ImmutableList.copyOf(this.communicationData);
        this.communicationData.clear(); return commData;
    }

    private void sendData( ProtocolTarget target, ExampleCommand command, ExamplePhase phase )
        throws IOException {
        ExampleResponse response = (ExampleResponse) target.send(command, phase);
        this.addData(command); this.addData(response);
    }

    private void addData(ExampleCommand command) {
```

```
        this.communicationData.addData( new BinaryCommunicationData( "OUT",
                Direction.OUT,
                command.getCommand() ) );
    }

    private void addData(ExampleResponse response) {
        this.communicationData.addData( new BinaryCommunicationData( "IN",
                Direction.IN,
                response.getResponse() ) );
    }
}
```

9.8.5 How should I obtain input data for my protocol?

If a **Protocol** implementation implements the **DataConsumer** interface, the **acquisition2** and **perturbation2** frameworks will supply an instance of **DataProvider** to the **Protocol** before its **run(...)** method is called. This instance of **DataProvider** is supplied via the **setDataProvider(...)** method of the **DataConsumer** interface. Data is obtained from the **DataProvider** by calling its **getBytes(...)** method. The first argument to this method is an ID specifying what the data is used for, while the second argument specifies the number of bytes the **DataProvider** should generate/provide.

```
package com.riscure.demos.protocol;
import java.io.IOException;
import com.riscure.signalanalysis.acquisition.DataConsumer;
import com.riscure.signalanalysis.acquisition.DataProvider;
import com.riscure.signalanalysis.acquisition.Protocol;
import com.riscure.signalanalysis.acquisition.ProtocolTarget;

// <code omitted>

public class ExampleProtocol implements Protocol, DataConsumer {

    // <code omitted>

    private DataProvider dataProvider;

    // <code omitted>

    @Override
    public void run(ProtocolTarget target) throws IOException {
        this.sendData( target, new ExampleCommand( this.dataProvider.getBytes("run()/1",
        4) ),
            new ExamplePhase("ExamplePhase 1") );
        this.sendData( target, new ExampleCommand( this.dataProvider.getBytes("run()/2",
        4) ),
            new ExamplePhase("ExamplePhase 2") );
        this.sendData( target, new ExampleCommand( this.dataProvider.getBytes("run()/3",
        4) ),
            new ExamplePhase("ExamplePhase 3") );
    }

    @Override
    public void setDataProvider(DataProvider dataProvider) {
        this.dataProvider = dataProvider;
    }
    // <code omitted>
}
```

9.8.6 How can I close resources opened by my protocol at the end of an acquisition or perturbation?

If a **Protocol** implements the `java.io.Closeable` interface, the frameworks will call this method at the end of an acquisition or perturbation (either because all measurements have been performed, the user has aborted the measurement or because an unrecoverable error has occurred). A simple example of opening a file in the `init()` method of the Protocol and closing it in the `close()` method of the Protocol is shown below.

```
package com.riscure.demos.protocol;
import java.io.IOException;
import java.io.Closeable;
import java.io.FileReader;
import com.riscure.signalanalysis.acquisition.DataConsumer;
import com.riscure.signalanalysis.acquisition.DataProvider;
import com.riscure.signalanalysis.acquisition.Protocol;
import com.riscure.signalanalysis.acquisition.ProtocolTarget;

// <code omitted>

public class ExampleProtocol implements Protocol, DataConsumer, Closeable {

    private FileReader inputStream;
    // <code omitted>

    @Override
    public void init(ProtocolTarget target) throws IOException {
        inputStream = new FileReader("path/to/file");
    }

    @Override
    public void close() throws IOException {
        if(inputStream != null) {
            inputStream.close();
        }
    }
    // <code omitted>
}
```

9.9 How do I implement an acquisition2/perturbation2 raw protocol?

A **RawProtocol**, unlike a **Protocol**, acts directly on the raw I/O device. The advantage is that there is direct control over the target. The disadvantage is that the system provides no support for determining the length of responses, powering or resetting the target. This gives the user a lot of freedom. We will first describe the basic developing work for creating a **RawProtocol** implementation. Then we present an example with a warm boot glitching attack on the XMEGA training target (as used in the tutorials).

9.9.1 Developing a Raw Protocol

Developing a RawProtocol is very similar to developing a Protocol as described in Section 9.8, "How do I implement an acquisition2/perturbation2 application protocol?". The easiest way

to start is to extend BasicRawProtocol and implement the `/run/` method. Instead of the `send` method in `ProtocolTarget`, `RawProtocolTarget` provides `writeBytes` and `readBytes`. In addition the method `checkAvailableBytes` returns the number of bytes in the read buffer. The number of bytes returned is the minimum number of bytes which can be read without blocking. Depending on the implementation of the specific raw I/O device this may return the actual number of bytes in the buffer or zero always. Therefore a zero returned by `checkAvailableBytes` does not indicate that a `readBytes` will not return anything.

The `RawProtocol` is also responsible for powering and resetting the device. For this purpose the `powerUp`, `powerDown`, and `reset` methods are provided.

Phases work in the same way as they do in `Protocol`. Input generators and user settings are currently not supported.

9.9.2 Embedded Boot Glitching With a Warm Reset

To illustrate the use of a raw protocol in practice we demonstrate a warm boot glitching attack on an embedded target. The scenario is as follows. We have an embedded target (the XMEGA device) and we want to glitch it after reset. Because a cold reset (i.e. cutting off the VCC line) generally takes more time than a warm reset (i.e. asserting the reset line) we would like to save measurement time by using a warm reset between runs.

First something about the module that we are going to use. From the Inspector menu we select "Perturbation" -> "Voltage/Clock" -> "Embedded" -> "Raw". The title of this module is "EP Perturbation after Reset", because that is the typical use case. However, the class name "RawEmbeddedPerturbation" actually describes it better. On the Embedded Glitch Setup tab we select the VC Glitcher as glitch device and let the "Reset and Trigger Source" be a "Glitch after reset". On the Target tab we have to select the Protocol. The supplied protocol XMEGA_ReadAll works perfectly fine for *cold* boot glitching, but since we want a warm reset we will have to write our own. More on that later. The "Trigger phase" (which actually means the phase where the VC Glitcher will be *armed*) has to be set to "Before power up". This is because for convenience in the code we will still make use of the standard POWER_UP phase, also for the reset operation.

As I/O device we have to create a composite device. This can be done from the Hardware Manager (under Tools menu option). For all three "underlying" devices (IO, power, and reset) we select the VC Glitcher. The power up time, power down time, and reset time make Inspector sleep after pulling down the VCC line or asserting the reset line, so don't set these values too high because they add up to the measurement time of each run. Of course you should also not set them too low, otherwise the target will not be properly reset.

Now for the protocol. We could use the Module Wizard (like we did for the basic protocol, see Section 9.8.1, "The Application Protocol Wizard") to create a new custom raw protocol, but since our warm reset protocol is very similar to the standard XMEGA_ReadAll class we just adapt that one. In the Inspector menu bar select "File" -> "Open module source" -> "acquisition2/protocol/embedded/XMEGA_ReadAll.java". Now save this file as "XMEGA_WarmReadAll.java" (or any other name). Inspector saves it as a "user module" and automatically renames the class name and the constructor in the java file.

The changes to be made are the following:

- We have to power up only once, so we move the `powerUp()` to the `init()` method. A `powerDown()` call is not strictly necessary (since Inspector will close all connections when the module finishes), but could be placed explicitly in `close()`.

- In the **run()** method we replace the **powerUp()** call with a (warm) reset instruction.
- However, this **reset()** method should not be called in the first run, because we already had an initial power up (causing the VC Glitcher to be armed and triggered). That is why we need a **firstRun** boolean which is set to false in the first run.
- The **powerDown()** call at the end of the run has to be removed.

Optionally the **onError()** method can be implemented to log communication errors, for example. Interested users are invited to look into the source of the superclass **BasicRawProtocol** (or any other standard implementation) via the "Open system module" menu option. Anyway, the resulting code should look like this:

```
package acquisition2.protocol.embedded;

import static com.riscure.signalanalysis.data.SimpleVerdict.INCONCLUSIVE;
import java.io.IOException;
import acquisition2.protocol.BasicRawProtocol;
import com.riscure.signalanalysis.data.SimpleVerdict;
import com.riscure.util.HexUtils;

/**
 * Do embedded boot glitching with a warm reset.
 */
public class XMEGA_WarmReadAll extends BasicRawProtocol {

    private boolean firstRun = true;

    /**
     * The maximum time in milliseconds that we wait for available bytes
     */
    public static final int TIMEOUT_MS = 1000;

    /**
     * The boot sequence returned by the XMEGA device
     */
    public static final byte[] BOOT_STRING = HexUtils.hex("FF 51 41 42 43 43 6F 75 6E
74 65 72 20 3D 20 35 30 30 30 0A");

    private static final byte[] RESET_STRING = HexUtils.hex("FF 51 41 42 43 FF");

    /**
     * Creates a ReadAll protocol for the XMEGA device, with command length 0 (because
     * no data will be sent to the target) and expected response length the boot
     * string length
     */
    public XMEGA_WarmReadAll() {
        super(0, BOOT_STRING.length);
    }

    @Override
    protected void init() throws IOException {
        powerUp();
    }

    @Override
    protected void run() throws IOException {
        //only reset in second or later runs
        if (!firstRun) {
```

```
    reset();
} else {
    firstRun = false;
}

verdict(INCONCLUSIVE);

// Set the phase to the trigger phase and read data
phase(TRIGGER_PHASE.getPhaseId());
byte[] response = readBytes();
addDataOut(response);

// Determine the outcome of the glitch attempt
if (HexUtils.startsWith(response, RESET_STRING)) {
    verdict(SimpleVerdict.INCONCLUSIVE);
} else if (HexUtils.startsWith(response, BOOT_STRING)) {
    verdict(SimpleVerdict.NORMAL);
} else {
    verdict(SimpleVerdict.SUCCESSFUL);
}
}

@Override
public void close() throws IOException {
}
}
```

A Third party software licenses

This appendix contains the exact licenses under which the third party software is used:

The Bouncy Castle Cryptography is licensed under an adaptation of the MIT X11 license, and is included below.

Copyright (c) 2000 - 2009 The Legion Of The Bouncy Castle (<http://www.bouncycastle.org>)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

ASM is distributed under the following license:

Copyright (c) 2000-2005 INRIA, France Telecom All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,

BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The LGPL license version 2.1

GNU LESSER GENERAL PUBLIC LICENSE

Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc. 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the

library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) The modified work must itself be a software library.
- b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

- c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
- d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

- b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.
- c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
- d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
- e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

- a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
- b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do

not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version

or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the library's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Also add information on how to contact you by electronic and paper mail. You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library 'Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990 Ty Coon, President of Vice

That's all there is to it!

Sun TreeTable is distributed under the following license:

Copyright 1997-2000 Sun Microsystems, Inc. All Rights Reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistribution in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of Sun Microsystems, Inc. or the names of contributors may be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES OR LIABILITIES SUFFERED BY LICENSEE AS A RESULT OF OR RELATING TO USE, MODIFICATION OR DISTRIBUTION OF THIS SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING

OUT OF THE USE OF OR INABILITY TO USE THIS SOFTWARE, EVEN IF
SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You acknowledge that this software is not designed, licensed or intended
for use in the design, construction, operation or maintenance of any nuclear
facility.

B Release notes

This appendix contains the release notes of previous Inspector releases and updates.

B.1 Release notes Inspector 4.0

INSPECTOR 4.0 RELEASE NOTES

Riscure, Delft July, 2009

Changes and new features of Inspector 4.0 SCA. The most significant changes and new features of this release of Inspector are:

- Several changes to aspects of the DESAdvancedAnalysis GUI, such as intermediates, encrypt/decrypt, 3DES options.
- New DPA method DESKnownKeyAnalysis with graphical representation of leakage per round.
- New DPA method DESMaskedInputAnalysis for situations in which the 3DES input data is masked with an unknown constant.
- New DPA method DESPartlyConstantAnalysis for situation in which the input of the 3DES implementation is partly fixed.
- New DPA method ECCByteMultipleAnalysis that targets the large integer operation in an ECC implementation.
- New SPA method for ECC called ECNRPartialNonceAnalysis that targets the scalar multiplication.
- New signal processing method Spectrogram to analyse progression over time of the spectrum.
- New alignment method called Elastic Alignment with several options for determining the best parameters for the alignment. Further, the GUI of the existing alignment functions have been changed to accomodate the elastic alignment.
- 21 tutorials available in the manual to practice alignment and DPA functions on trace sets supplied on the Trace Sets DVD.
- 3 new training cards are supplied with this release. Two cards include configurable countermeasures for its software AES and DES implementation. One card includes a hardware DES and ECC implementation.
- New computational core using the data abstraction that was introduced in Inspector 3.0 and adding operation abstraction to this. Backwards compatible with 3.0. Guidance on the new API is found in the manual.

- New method called SegmentedChain adds the following to Chain: it segments the original trace set in multiple trace sets which are processed separately by the filter chain.
- New communication interfaces to control embedded devices. Interfaces for serial port, tcp/ip, and for using the external serial interface of the Power Tracer. A reference C implementation is provided for the embedded device under evaluation.
- Separation of user and system space to ease organisation of trace sets and custom written modules. User space will not be touched by future software updates. User written modules are also visible in the GUI separate from system provided modules. Note that your own existing modules should now write data to the user space.
- Bug fixes.

B.2 Release notes Inspector 4.1

INSPECTOR 4.1 RELEASE NOTES

Riscure, Delft January 25th, 2010

These are the release notes of Inspector 4.1. The general focus for this release is stability and usability, and in select areas speed improvements. Significant specific changes are listed below.

New Inspector version

A major change introduced in Inspector 4.0 release is the addition of a new Inspector version: Inspector FI. Inspector FI is designed for users interested in performing Fault Injection or Perturbation. The addition of this new flavour logically changes the name of the previous released Inspector to: Inspector SCA (Side Channel Analysis). These Inspector versions consist of different modules, and can be purchased and used separately, but also combined in one Inspector (SCA+FI). The type of update you receive with this release depends on the Inspector version you have purchased.

New Inspector Hardware support

Inspector FI supports Riscure's VC Glitcher and Riscure's Diode Laser Station (DSL) for physical fault injection in target objects. Both Inspector SCA and FI support Riscure's icWaves device for real-time pattern matching.

New features

The following features have been added to the Inspector core to improve handling and performance.

- GPU (via CUDA) support is added to allow calculations to be done on the GPU. Inspector automatically determines if it is faster to run calculation on the GPU.
- A text file log is created when a module is run to save the settings that are used. This log is viewable from the GUI.
- A reset button is added for all modules to enable returning to the default settings of the module.
- Module files are always saved as UTF-8 regardless of system settings, to allow compiling of modules on systems with non UTF-8 settings.
- The correlation module can handle more than 64bits simultaneously.
- Lower sample rates can be selected on the Picoscope 5000.
- Support is added for a negative delay using the Picoscope 5000.
- icWaves can be configured from a menu item in the tools menu.
- For Inspector FI, several menu items have been added in the tools menu for handling and configuring the camera and also to view the perturbation log.
- Many new modules are added like, DES template analysis, SEED analysis, AES advanced analysis, AES known key analysis, AES DFA analysis, Acquisition support for Sasebo embedded board. For the full list see "New Modules" below.

Bugfixes

- Fixed crashing of Inspector during performing of FFT on large sample traces
- Picoscope stability improvements
- For the Picoscopes the correct impedance(1MOhm) is shown
- Statistics modules can be now be loaded into the (segmented)chain
- The "Open" icon remembers the location where the last trace was opened
- Fixed occurrence of random changes in trace-set during trace-set processing

New Documentation

The manual has been revised and extended. It now presents more information about Inspector and its components in a more structured manner. Due to the large growth of tutorials to practise Side Channel Analysis and Fault Injection, a separate document has been made that contains all the tutorials.

New Modules

For both Inspector versions the following module had been added to support the icWaves hardware.

- icwaves/

- icWavesConfiguration - module to configure the icWaves

For Inspector SCA the following modules are added to the new release:

- aqcuisition/
 - SASEBO aqcuisition - acquisition module for the embedded Sasebo demo board
- crypto/
 - AesAdvancedAnalysis - retreive an AES key using sophisticated statistical analysis methods
 - AesKnownKeyAnalysis - determine data leakage of intermediate AES results using a known key
- DesTemplateAnalysis - retrieve a DES key using template analysis methods
- SeedAnalysis - retrieve a SEED key using sophisticated statistical analysis methods

For Inspector FI the following modules are added to the new release:

- crypto/
 - AesDFA - retreive an AES key by analyzing fault-injected outputs
 - DesDFA - retreive a DES key by analyzing fault-injected outputs
 - RsaCrtDFA - retreive a RSA private key by analyzing fault-injected outputs
- perturbation/
 - AESOpticalPerturbation
 - AESPerturbation - perform a perturbation attack on a AES implementation
 - ATROpticalPerturbation - perform an optical perturbation attack on a AES implementation
 - ATRPerturbation - perform a perturbation attack on a ATR implementation
 - DESOpticalPerturbation - perform an optical perturbation attack on a ATR implementation
 - DESPerturbation - perform a perturbation attack on a DES implementation
 - PIN2OpticalPerturbation - perform an optical perturbation attack on a DES implementation
 - PIN2Perturbation - perform a perturbation attack on a double checked PIN implementation
 - PINOpticalPerturbation - perform an optical perturbation attack on a double checked PIN implementation

- PINPerturbation - perform a perturbation attack on a single check PIN implementation
- RSACRTOpticalPerturbation - perform an optical perturbation attack on a single check PIN implementation
- RSACRTPerturbation - perform a perturbation attack on a RSA-CRT implementation
- SmartCardPerturbation - the application independent module for performing a perturbation attack
- SmartCardOpticalPerturbation - the application independent module for performing a optical perturbation attack
- xyz/
 - AveragePlot

To see if any of the other modules is included with the SCA or FI version, please refer to the Inspector manual.

Known issues

- The LeCroy oscilloscope sometimes performs an internal calibration, that takes about a second or two. Inspector can time out during this calibration. It happens mostly with the first trace, but can also happen during an acquisition. The result is an incorrect trace present in the trace set. When doing signal processing like alignment this trace will be removed from the set and will not have any impact. However when running spectralintensity it will result in a single spot with high intensity and the rest of the plot will be blue. After stopping the acquisition and re-starting it, the acquisition will run fine. Also the 'auto calibration based on temperature' can be manually set the to off on the LeCroy.
- When selecting 'Com1' as interface for XYZ-table, the first time the pulldown might drop back to 'DummyXYZTable'. Selecting 'com1' a second time will make it stick and enable use of the xyz-table.
- When the VC++ runtime is not installed on the PC, trying to open the camera results in the following error:

"java.lang.UnsatisfiedLinkError: C:\Program Files\Inspector-4.1\lib\Win32\riscurve-camera.dll: This application has failed to start because the application configuration is incorrect. Reinstalling the application may fix this problem."

This can be solved by installing the MS VC++ redistributable package from here [<http://www.microsoft.com/downloads/details.aspx?FamilyID=9b2da534-3e03-4391-8a4d-074b9f2bc1bf>], which is also included on the Inspector 4.1 installation CD in the "MS VC++ redistributable package" folder

- Running the align module might result in the following messages in the message pane:

Failed operation r2hc on
com.riscure.signalanalysis.inspector.R2HCCuda@88e83d, retrying other provider...

Failed operation normalize on
com.riscure.signalanalysis.inspector.NormalizeCuda@a40f0e, retrying other provider...

Failed operation hc2r on
com.riscure.signalanalysis.inspector.HC2RCuda@1bc081f, retrying other provider...

...

These messages do not have any effect on the alignment function, but merely means that the GPU is not used. The messages can be avoided by switching off the GPU in File->Settings->Other and unchecking the GPU tickbox. However, this switches off the use of the GPU with other processing as well.

B.3 Release notes Inspector 4.1.1

INSPECTOR 4.1.1 RELEASE NOTES

Riscure, Delft July 12th, 2010

These are the release notes of Inspector 4.1.1. The focus in this release is increased stability by having tackled issues present in the Inspector 4.1 release. No major new features have been added. The most significant user related changes are listed below.

New Features

- training cards can now also be configured with the VC Glitcher in the TrainingCardConfig module
- training card 6 can be configured in the TrainingCardConfig module
- documentation is added to the manual explaining how to use Eclipse for module development

Resolved Bugs in Inspector Core

- fixed OutOfMemoryError during some memory demanding operations
- fixed an OutOfMemory exception in the Picoscope driver
- use of Java Generics in the module editor is handled correctly
- saving is disabled during acquisition to prevent data loss

- fixed Inspector window focus, to prevent dialog windows moving behind inspector

Resolved Bugs in Inspector SCA

- fixed incorrect first trace with the LeCroy oscilloscope
- fixed hanging of Inspector during abort of acquisition with Power Tracer
- correctly parsing the R-block block number in the PowerTracer driver, fixing communication halt when switching between T=1 and raw modes
- calculation of AESAdvancedAnalysis uses the shiftrows operation for all rounds
- fixed static alignment reporting correlation to be 0
- changed AesKnownKeyAnalysis fixing an exception when a single bit and a single key byte are attacked
- AveragePlot uses firstTraceIndex attribute fixing a fail if the first trace is not trace 0
- selecting a location in SpectralIntensity correctly displays the corresponding trace
- selecting of the Embedded Tracer does not result in "requires firmware version 2.3" error
- fixed functionality of the frequency sliders in SpectralIntensity module

Resolved Bugs in Inspector FI

- PostgreSQL account/password does not expire anymore
- fixed crash when compiling the perturbation module during the initialisation of the VC Glitcher

Resolved Bugs in icWaves

- fixed occasionally occurring "ERROR: Could not set the filter frequency."
- filter frequency is correctly loaded in the icWaves configuration menu item
- fixed crash when selecting icWaves configuration, caused by corruption in the registry configuration
- stored trace pattern 1 is reset when pattern 2 is enabled/disabled

Known issues

- Selecting a large plot with the spectral intensity module, results in a OutOfMemoryError.

Workaround: pressing the free memory(CG) button once in a while during the plot, frees memory

- When a card is present on the RF Tracer and windows is shutdown, windows ends in a crash.

Workaround: remove card before shutting down

- Running AesSecondOrderAnalysis, progress value stays at 0.0%
- When the 3D view is open in modules such as AveragePlot or SpectralIntensity, and the log scale is toggled, inspector hangs and needs to be terminated [1024]
- Trying to open the DLS camera results in the following error:

"java.lang.UnsatisfiedLinkError: C:\Program Files\Inspector-4.1\lib\Win32\riscure-camera.dll:

This application has failed to start because the application configuration is incorrect.

Reinstalling the application may fix this problem."

Workaround: This occurs when the VC++ runtime is not installed on the PC and can be solved by installing the MS VC++ redistributable package from here [<http://www.microsoft.com/downloads/details.aspx?FamilyID=9b2da534-3e03-4391-8a4d-074b9f2bc1bf>].

B.4 Release notes Inspector 4.2

=====

INSPECTOR 4.2 RELEASE NOTES

Riscure, Delft October, 2010

=====

These are the release notes of Inspector 4.2. The focus for this release has been some major new features. The most significant user related changes are listed below, ordered in "New Features", "Bug fixes", "Known issues" and "Other changes".

New Features in Inspector Core

- Inspector supports 64 bit Windows 7. The installer provides an option to install 64 bit support (disabled by default). When checked, an additional start menu item is created to run Inspector in 64-bit mode. Note that 64-bit mode will only work on a 64 bit OS, is currently only supported for Windows 7 and can only use drivers which provide a 64-bit API. Currently this excludes the PicoScope and the RFTracer, see known issues below.
- A new contactless reader, the MicroPross MP300 TCL1/TCL2 is supported.

- Experimental support for IVI-compliant oscilloscopes has been added to enable the use of more oscilloscope types via this driver. The wide range of oscilloscopes is shown in the driver registry from the IVI foundation. See here [http://www.ivifoundation.org/registered_drivers/driver_registry.aspx].
- A hardware manager was introduced in order to centrally manage initialization and allocation of hardware resources. This results in a better handling of hardware in future releases. It currently has no user visible effects. However, modules accessing hardware which do _not_ extend Oscilloscope, SideChannelAcquisition, or SmartCardPerturbation will have to change the way devices are queried.
- The resample granularity of SyncResample is improved.
- Exception logging has been added. This log can be used to provide Inspector support with additional information to diagnose problems. Exception logging can be enabled in the settings dialog.
- Reloading of default settings have been added to all modules.
- When opening traces corrupted during acquisition, Inspector will now attempt to recover them automatically(after asking the user for permission). For heavily corrupted traces the Recover module can still be used.
- A pause button has been added to the user interface. This can be used to temporarily suspend the running of the module, for example to free up some processor time.

New Features in Inspector SCA

- Many new modules have been added to Inspector. This has resulted in a new order of arrangement of the modules, which is very noticeable. A new package called Compress has been created. This package contains the Resample and SyncResample modules, previously located in the Filter package. In addition, two new modules (RFResample and WindowedResample) were added to this package. Also the InvNotchFilter module has been added to the Filter package.
- A crypto2 package is created with the following new modules; Calculator, First Order Analysis, KnownKeyCorrelation, Simulator, TemplateAnalysis and Verify. This package is built as a DPA framework with a dynamically generated user interface. The cipher/attack selection dialog is generated based on (annotated) constructor arguments. For more information on the modules refer to the Inspector manual, modules section.
- New crypto algorithms available in the Crypto2 package are: Misty, Camellia, ECDSA, DSA, AES, (3)DES. For more information on the modules refer to the Inspector manual, modules section.
- The Stretch module is added to the align package which can be used to scale traces that were previously statically aligned. It enables unstable clocks or random process interrupts to be corrected.
- Euclidean DPA option has been added into AdvancedDifferentialAnalysis, for more information turn to the description of DesAdvancedAnalysis in the modules section.

New Features in Inspector FI

- TriggeredPerturbation modules have been added for DES, DES-optical and RSA-optical. This allows the VC Glitcher to be used with other IO devices (e.g. a contactless card reader or embedded target). For more information on the modules refer to the Inspector manual, modules section.
- The FI database connection now uses a pure Java driver instead of connecting through ODBC.

Bug Fixes

- Fixed average ranking failure with intermediate function, in modules DesKnownKeyAnalysis and AESKnownKeyAnalysis.
- The "Append" function is disabled during module execution to prevent unwanted behaviour.
- The internal float representation of certain "int" values in the VC Glitcher is fixed.
- The 3 align options have been separated into 3 modules to resolve some incompatibilities.
- The "detect round" combo-box value is correctly saved in the parameters file
- New acquisition parameters have been added in tutorials document to more precisely describe acquisitions on the training cards for DES and AES. Also training trace sets on the DVD have been changed.- when communication with dongle is lost a grace period allows for reestablishing without going into standby
- running multiple Inspector instances sometimes crashes on java3d, this is fixed with latest Java 3D version
- Error message on PCs that do not support CUDA are removed.
- Recover function can recover files larger than 2 GByte.
- Changing the "Use GPU" option no longer requires restarting Inspector before it is applied.

Known Issues

- ISSUE: Sometimes a connection with the Power Tracer cannot be established under Windows 7 when using a USB hub.
WORKAROUND: Unplug the Power Tracer from the USB hub and then plug the Power Tracer directly into the computer.
- ISSUE: As mentioned above, 64 bit drivers are currently not available for the PicoScope and the RF Tracer.
WORKAROUND: The PicoScope can be used in Windows 7 32 bit compatibility mode and Windows XP 32 bit. The RFTracer should only be used in Windows XP 32 bit.

- ISSUE: At startup the following exception might occasionally occur.

```
Exception in thread "Basic L&F File Loading Thread"
java.lang.NullPointerException

at sun.awt.shell.Win32ShellFolder2.equals(Win32ShellFolder2.java:491)
at sun.awt.shell.Win32ShellFolderManager2.isFileSystemRoot(Win32ShellFolderManager2.java:350)
at sun.awt.shell.ShellFolder.isFileSystemRoot(ShellFolder.java:242)
at javax.swing.filechooser.FileSystemView.isFileSystemRoot(FileSystemView.java:323)
at javax.swing.filechooser.WindowsFileSystemView.isTraversable(FileSystemView.java:657)
at javax.swing.JFileChooser.isTraversable(JFileChooser.java:1561)
at javax.swing.plaf.basic.BasicDirectoryModel
$LoadFilesThread.run0(BasicDirectoryModel.java:232)
at javax.swing.plaf.basic.BasicDirectoryModel
$LoadFilesThread.run(BasicDirectoryModel.java:211)
```

WORKAROUND: This exception does not seem to block running Inspector normally.

- ISSUE: The MP300 does not powerDown() first if powerUp() is called. Apparently some tools *only* call powerUp() to perform a cold reset on a card. Running powerUp() a second time on an already selected card causes the MP300 driver to rerun the select/wakeup procedure, which will fail and the driver responds that no card is present.
WORKAROUND: call powerDown() before calling powerUp()
- ISSUE: The DesKnownKeyAnalysis sometimes shows vertical blue lines in the plot window. This only occurs when a word size of 32 bits is used.

Other Changes

- The CUDA toolkit is now included in the drivers folder of the Inspector installation. The CUDA driver is no longer included because the stock NVIDIA driver now includes all CUDA functionality.
- FFTW was upgraded to version 3.2.1 to increase speed

B.5 Release notes Inspector 4.2.1

=====

INSPECTOR 4.2.1 RELEASE NOTES

Riscure, Delft November, 2010

=====

These are the release notes of Inspector 4.2.1, which is an appendix to the release notes of version 4.2. This version is mainly a patch for certain bugs found in the 4.2 release.

It mainly concerns patches for two items:

- the icWaves driver
- and the Spectrum module

icWaves driver

Due to a change in the hardware manager, the icWaves driver is not behaving as expected. As a result, icWaves users will not be able to start an acquisition with the ICWavesScope in Inspector 4.2. This bug does not affect the quality of your measurements. Previous releases of Inspector are not affected by this bug.

Spectrum module

Due to a rounding bug in the module, the calculated spectrum was not accurate enough. This could result in visible computational errors when for example running the Xtalclear filter.

B.6 Release notes Inspector 4.3

=====

INSPECTOR 4.3 RELEASE NOTES

Riscure, Delft March, 2011

=====

These are the release notes of Inspector 4.3. The focus for this release has been a new attack method cross-correlation which is currently implemented on RSA. The attack is based on the paper of Wittman et al. (Defeating RSA multiply-always and message blinding countermeasures [http://www.riscure.com/fileadmin/images/Docs/rsacc_ctrsa_final.pdf]) which was presented at The RSA Conference 2011. Additionally, we increased support for controlling target devices.

The most significant user related changes are listed below, ordered in "New Features" and "Resolved Bugs".

New Features

- INS-1439: Added modules to support neighbor-/cross-correlation on RSA
- INS-1354: Smart cards can now be managed through the Target Manager. This allows for loading and unloading of code on the target.

- INS-1353: In order to facilitate developing plug-ins an automatic user interface generation framework was added.
- INS-1356: Acquisition 2 framework for implementing transport protocol was added. Protocols are no longer embedded with a device, but an entity in their own right.
- INS-1322: HMAC protocol has been added to crypto2.
- INS-1185: A hardware manager was added to the Tool menu to allow managing devices such as oscilloscopes and card readers.
- INS-1113: The module TraceSort that sorts Traces has been implemented
- INS-1383: Added new TLV based application protocol for embedded devices
- INS-923: Filter by color in the VC Glitcher report added.
- INS-1407: Added 03.48 OTA/SCP80 protocol support
- INS-1529: Inspector now can detect Plug and Play events for certain devices.
- INS-1112: Input fields can be entered using SI prefixes (k, M, G, m, u, p, etc.) in new dynamic user interface

Resolved Bugs

- INS-1433: Embedded Java upgraded to version 1.6.0_22
- INS-1245: Updated to JNA 3.2.7
- INS-1473: Fixed crash in dynamic alignment
- INS-1501: Fixed crash during FFT operations
- INS-1526: EMPS moving in wrong direction in rare cases has been fixed
- INS-1465: Fixed addition of new parameters to existing perturbation modules
- INS-1589: Sample DES module expected wrong data, this has been fixed
- INS-1605: Settings changed in the hardware manager are persisted when Inspector is closed and restored when started again
- INS-1394: Perturbation modules will now default to dummy VCGlitcher if no actual VCGlitcher is present
- INS-1494: Added description of training card 7
- INS-1502: Fixed FirstOrderAnalysis options 'Diff' and 'Amplified correlation'
- INS-1486: API documentation extended with interfaces used by hardware and target manager

- INS-1518: Regression regarding loading of saved modules parameters resolved
- INS-1669: For testing purposes an APDUEcho device was introduced in acquisition2. This device will echo the APDU sent adding 90 00
- INS-1428: Module exception handling has been improved
- INS-1349: Added receiveAnswer(Command command, int getResponse, int getResponseClass) method for choosing the class of the GET_RESPONSE APDU in perturbation modules
- INS-955: Bit number in AES modules is now consistent from 1 to 8
- INS-997: Documentation has been updated to clarify VCGlitcher trigger pulse behaviour
- INS-1624: Resolved issue in which the module wizard generated invalid code for specific analyzer modules
- INS-1539: The DPA 'diff' option in crypto and crypto2 has received a bugfix that affected the peak calculation in case of multibit-DPA
- INS-1459: NPE when acquiring with icWaves scope was fixed
- INS-1471: Fixed rounding bug in Spectrum

B.7 Release notes Inspector 4.4

INSPECTOR 4.4 RELEASE NOTES

Riscure, Delft November, 2011

These are the release notes of Inspector 4.4. The focus for this release has been the new Acquisition framework and hardware manager to improve the overall stability of the drivers.

The most significant user related changes are listed below, ordered in "New Features" and "Resolved Bugs". There are some known issues with this release listed under "Known Issues"

New Features

- INS-1408 Added ePassport acquisition protocol
- INS-1524 EM Probe Station can now be configured in hardware manager
- INS-1564 Acquisition 2 modules can now be added as the first module in a chain
- INS-1409 Added USIM acquisition protocol
- INS-1406 Added EMV acquisition protocol

- INS-1110 Function previously only available in the trace context menu can now be accessed from the Edit menu
- INS-1509 Serial port devices are now automatically detected and added to the hardware manager. In order to obtain serial behaviour of previous version create a protocol device using the serial device and the "Old embedded" protocol
- INS-2053 Support for ARIA cipher in crypto2
- INS-2122 64 bit version of Inspector is now installed by default
- INS-1403 XY scanning implemented in new acquisition framework
- INS-1510 TCP/IP devices for a specific hostname/port can be added through the hardware manager
- INS-1838 Support for importing and exporting traces to MatLab for use in e.g. OpenSCA has been added
- INS-2008 SASEBO-W is supported through a generic raw serial card reader interface
- INS-1485 Creating new ciphers, application protocols and trace sorters can be creating using the Inspector module development wizard
- INS-1879 Elastic alignment FastDTW code rewritten for a 2-3x speedup
- INS-1960 Closing Inspector shows confirmation dialog to store unsaved traces

Resolved Bugs

- INS-2059 WindowedResample now produces correct result when used with Windows size 2 and overlap ratio 0.0
- INS-1949 Small scaling bug with trace windows was resolved
- INS-1926 User acquisition module doesn't show up in Run -> User -> Acquisition menu (Inspector CORE)
- INS-2022 Under certain conditions when performing an acquisition with both Power Tracer and EMPS attached Inspector would crash, this has been resolved
- INS-1980 Corrections for PINPerturbation tutorial 5.2.1
- INS-1957 Fixed rounding error in perturbation modules
- INS-1743 Documented icWaves armed by trigger feature

Known Issues

- INS-1838 (MATLAB) Files over several hundred megabytes can not be imported in the 32 bit version of Inspector. Solution: Use the 64 bit version of Inspector

- INS-2008 (SASEBO-W) Certain cards do not work correctly with SASEBO-W, this includes training card 5 and 8. Workaround: Use Power Tracer
- INS-2340 (Acquisition2) Viewing traces in Trace Window at 10 kS 1Gss/s occasionally causes missing the trigger pulse with the following settings ScopeAcquisition > Power Tracer, trigger delay 0 us > TC8 > PicoScope 1 Gss/s 10 kSamples, External trigger, Channel A. Workaround: align traces so erroneous traces are rejected from the traceset

B.8 Release notes Inspector 4.5

===== INSPECTOR 4.5 RELEASE NOTES

Riscure, Delft July, 2012
=====

These are the release notes of Inspector 4.5. The focus for this release has been the new Perturbation framework "Perturbation2" and user guidance / user support.

The most significant user related changes are listed below, ordered in "New Features" and "Resolved Bugs". There are some known issues with this release listed under "Known Issues"

New Features

- INS-2396 Perturbation 2 framework based on Acquisition 2: This part will give the user the freedom to use system or user defined application protocols from Acquisition2 Framework and to glitch within a protocol. No backwards compatibility for Perturbation1 modules. Perturbation 1 remains available for user
- INS-1889 PatternPad: preprocessing step to improve the alignment output of Elastic Alignment
- INS-1893 PatternExtract: Improved version of PatternMatch, allowing non-overlapping pattern extraction
- INS-1844 FilterGuidance: gives suggestions on the choice of appropriate frequency filters for sidechannel signals. It automatically constructs multi-bandpass filters based on cross-correlation between narrow-band side-channel traces
- INS-2463 DLS navigation: coordinate system normalization in XY such that moving targets or resetting the stage can be rectified for. Normalization would overcome the limitation of the absolute stage positions
- INS-2399 Dual location DLS UI features. This allows the user to manoeuvre the XYZ manipulator, calibrate the XY corner coordinate of the scan area and add the XY glass fiber position to the perturbation parameters
- INS-2787 User can change default folder for saving tracesets

- INS-1512 MP300 driver support by Hardware Manager and integrated in Acq2, ISO 14443 refactored and isolated from MP300 driver
- INS-1476 PicoScope driver support by Hardware Manager
- INS-2346 VCGlitcher driver support by Hardware Manager
- INS-1525 Tango driver support by Hardware manager
- INS-2883 RiscureLV transport protocol for embedded devices

Resolved Bugs

- INS-3050 prevent potential Null Pointer Exception in APDUEcho
- INS-3049 Added check to avoid Null Pointer Exception error in SerialDevice
- INS-2923 Helpfile for AES Advanced Analysis corrected
- INS-2921 In AESAdvancedAnalysis, Target = MixColumns" + "Model = S-box" makes only sense when attacking encryption with output or attacking decryption with input. Invalid options block in UI
- INS-2897 Fixed a memory leak in the Hardware Manager
- INS-2879 RsaNeighborCorrelation only computes first half of correlation values when step 2 is chosen (which is one of two normal values) Due to this bug the correct key is not found, even when correlation is good
- INS-2876 Selecting a point in the cross correlation window with a mouse click did not select the correct point if window was rescaled. Now fixed
- INS-2831 Acquisition 2 now will log the initial part of protocol
- INS-2806 Jama matrix library is now included in the installer which fixes TemplateAnalysis
- INS-2776 Manual entry for Elastic Radius corrected
- INS-2751 Bug fix for DESKnownKeyAnalysis and DesAdvancedAnalysis regarding HD of Sbox
- INS-2743 When set resample frequency as "1.0" and "Margin"!=0, Inspector uses resample frequency 5e8 instead of 1.0. This is now fixed
- INS-2690 ISO stack does not read PPS checksum. This is now fixed
- INS-2663 Round key in DesKnownKeyAnalysis decrypt is wrong. Now fixed
- INS-2598 Added documentation for recently added frameworks Acquisition 2, Perturbation 2 and Crypto 2
- INS-2571 Some language improvements in the Manual and Tutorial
- INS-2543 Inverse Key Schedule in AesAdvancedAnalysis does not work correctly for 192 and 256 bit. This is now fixed

- INS-2523 Opening a file in the file browser dialog window with detailed view caused an error when opening a second time. Now solved
- INS-2517 Fixed bug in AesKnownKeyAnalysis in decrypt mode
- INS-2497 AGG generated file picker doesn't give user feedback on currently selected item. This is now fixed
- INS-2375 New drivers and firmware upgrades on Splitter, icWaves and VC Glitcher
- INS-2372 Acquisition 2 allows loading/saving of module parameters
- INS-2350 Adjusted PatternPad calculation of the right hand side of the search window
- INS-2336 Added figures to the Hardware Manager entry in the manual
- INS-2326 Corrected Y scaling in ScopeAcquisition
- INS-2305 Added documentation on XYAcquisition
- INS-2279 Fixed missing JavaDoc for Trace, TraceSet and ProgressInterface
- INS-2274 Help file of DynamicAlign now shows information as expected
- INS-2268 Now properly inform the user when his/her system is not suited for 64-bit Inspector
- INS-2252 Added note to documentation that correlation may exceed 1 when 'amp' is used as option
- INS-2211 Now also apply the filter for XtalClear when there is a 0Hz cutoff frequency
- INS-2180 Cipher templates in New Module Wizard now does compile
- INS-2140 Fixed bug in TriggeredPerturbation in case of communication errors. If Inspector detects a problem (normally low level communications or deselect in MP300 for one transaction), the "logdata" window is initialized in another window and the attack starts again
- INS-2139 Fixed several bugs in DesKnownKeyAnalysis
- INS-1738 Link to help file of New Module wizard was broken
- INS-1282 The DesKnownKeyAnalysis sometimes shows vertical blue lines in the plot window. An explanation for this is added to the manual

Known issues

- INS-2882 Using a USB-to-Serial convertor is known to give issues when connecting it to the EM Probe Station. The device may stop scanning at a random interval and loose connectivity to Inspector. Users are advised to connect the EM Probe Station directly to the serial port.
- INS-2793 The state 'Show advanced settings'-checkbox selection is not stored in acquisition2 and perturbation2 modules. This can be observed

when re-opening one of these modules. Advanced settings themselves however, are stored. To see these settings, users will have to manually check the 'Show advanced settings'-checkbox.

B.9 Release notes Inspector 4.5.1

INSPECTOR 4.5.1 RELEASE NOTES

Riscure, Delft November, 2012

These are the release notes of Inspector 4.5.1. The focus for this release is Power Tracer 4.

The most significant user related changes are listed below, ordered in "New Features" and "Resolved Bugs".

New Features

- Power Tracer 4 works with laptop
- Power Tracer 4 features a configurable smart card clock, ranging from 1MHz to 10MHz
- Power Tracer 4 has faster acquisition speed
- Power Tracer 4 can receive firmware upgrade
- Power Tracer 4 has better signal quality
- Power Tracer 4 features a front-mounted LCD panel displaying common parameters
- Power Tracer 4 improves the performance of internal amplifier
- Power Tracer 4 clean DC power supply has a larger capacity
- Power Tracer 4 is powered by a stand-alone 15v PSU
- Power Tracer 4 can detect clean DC power shortage and charge it automatically
- Power Tracer 4 features a set of ports that duplicate common signals of smartcard interface, which includes RST, CLK and IO
- Power Tracer 4 combines the original RS232 port and +12 V port into a single P/S2 socket
- Power Tracer 4 features a new housing style
- Inspector Installers will create new desktop shortcuts by default
- Manual trigger setting on LeCroy

- Embedded Protocols are now available in FI only installer also

Resolved Bugs

- Embedded Protocols will return correct data length on error
- Added all 3rd party drivers to \hardware folder
- Fixed handling of xor mask in AesAdvancedAnalysis and AesKnownKeyAnalysis
- Fixed issue in AesKnownKeyAnalysis when using Decrypt, Intermediate > 0, and round = all
- Fixed calculation of "average ranking" in AesKnownKeyAnalysis when only one key byte is attacked and multiple intermediate results are used

B.10 Release notes Inspector 4.6

INSPECTOR 4.6 RELEASE NOTES

Riscure, Delft March, 2012

These are the release notes of Inspector 4.6. The most significant user related changes are listed below, ordered in "New Features", "Experimental Features", "Documentation", "Known Issues" and "Resolved Bugs".

New Features

- INS-3188 User is now able to specify a custom, configurable, perturbation snippet for each phase of the protocol.
- INS-3153 The menu structure has been totally redesigned to put modules in a logical place and reduce the number of clicks. In addition to the default structure 4 additional profiles are included which provide a menu structure for both basic and advanced workflows. If you prefer the old structure a Classic profile is also included. User can create own profile also.
- INS-3191 Similar to the perturbation fragment, the user is able to edit a complete perturbation program. The GlitchProgram editing is similar to that of the fragment except that some additional instructions are available and the user has full control over label names and memory locations.
- INS-3119 Serial port timeout can now be defined in the hardware manager
- INS-3453 Round numbers and reverse round numbers have been unified for all ciphers
- INS-3061 New @Tunable annotation introduced, which allows properties to be hidden or shown from device live tuning view

- INS-3161 Known key analysis is now fully supported in crypto2. In addition all attacks available in crypto1 can now be selected in the similar user-friendly way, this includes the 'all' option. Both round and sub keys can be set. If the default attacks provided are not sufficient, user can switch to expert mode and define their own leakages without writing a single line of code. If the leakage models provided do not work for the target, user can create their own by writing a couple of lines of code. Switching Distance have been included to demonstrate this
- INS-3244 Inspector now supports the Embedded Glitcher which is part of the VC Glitcher 2. This mode allows for glitching an embedded target without being limited by a smart card clock cycle. The glitches can be started either on a trigger provided by the target or the icWaves (command glitching) or synchronized to the reset of the target (boot glitching). Highly complex glitches with a resolution of 2 nanoseconds and lasting up to several minutes are supported. Example code for the target has been included as well as a tutorial on how to set everything up
- INS-2838 Inspector keyboard shortcuts have been enriched
- INS-1567 Extended UI validation to include invalid spinner values
- INS-2956 Enable Java 7 features in Inspector
- INS-3063 Creating an application protocol has been greatly simplified with the introduction of a number of utility methods and an easy to use wizard. Addition of the application protocol has been given full control of configuring the target and determining the outcome of the perturbation attempt (verdict). The training target protocols have been extended to allow configuring the target (replacing TrainingCardConfiguration) and now will automatically determine whether a fault was injected based on the cipher key provided. In case fixed or random input is not sufficient 3 additional data generators have been added. A custom generator can be easily written.
- INS-3403 Because of the implementation of parameter files, the auto save beans in every 5 seconds feature has been removed
- INS-3364 Added source code for XMEGA training target
- INS-3159 Added a general setting that will speed up reference correlation, but will possibly invalidate the results
- INS-3266 Added a checkbox to ignore Inspector side trigger settings for the LeCroy allowing more advanced trigger settings to be applied on the scope side
- INS-3176 Added 2 new data generators:
 - Counter: return data starting from user supplied value and increase by a user supplied stepsize
 - Fixed Input List: return data from a fixed list of values in round robin fashion Added a feature that allows you to repeat a specific data generator after n traces
- INS-1183 Add icWaves to hardware manager

- Improved UI for configuring icWaves.
- Capable of working with multiple icWaves simultaneously.
- Make icWaves available from Acquisition2 and Perturbation2.
- Enabled run-time tuning filter value during acquisition2.
- Enabled run-time tuning diverse parameters during acquisition2 and perturbation2.
- Added icWaves trigger info into Perturbation2 Log.
- The icWaves 2 support for switching coupling between AC and DC

Experimental Features

- INS-3531 Basic POI selection.
- INS-3240 Update TemplateAnalysis to work with new leakages.
- INS-3194 Template analysis for Crypto2.

Documentation

- INS-3339 Updated section on Amp method in the DesAdvancedAnalysis module description.
- INS-3354 Fixed CSS for html documentation.
- INS-2402 Updated links to Riscure website.
- INS-3356 Updated firmware upgrade process description for VCGlitcher.
- INS-2412 Documentation improvements in section "Writing perturbation modules" under subsection "void runPerturbation".
- INS-3904 Documentation about writing custom Raw Protocols.
- INS-3091 Corrected CleanWave specification in the manual.
- INS-3796 A conversion guide is added to the manual for upgrading user's Inspector 4.5 Cipher implementations to the Inspector 4.6 standard.

Known Issues

- INS-3933 Application protocol wizard tutorial is outdated.
- INS-3772 Inspector fails to start in 32 bit mode when more than 4 GB of memory is in use.

Resolved Bugs

- INS-2763 Fixed a bug of auto-renaming issue when there is a '.' in traceset name.

- INS-2983 Change inspector icon to high resolution one.
- INS-1948 Zooming on traces will now exactly match the selected zoom area.
- INS-3884 Using the MP300 in perturbation2 now correctly logs communication data
- INS-3494 The x axis scaling has been improved and will no longer just print "..." before large truncated numbers without any reference.
- INS-3185 The data provider is now initialized once per acquisition rather than once per attempt
- INS-2777 SyncResample module warns user when sample frequency is too high
- INS-3776 Sum criterion is now outputs the same results when used in crypto2 compared to crypto1
- INS-3732 Solved bootstrap class path warning when compiling using the Inspector IDE
- INS-3025 Several bugs in the Chain module were fixed, such as persistence, and the user interface has been made more robust and friendly
- INS-3107 Selecting a PicoScope when the driver is not installed (properly) will no longer cause the selected module to crash.
- INS-3907 Retrieving the same data from the PicoScope twice now works without error
- INS-3759 Protocol device no longer appears multiple times after Inspector restart
- INS-3860 Perturbation modules now indicate a trace is unusable for DFA when the verdict is inconclusive rather than only when a timeout occurs
- INS-3628 Changing glitch voltage affects VCC output (VC Glitcher issue)
- INS-3141 Lower the highest supported RS232 baudrate to 115200
- INS-3777 KnownKeyCorrelation now remembers the index of the input trace in the output traces
- INS-3679 KnownKeyCorrelation now outputs correct y-axis title and scale
- INS-3605 Inspector no longer crashes if a significant number of user modules is compiled
- INS-3627 Increased Java working memory for Inspector to resolve Out Of Memory errors.
- INS-3363 In acquisition2, Powertracer4 will not expose the smart card frequency and channel (smart card/serial) info during the acquisition.
- INS-3663 Improved performance in "Perturbation" tab of perturbation2 modules.

- INS-2764 Improved how modules reaction to "input TraceSet missing"
- INS-3101 HMAC-SHA1 now correctly stores keys in intermediates object
- INS-3527 Glitch tab will never show empty clock speed list.
- INS-3831 Glitch offset is now used everywhere consistently including perturbation log
- INS-3718 Fixed simulation for 8 bits HMAC-SHA1
- INS-2411 Fixed link to Inspector API in Help menu
- INS-1892 Fixed help to RFResample module
- INS-3262 Fixed calculation of "average ranking" in AesKnownKeyAnalysis when only one key byte is attacked and multiple intermediate results are used
- INS-1885 Fixed bug in Resample module where one sample too many was generated
- INS-1903 Fixed bug in ABS module w.r.t. faulty handling of -128 in byte encoded tracesets
- INS-3700 Fixed a bug where changes made to a Protocol Device would not be stored and restored properly.
- INS-3375 Fix for DesAdvancedAnalysis when candidates = 1
- INS-3859 EmbeddedProtocol now clears in/output data from previous attempt before starting new attempt
- INS-3899 EM Probe Station is now supported officially by Perturbation2
- INS-3100 ECDSA now correctly stores keys in intermediates object
- INS-3825 Devices depending on other devices now work properly if underlying device is changed
- INS-3553 Corrected spelling of 'cut off' in Trim module
- INS-2278 Changed Inspector default installation folder to windows system %PROGRAMFILES%
- INS-3098 Camillia now correctly stores keys in intermediates object
- INS-3370 Acquisition trace numbering now properly starts at 0.
- INS-2546 - AES in decrypt mode now starts from round 0 rather than 10/12/14. DES in decrypt mode now starts from round 1 rather than 15/31/47

C Keyboard Shortcuts

The following table presents a list of the most commonly used keyboard shortcuts in Inspector:

Table C.1. Keyboard shortcuts

| Function | Keys |
|----------------------------|-----------------------------|
| Open manual | F1 |
| Close dialog | ESC |
| Zoom (fine) | Arrow up/down (ctrl) |
| Zoom to selection | Enter |
| Previous zoom | Backspace |
| Scroll time axis (fine) | Arrow left/right (ctrl) |
| Scroll traces | PgUp, PgDown |
| First/Last trace | Ctrl+Home, Ctrl+End |
| Lines per trace | Ctrl+PgUp, Ctrl+PgDn |
| Number of traces | Shift+PgUp, Shift+PgDn |
| Overlap | Shift+A |
| Delete/ Copy / Cut / Paste | Del, Ctrl+C, Ctrl+X, Ctrl+V |
| Save current trace set | Ctrl+S |
| Select next trace set | Ctrl+Tab |
| Close current trace set | Ctrl+F4 |
| Open new trace set | Ctrl+O |
| Go to trace | Shift+T |
| Screenshot | Shift+S |
| Current trace set settings | Shift+P |
| Global settings | Ctrl+P |
| Suppress scaling | Ctrl+U |
| Compile module | F9 |

D MP300 Device driver parameters

The following table presents a list of the of MP300 Device driver parameters.

Table D.1. Parameters

| Category | Property name | Description | Allowed values |
|---------------------|----------------------------------|--|---|
| Connection | device.mp300.connectstring | Connection string with the syntax as defined in MP300 documentation. One additional connection method is supported by the Inspector driver, which is native and does not require the MP300 communications driver DLLs to be present: <i>direct:[port]:[ip address or hostname]</i> - uses the internal TCP driver. | string values |
| | device.mp300.timeout | Connection timeout in milliseconds (used by the native driver) | Integer value (default is 1000) |
| RF Field parameters | device.mp300.reqslots | Number of REQ-B slots to reserve | 1,2,4,8 (default is 1) |
| | device.mp300.fieldstrength | Output power of RF unmodulated carrier field as a percentage of the maximum output power | 0-150 (default is 100) |
| | device.mp300.carrier.frequency | Carrier frequency of the RF field, in Hz, with ranges as specified in the MP300 documentation | 13560000 (TCL1/TCL2: 12560000-14560000) |
| | device.mp300.modulation.index | Modulation index of the RF Field as a percentage to output power | not set |
| | device.mp300.modulation.risetime | duration of low-to-high transition in a modulated carrier in nanoseconds | not set |
| | device.mp300.modulation.falltime | duration of high-to-low transition in a modulated carrier in nanoseconds | not set |
| Generic | device.mp300.coupler | Program coupler number to use on MP300. | Integer value (1). |
| | device.mp300.protocol | Selects card protocol to negotiate. Autodetect attempts to select Type A, then Type B. | Autodetect(*), Type A, Type B |
| Triggers | device.mp300.trigger1.mode | Select the trigger mode for trigger out 1. | FORCE, TX_ON, TX_OUT, RX_ON, DELAY_AFTER_TX (*), CARRIER, UNSET |
| | device.mp300.trigger1.delay | (optional) delay value in nanoseconds. When this parameter is set, it overrides the device delay parameter in the device tab. | not set (integer value) |

MP300 Device driver parameters

| | | | |
|--|-----------------------------|---|---|
| | device.mp300.trigger2.mode | Select the trigger mode for trigger out 2 | FORCE, TX_ON (*), TX_OUT, RX_ON, DELAY_AFTER_TX, CARRIER, UNSET |
| | device.mp300.trigger2.delay | (optional) delay value in nanoseconds | integer value (default is 0) |
| | device.mp300.trigger3.mode | Select the trigger mode for trigger out 3 (TCL2 only) | FORCE, TX_ON, TX_OUT, RX_ON, DELAY_AFTER_TX, CARRIER, UNSET(*) |
| | device.mp300.trigger3.delay | (optional) delay value in nanoseconds (TCL2 only) | integer value (default is 0) |

E Modules List

This chapter describes all standard Java modules that come with for and can be supplied with Inspector. The exact modules that are present in your current installation depends on the configuration you purchased (SCA or FI). In the description of every module it is noted with which Inspector version it is delivered, either FI, SCA or both (SCA+FI).

E.1 Classic Acquisition

This section contains the descriptions for all the Classic Acquisition modules. These modules are extensions of the SideChannelAcquisition module.

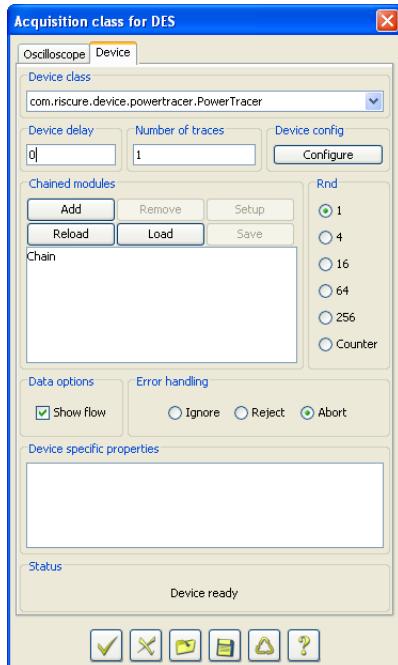
E.1.1 SideChannelAcquisition

Version: SCA+FI

Purpose

To collect side channel information from a device. This module is application-independent, and cannot be executed directly. It must be extended for a specific application.

Dialog



Input

The *Oscilloscope* tab is explained in the help file for the Oscilloscope module. The following additional controls are provided:

- The *Reader class* panel allows the selection of a device controller (e.g. a smart card reader). Devices can be used if a Java class is available that implements the Device interface. Riscure provides several classes:
 - com.riscure.device.powertracer.PowerTracer: driver for the Power Tracer.
 - com.riscure.device.serialport.SerialPortDevice: driver that connects directly to the serial port.
 - com.riscure.device.pcsc.PcscReader: driver that connects to any attached PC/SC reader.
 - com.riscure.device.rftracer.RfTracer: driver for Riscure's RF Tracer to test contactless cards.
 - com.riscure.device.dummy.NullDevice: An empty driver that ignores all data.
 - com.riscure.device.echo.EchoDevice: An test driver that returns all sent data.
- The *Device delay* edit field includes a delay, expressed in microseconds, that is applied by the device before generating a trigger signal for the scope. This delay is supplementary to the scope delay in the left half of the dialogue which defines a delay for the scope as a percentage of its window. The device trigger delay value is forwarded to the device driver using the arm method, and should be processed by the actual Device implementation.
- The *Number of traces* edit field selects the number of times the process must be repeated.
- The *Configure* button can be used to perform device specific configuration in a separate dialog.
- The *Chained modules* panel allows the configuration of a filter list. In this way the acquired signals can be filtered (for instance low-pass, resampling etc), on the fly. The modules in the chained module list will be applied as described in Section 4.7. The two additional buttons Load and Save serve to load a chain file or store the current parameters in a chain file.
- The *Device specific properties* panel allows additional properties to be set for a device. The format is *property = value*, where properties are separated by semicolons (';').
- The *Options* panel selects whether the application protocol data flow is reported in the output window.
- The *Error Handling* panel selects the action to be taken in the event of an unexpected device response. Correctness of response is verified by the 'test' method in the SideChannelAcquisition class. The following responses (status words) are accepted: 0x61XX, 0x6CXX, 0x91XX, 0x9FXX, and 0x9000.

Background

This module extends the Oscilloscope module to run a session in which a SIM is repeatedly requested to perform an operation which is subsequently recorded by means of the oscilloscope. This module can control the external device through the so-called Device interface. This interface allows the module to start the operation to be monitored and trigger the oscilloscope at the appropriate time.

Acquisition modules that extend this module are DES Acquisition, AES Acquisition, ECC Acquisition, RSA Acquisition and GSM Acquisition.

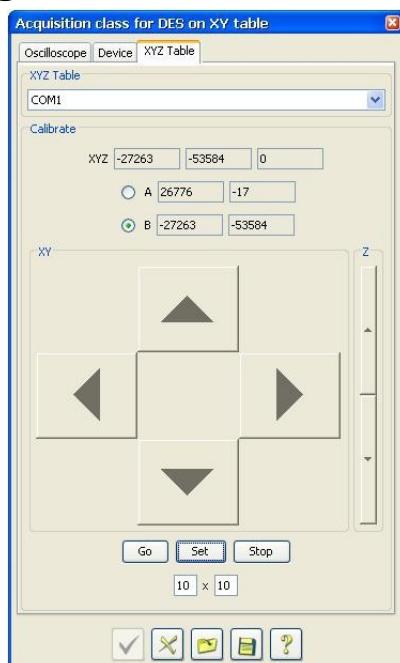
E.1.2 XYAcquisition

Version: SCA

Purpose

The *XYAcquisition* module allows you to set all the parameters for an EM scan of a chip area and start the scan. This module is used as a base class for the other modules like the *DESXYAcquisition*, *AESXYAcquisition* and the *RSAXYAcquisition*. The result of the EM scan can be presented and analyzed in the *SpectralIntensity* module. The analyst may select the optimum position for the EM probe based on the analysis of the EM scan. With the EM probe at this location, the analyst can acquire the trace set that is necessary for SEMA or DEMA.

Dialog



Input

The input is grouped in three panels: oscilloscope (left), acquisition (centre) and XYZ table (right). Help on the left and centre panels can be found in the help files for the 'Oscilloscope' and 'DESAcquisition' modules.

- The XYZ Table pull down menu only shows the controller TMCM310 of the XYZ table after a connection between the controller and the PC is established. For this purpose, the controller and the PC must be connected via the serial cable and the controller must be powered up by connecting the controller power supply. Select the TMCM310 motor controller in the pull down menu. It should be possible to move the EM probe with the movement control buttons
- Pressing the XY movement control buttons (\leftarrow , \rightarrow , \uparrow , \downarrow in the XY panel) moves the EM probe in a horizontal plane. Caution: the EM probe tip can be damaged if the tip is pressed against an object by the motors.

- Pressing the Z movement control buttons ($\uparrow \downarrow$ in the Z panel) moves the EM probe up and down. Caution: the EM probe tip can be damaged if the tip is pressed against the chip by the motors.
- The calibrate panel is used to calibrate the corner positions of scan area. The following procedure must be followed:
 - Move the probe tip to corner 'A' of the scan area using the XY movement control buttons.
 - Select corner 'A' for calibration by pressing the radio button next to 'A'.
 - Press the 'set' button. The text fields next to 'A' fill with the X and Y position of corner 'A'.
 - Move the probe tip diagonal to the other corner 'B' of the scan area using the XY movement control buttons.
 - Select corner 'B' for calibration by pressing the radio button next to 'B'.
 - Press the 'set' button. The text fields next to 'B' fill with the X and Y position of corner 'B'.
 - At the bottom of the dialog window the number of rows and columns [X] x[Y] can be entered. Press <enter> after completion.

The calibration procedure is now completed. By inserting the SpectrallIntensity module in the chain, the results of the scan will be presently directly by the SpectrallIntensity module.

Results

The XYAcquisition results in a trace set of X x Y traces (X = # of rows; Y = # of columns). Each trace is measured at one position of the X x Y grid. To combine the measurement at a single grid position with an encryption, use the DESXYAcquisition, AESXYAcquisition or the RSAXYAcquisition module. The SpectrallIntensity module can be used the present and analyse the resulting trace set.

WARNING! When modifying or extending XY Acquisition modules, you SHOULD NOT call setData from the runAcquisition method. The SpectrallIntensity module depends on this data to move the probe to the correct location.

E.1.3 AES Acquisition

Version: SCA

Purpose

To collect side channel information from a device with an AES implementation. For details, see the generic SideChannelAcquisition module. This module can be used to collect multiple traces of the same process in order to perform a subsequent differential analysis.

Background

The module can be used with training card 3 [[..../tutorials/sampletype3.html](#)] [[PDF](#)] [[..../tutorials/Tutorials.pdf#sampletype3](#)] and training card 4 [[..../tutorials/sampletype4.html](#)] [[PDF](#)] [[..../tutorials/Tutorials.pdf#sampletype4](#)].

Other acquisition modules are ECC Acquisition, DES Acquisition, RSA Acquisition and GSM Acquisition.

Inputs

The module code is quite compact, as it only contains application specifics. The analyst may need to rewrite the three methods to conform to the application protocol used for his application. The first method *initAcquisitionModule* describes the module; the second method *initAcquisitionProcess* performs all actions needed before batch processing can start (for example: application selection); the third method *runAcquisition* performs the actions needed to collect one trace, and may be invoked repeatedly. See the DES Acquisition documentation for more details.

E.1.4 DES Acquisition

Version: SCA

Purpose

To collect side channel information from a device with a DES implementation.

For details, see the generic SideChannelAcquisition module. This module can be used to collect multiple traces of the same process in order to perform a subsequent differential analysis.

Background

The module can be used with training card 1 [..../tutorials/sampletype1.html] [PDF] [..../tutorials/Tutorials.pdf#sampletype1], training card 2 [..../tutorials/sampletype2.html] [PDF] [..../tutorials/Tutorials.pdf#sampletype4] and training card 4 [..../tutorials/sampletype4.html] [PDF] [..../tutorials/Tutorials.pdf#sampletype4].

Other acquisition modules are ECC Acquisition, AES Acquisition, RSA Acquisition and GSM Acquisition

Inputs

The module code is quite compact, as it only contains application specifics. The analyst may need to rewrite the three methods to conform to the application protocol used for his application. The first method describes the module; the second method performs all actions needed before batch processing can start (for example: application selection); the third method performs the actions needed to collect one trace. The current method implementations are:

```
/** initialize module description here
*/
protected void initAcquisitionModule() {
    moduleTitle = "Acquisition class for DES";
    moduleDescription = "Acquisition of DES traces for Riscure's test applet";
    moduleVersion = "1.4";
}
/** initial processing before entering the main loop */
protected void initAcquisitionProcess() {
    // select application
    response = command("00 a4 04 00 07 A0 00 00 00 FF F0 01");
```

```
}

/** actions repeated in the loop */
protected void runAcquisition() {
    arm(); // arm the oscilloscope
    // the next command uses randomized data
    nextCommand = randomize("A0 04 00 00 08 C7 39 D7 EA FA E4 ED A3 00", 5, 8 );
    // save the data included in the APDU
    setData( nextCommand, 5, 8 );
    response = command( nextCommand ); // send the APDU
    // save the response data
    addData( response, 0, 8 );
}
```

E.1.5 DSA Acquisition

Version: SCA

Purpose

To collect side channel information from a device with a DSA implementation. For details, see the generic SideChannelAcquisition module. This module can be used to collect multiple traces of the same process in order to perform a subsequent differential analysis.

Other acquisition modules are ECC Acquisition, DES Acquisition, RSA Acquisition and GSM Acquisition.

Inputs

The module code is quite compact, as it only contains application specifics. The analyst may need to rewrite the three methods to conform to the application protocol used for his application. The first method *initAcquisitionModule* describes the module; the second method *initAcquisitionProcess* performs all actions needed before batch processing can start (for example: application selection); the third method *runAcquisition* performs the actions needed to collect one trace, and may be invoked repeatedly. See the DES Acquisition documentation for more details.

E.1.6 ECC Acquisition

Version: SCA

Purpose

To collect side channel information from a device with ECC algorithm implementation. For details, see the generic SideChannelAcquisition module. This module can be used to collect multiple traces of the same process in order to perform a subsequent differential analysis.

Background

The module acquires data from the training card 4 [..../tutorials/sampletype4.html] [PDF] [..../tutorials/Tutorials.pdf#sampletype4].

Other acquisition modules are AES Acquisition,DES Acquisition, RSA Acquisition and GSM Acquisition.

Inputs

The module code is quite compact, as it only contains application specifics. The analyst may need to rewrite the three methods to conform to the application protocol used for his application. The first method *initAcquisitionModule* describes the module; the second method *initAcquisitionProcess* performs all actions needed before batch processing can start (for example: application selection); the third method *runAcquisition* performs the actions needed to collect one trace, and may be invoked repeatedly. See the DES Acquisition documentation for more details.

E.1.7 ECDSA Acquisition

Version: SCA

Purpose

To collect side channel information from a device with a ECDSA implementation. For details, see the generic SideChannelAcquisition module. This module can be used to collect multiple traces of the same process in order to perform a subsequent differential analysis.

Other acquisition modules are ECC Acquisition, DES Acquisition, RSA Acquisition and GSM Acquisition.

Inputs

The module code is quite compact, as it only contains application specifics. The analyst may need to rewrite the three methods to conform to the application protocol used for his application. The first method *initAcquisitionModule* describes the module; the second method *initAcquisitionProcess* performs all actions needed before batch processing can start (for example: application selection); the third method *runAcquisition* performs the actions needed to collect one trace, and may be invoked repeatedly. See the DES Acquisition documentation for more details.

E.1.8 Gsm Acquisition

Version: SCA

Purpose

To collect side channel information from a GSM SIM.

For details, see the generic SideChannelAcquisition module. This module can be used to collect multiple traces of the same process in order to perform a subsequent differential analysis.

Background

The module acquires data from a SIM device.

Other acquisition modules are ECC Acquisition, AES Acquisition, DES Acquisition and RSA Acquisition

Inputs

The module code is quite compact, as it only contains application specifics. The analyst may need to rewrite the three methods to conform to the application protocol used for his application. The first method *initAcquisitionModule* describes the module; the second method *initAcquisitionProcess* performs all actions needed before batch processing can start (for example: DF selection, PIN verification); the third method *runAcquisition* performs the actions needed to collect one trace, and may be invoked repeatedly. See the DES Acquisition documentation for more details.

E.1.9 IOCTL Acquisition

Version: SCA

Purpose

This module implements an example protocol that can be used by a driverless USB token. For details, see the generic SideChannelAcquisition module. This module can be used to collect multiple traces of the same process in order to perform a subsequent differential analysis.

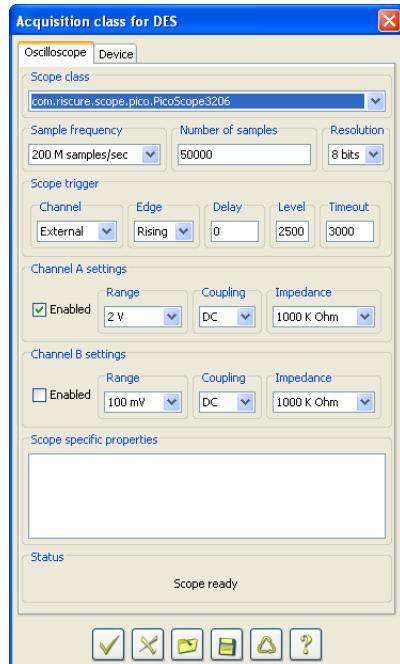
E.1.10 Oscilloscope

Version: SCA+ FI

Purpose

To control signal acquisition equipment.

Dialog



- **Input**The *Scope class* panel is used to select a data acquisition device. Devices can be used if a java class is accessible that implements the Scope interface. Riscure has developed four implementations:
 1. **SoundCardScope**: An implementation that uses the sound card device for data acquisition. This device is typically limited to low acquisition speeds.
 2. **PicoScope3206**: An implementation that uses the PicoScope model 3206. This device has two acquisition channels and a maximum sampling frequency of 200 MHz.
 3. **PicoScope5203**: An implementation that uses the PicoScope model 5203. This device has two acquisition channels and a maximum sampling frequency of 1 GHz.
 4. **GageScope**: An implementation that uses Gage oscilloscopes.
 5. **LeCroyScope**: An implementation that can access any LeCroy oscilloscope equipped with an Ethernet port and the XDEV package to support the fast wave port used to exchange the sampled data. More information on the use of LeCroy can be found [here](#).
- The *Sample frequency* panel is used to select a sampling speed supported by the selected oscilloscope device.
- The *Number of samples* panel is used to select the number of samples to be acquired.
- The *Trigger* panel controls the various trigger parameters:
 1. *Channel*: Specifies the channel that listens for the trigger signal.
 2. *Edge*: Specifies whether triggering occurs on the rising (positive) edge, or falling (negative) edge.
 3. *Delay*: Specifies a positive or negative delay in micro seconds in signal recording after the trigger signal occurred. The maximum delay value is limited to the scope window (number of samples/ sampling speed).
 4. *Level*: The trigger threshold expressed in millivolts.
 5. *Timeout*: The maximum time the device waits for the trigger signal before triggering automatically.
- The *Resolution* panel is used to select the number of bits to be used for signal acquisition.
- The *Impedance* panel is used to select the input impedance of the acquisition channels.
- The *Channel A* settings panel controls various channel acquisition parameters for channel A:
 1. *Enabled*: Specifies whether the channel is used for signal acquisition.
 2. *Range*: The maximum voltage that can be measured on the channel.
 3. *Coupling*: Selects AC or DC acquisition

- The *Channel B* settings panel controls the channel acquisition parameters for channel B.
- The *Status* panel shows whether the oscilloscope is ready for data acquisition. Reasons for not being ready include absence of the device, and the need to reset the device. The status is linked to the OK button, so the acquisition process cannot be started if the device is not ready.

Analysts can add their own implementations of the Scope [..]/avadoc/com/riscure/scope/Scope.html] interface and provide them either in a jar file in the lib folder under the Inspector home directory, or as a class file in a package path under this lib folder. Inspector automatically reads the jar files and classes in this lib folder and adds them to the Scope class selection box.

E.1.11 RSA Acquisition

Version: SCA

Purpose

To collect side channel information from a device with an RSA implementation. For details, see the generic SideChannelAcquisition module. This module can be used to collect multiple traces of the same process in order to perform a subsequent differential analysis.

Background

The module acquires data from the training card 1 [..]/tutorials/sampletype1.html] [PDF] [..]/tutorials/Tutorials.pdf#sampletype1].

Other acquisition modules are ECC Acquisition, AES Acquisition, DES Acquisition and GSM Acquisition

Inputs

The module code is quite compact, as it only contains application specifics. The analyst may need to rewrite the three methods to conform to the application protocol used for his application. The first method *initAcquisitionModule* describes the module; the second method *initAcquisitionProcess* performs all actions needed before batch processing can start (for example: application selection); the third method *runAcquisition* performs the actions needed to collect one trace, and may be invoked repeatedly. See the DES Acquisition documentation for more details.

E.1.12 Sasebo Acquisition

Version: SCA

Purpose

To collect side channel information from a SASEBO R/G/G-II board.

Dialog



Input

The *Oscilloscope* tab is explained in the help file for the Oscilloscope module, while the *Device* tab is described in the help file for the SidechannelAcquisition module.

The SASEBO tab allows the user to choose between the different options offered by the SASEBO boards. The following settings are available:

- *Algorithm*: The cryptographic algorithm to be executed. Keep in mind that not all algorithms are supported by a particular SASEBO board. Read the Background information section to find out which algorithms are supported by your board.
- *Operation*: When the selected implementation supports encryption and decryption, it is possible to choose between them using these buttons. If only encryption is supported, these buttons are disabled.
- *Key*: The key for symmetric algorithms can be entered in this field. When RSA is selected, this field is disabled.
- *RSA Modulus and RSA Exponent*: When RSA is selected, it is possible to enter the RSA modulus and the RSA exponent in these fields. Otherwise, these fields are disabled.

Background information

There are 5 different flavours of SASEBO boards, all of them designed by RCIS (AIST) [<http://www.rcis.aist.go.jp>] in Japan. Four of these boards are based on FPGAs (SASEBO, SASEBO-G, SASEBO-B and SASEBO-GII), while another one is based on a custom cryptographic LSI (SASEBO-R).

The FPGA versions are configured with an AES implementation, while the SASEBO-R includes all the supported algorithms in the LSI. Additionally, Verilog sources for AES and DES are provided by RCIS for the SASEBO, SASEBO-G and SASEBO-GII; users might choose to integrate other implementations in them as well.

Beware that the provided module is a general module and does not check the availability of the requested algorithms. The only check performed by the module is whether there is communication with the serial port, and if the reported version is the exported version of the Sasebo-R board. Since this particular version has some bits of the key fixed to certain values, the module checks those bits and issues a warning informing the user of the actual key that will be used by the board.

Therefore, depending on your board and the uploaded configuration in the case of FPGA based boards, the available implementations might vary. In case a not implemented algorithm is supported, you will most likely see a response consisting of all zeros for every encryption/decryption request.

For more information on the SASEBO boards, see <http://www.rcis.aist.go.jp/special/SASEBO/index-en.html>.

E.1.13 USB Token Acquisition

Version: SCA

Purpose

This module implements an example protocol that can be used by a USB token. For details, see the generic SideChannelAcquisition module. This module can be used to collect multiple traces of the same process in order to perform a subsequent differential analysis.

E.2 Acquisition2

This section contains the descriptions for all the acquisition2 modules.

E.2.1 Acquisition2 Modules

Version: SCA

Purpose

To collect side channel information from a target. Acquisition2 currently contains two modules, ScopeAcquisition and XYAcquisition. both modules can be used to collect multiple measurements of the same process in order to perform a subsequent side channel analysis.

Input

The ScopeAcquisition module combines target control through a user-selectable application protocol with an oscilloscope measuring the target. For XYAcquisition, most of the settings are identical to ScopeAcquisition except for additional hardware configuration and measurement

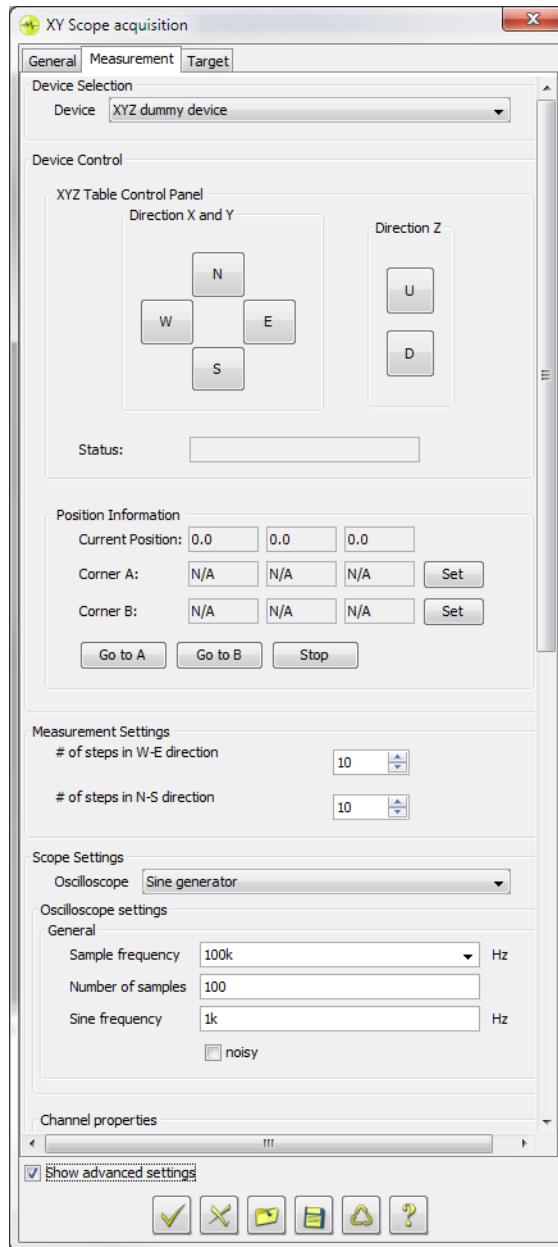
settings introduced by a hardware device called the XY Stage. Note that some additional options are available when the user checks the *Show advanced settings* option at the bottom of the module frame.

General Acquisition settings

- *Number of measurements (ScopeAcquisition)*: how many accepted measurements to take before terminating the acquisition. Any non-accepted measurements (e.g. due to error) are not counted.
- *Accept measurements with errors*: whether measurements that were taken in error conditions on the target or on the measurement setup should be accepted and stored. If this option is unchecked, a retry will be executed.
- *Limit errors*: Limit the number of consecutive errors allowed during acquisition or running the target.
- *Maximum consecutive errors*: how many consecutive errors to allow before aborting the measurement. If 0, the measurement will be aborted immediately upon an error. For any larger number, that number of retries which result in an error are allowed before aborting. (Only applicable when *Limit errors* is checked.)

Measurement setup

Figure E.1. Measurement tab of XYAcquisition module

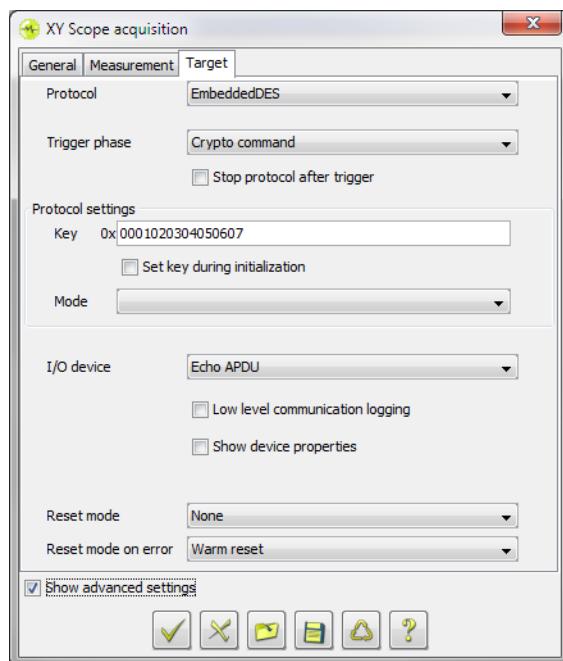


- **Device Selection (XYAcquisition):** The XY Device, e.g. EM Probe station, to use for this acquisition.
- **Device Control (XYAcquisition):** The XY Stage setting panel has two purposes. Firstly, to manipulate the Stage through the control panel. There are buttons for all four X and Y directions: N(North), E(East), W(West) and S(South). If the device facilitates this, there are U(Up) and (D)Down buttons for controlling the Z-axis. Secondly, to set Corner A and corner B for acquisitions within a certain area.

- **Measurement Settings (XYAcquisition):** The XY Stage will move from Corner A to Corner B following a row-by-row pattern. Thus the device will eventually cover a rectangular area. This setting concerns the number of stops the Stage should make in the directions of N(North), E(East), W(West) and S(South) within this rectangular area. There will be one measurement acquired per stop.
- **Oscilloscope:** The oscilloscope to use for this acquisition.
- **Oscilloscope settings:** Oscilloscope-specific settings.
- **Channel properties:** Allows setting the properties per channel of the oscilloscope.

Target

Figure E.2. Target tab of XYAcquisition module



- **Protocol:** The application protocol to use when communicating with the target
- **Trigger phase:** Phase within the application protocol to arm the measurement setup on. The following trigger (caused by a communication or reset event) will cause the measurement to start. With some protocols there may be more than one potential moment to measure, which can be selected here. See also Protocol Phases [57].
- **Stop protocol after trigger:** Whether to stop the protocol after the trigger, and not complete the remaining phases. Only use this if it is safe to abort the protocol flow (e.g. it does not result in an increase in some error counter)
- **I/O device:** The I/O device to which the target is connected.
- **Reset mode:** The "cold reset" means that the smart card is reset before starting a perturbation attempt by disabling the power supply and pulling down the reset line. The "warm reset" only pulls down the reset line. "None" means that no reset is performed between measurements.

- *Reset mode on error*: Like "Reset mode", but only performed when an error is detected in the smartcard communication.



Note

The "Reset on error" mode should always be equal or stronger than the normal reset mode and the UI enforces this. For example when the user selects "reset=cold", the "reset on error" field automatically switches from "none" to "cold". If the user downgrades the field to "reset on error=warm" the "reset" field automatically switches to "warm".

This module can be inserted in a chain by using the Chain module. It must be inserted in the first position of the chain. Subsequent modules in the chain will process traces acquired using the Acquisition2 modules.



Note

The acquisition framework was designed for using the Power Tracer (see Section 6.3.2, "Power Tracer 4"). However, it is possible to use the VC Glitcher (see Section 6.4.1, "VC Glitcher"). In that case the clock frequency will be fixed to 4MHz and the VCC is set to 5V. Also keep in mind that the Power Tracer produces cleaner measurement results due to its capacitors, which provide a stable VCC line. An alternative option when only a VC Glitcher device is available (and the VCC and/or clock settings need to be set manually) is to use a Perturbation module with glitch voltage 0.

E.3 Align

This section contains the descriptions for all the alignment modules.

Static alignment, being the simplest and fastest of the align modules, is the usual starting point for alignment of a trace set. If the misalignment present in the measurements is too high or randomized (e.g. by random process interrupts), static alignment is often not enough and elastic alignment needs to be used.

If elastic alignment doesn't succeed, then the more costly dynamic alignment technique might be useful. This technique should be more noise-resistant than the previous ones.

Note that when using two instances of Align in a Chain in Store-Apply mode, you should select the same tab for both instances. Otherwise, the Align module in Apply mode will not be able to find the right parameters for the alignment and no output will be produced.

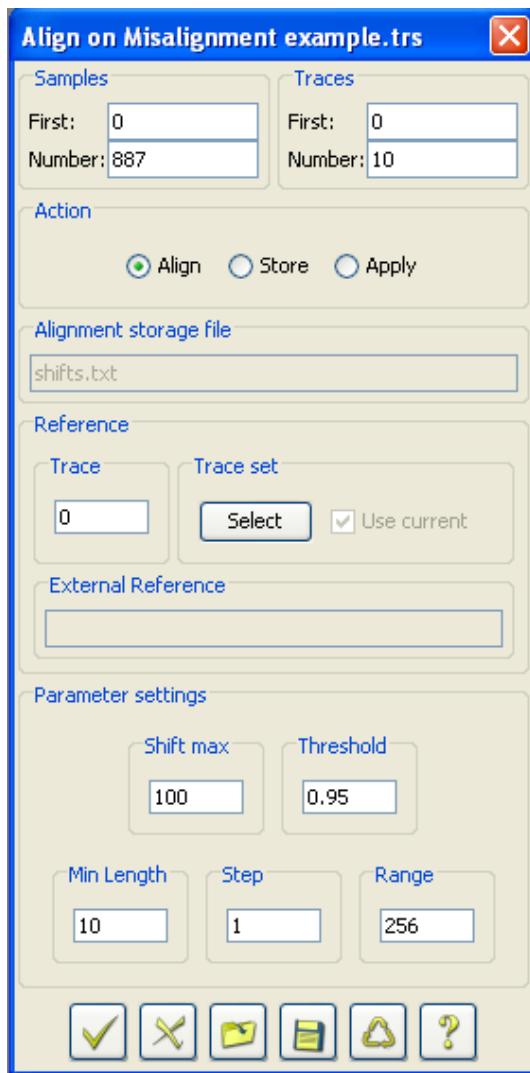
Additionally, note that when Static or Dynamic alignment are used inside Chain, you cannot enter the reference pattern by selecting it in the input trace set. In that case, you will need to load Chain with no selection at all and enter the reference pattern manually after loading the Align module.

As an example, in the Filter Example Tutorial [..../tutorials/sectionsHardwareFilterTutorial.html] [PDF] [..../tutorials/Tutorials.pdf#sectionsHardwareFilterTutorial] Static Alignment is used in Store and Apply mode inside a Chain.

E.3.1 Dynamic Alignment

This process starts by performing a Static Alignment (see above). After static alignment has completed, it performs repeated shifts to achieve continuous alignment with the reference trace

Dialog



Input

Besides the parameters available for Static Alignment, the dynamic alignment process requires several parameters that must be carefully chosen for optimal quality and speed.

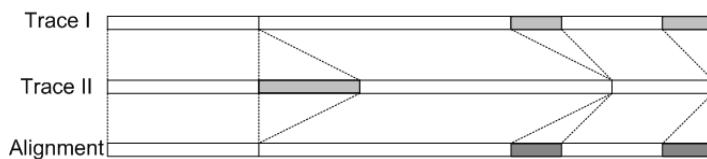
- *Min length*: Dynamic alignment assumes that reference-trace and trace-to-align have corresponding fragments that got desynchronized e.g. due to wait states or variable frequency. First you have to select a fragment length (Min Length) of the traces for which you expect similar fragments, e.g. for wait states this would be the number of samples in between two sequential wait states.
- *Step*: Step is the number of samples to skip when the next option is tried. In the case of wait states it would make sense to set step equal to the number of samples that a wait state takes. When you use a step size bigger than 1 you will notice a much faster performance,

but you take the risk of 'stepping' over the best match. If the nature of the misalignment cause is unknown, it is safe to set the step value to 1.

- **Range:** Range is a bigger part of the trace that you will use for one dynamic alignment cycle. Within range samples the algorithm will try to find the best 'path' where each solution is tried. Ideally you would make range as large as possible, but this degrades the performance too much. A value of 128 is often a good choice for range

Background

Dynamic alignment proceeds after static alignment, and uses the fragment used for static alignment. It is therefore important that the static alignment result in a reasonable quality. Dynamic alignment can be complicated and can only be achieved by a trial-and-error process in which you tune the parameters until the alignment is satisfactory. Dynamic alignment assumes that the reference trace and the trace to be aligned have corresponding fragments that got desynchronized, e.g. due to random process interrupts or variable frequency. To better understand the dynamic alignment principle, consider the following example.



We have two traces, which resemble each other to some extent. That is, the traces can be partitioned into common fragments, which show some degree of similarity, and differing fragments. In the picture above, the resemblance between the common fragments is marked with dotted lines, and the differing fragments are marked grey. There are three common fragments, and three differing fragments:

- Trace II contains one fragment not present in Trace I
- Trace I contains two fragments not present in Trace II

Suppose Trace II is to be aligned with Trace I. As a first step, alignment can be constructed from the common fragments. Secondly, the single fragment present in Trace II but absent in Trace I can be left out. Finally, filler fragments have to be inserted to compensate for those fragments present in Trace I but absent in Trace II. In theory, there are a number of possible choices for the contents of these filler fragments. For instance, the filler can be constructed from samples of value 0, or from the corresponding samples from Trace I. It has been chosen to repeat part of the previous fragment, in order not to disturb the continuity of the trace signal.

Result

The Dynamic Align process produces output detailing the results per trace.

```
Including trace 0, shift: 0, correlation: 1.0
Including trace 1, shift: -563, correlation: 0.990467
Including trace 2, shift: -462, correlation: 0.990083
Including trace 3, shift: -552, correlation: 0.987422
Including trace 4, shift: -204, correlation: 0.992715
Including trace 5, shift: -428, correlation: 0.994385
Excluding trace 6, correlation: 0.141
Including trace 7, shift: -867, correlation: 0.958981
```

Including trace 8, shift: -337, correlation: 0.994892

The output shows that trace 0 has a perfect correlation; this is because this trace was chosen as the reference.

Success is not always guaranteed if the traces are very noisy. In that case it is recommended to filter the traces prior to alignment. The filtering could be based upon moving average or spectral filtering. Even when signals are extremely noisy, a very low-pass filter will reveal a trend in the signal that can be used for alignment. However, the filtering process could lead to signal loss. Therefore the module allows the alignment shifts computed for a filtered signal to be stored. These shift values can be reused later with the Shift module to apply to the unfiltered signal. In this way the original signal can be aligned without losing spectral information.

E.3.2 Elastic Alignment

Elastic alignment differs from static and dynamic alignment by using local compression and stretching of traces to perform the synchronization, and the absence of the necessity to select a reference pattern.

Dialog



Input

- *Action*: The main action to be executed by the module.
- *Align* aligns the trace set to the specified reference trace.
- *Store* stores the alignment information of the trace set with respect to the specified reference trace to a file.
- *Apply* uses the alignment information stored in a file to align the trace set.
- The *alignment storage file* allows specifying in which file alignment information should be stored or from which file it should be loaded. This is therefore only applicable to the *store* and *apply* actions.
- *Reference*: Reference trace selection options. Only applicable to *align* and *store* actions.
 - *Trace#* specifies the reference trace index in the reference trace set.
 - *Trace set* allows selecting between using a reference trace from the current trace set, or from an external trace set file.
 - *External reference* indicates which external trace set has been selected, or is empty if the current trace set is used.
- *Auto tune* selects whether the alignment algorithm should find a good value for the *Radius* and *Window* parameters automatically.
- *Radius* is a parameter to the underlying trace matching algorithm. It is a trade off between speed (lower values) and quality (higher values) of the alignment.
- *Window* determines the context window to use for matching pairs of traces to determine a proper alignment. At 0, only individual samples between two traces are matched; at e.g. 5 the five neighbouring samples to both the left and right are also matched. It is a trade off between speed (lower values) and quality (higher values) of the alignment.
- *Minimum correlation* allows filtering traces based on the amount of misalignment. If the correlation between the reference trace and the aligned trace is lower than the threshold value, the aligned trace is rejected.

Result

The trace set output is elastically aligned to the specified reference trace. Normally, the output trace set has the same number of samples as the input trace set. However, if the action is *align* and an external reference trace is selected, this external trace set determines the number of samples in the output trace set.

When performing the *align* or *store* actions, the ElasticAlign module outputs the correlation of each aligned trace with the reference trace, and whether this correlation is sufficient for the trace to be included. An example output is given below:

```
Trace 0 is reference trace
Trace 1 included (0.95243174)
Trace 2 included (0.9510317)
Trace 3 included (0.9694323)
Trace 4 included (0.950665)
```

```
Trace 5 included (0.95986533)
Trace 6 excluded (0.80056)
Trace 7 included (0.92126405)
Trace 8 included (0.94199806)
Trace 9 included (0.9470649)
```

Background

We envision a number of use cases for this module:

- *Simple alignment*: align all traces in the set to one reference trace by using action *align*.
- *Alignment to external reference trace*: align all traces in the set to one externally specified reference trace. When doing this, the traces can be aligned to a preprocessed trace. This preprocessed trace can e.g. be the output of Elastic Average, and therefore does not bias the traces to the timing of one particular raw trace.

Another advantage is that the external reference trace can be of a different length; the output trace is resampled to the length of the external reference trace. This allows more advanced operations such as aligning to a trace where interrupts and wait states have been cut out.

- *Alignment at a lower resolution*: align traces based on alignment information obtained at a lower resolution. This has the advantage that alignment information is calculated based on slower processes, and fast variations in the traces do not influence the alignment. This type of alignment is performed by storing alignment information obtained from a trace set at lower resolution, and applying the alignment to the original trace set. Note that the alignment storage file potentially grows as large as the input trace set. However, compression is applied such that typically this file does not grow to overly large sizes.

The ElasticAlign module offers a number of quality versus speed trade offs. Finding the optimal settings depends on the type of alignment problem.

The *auto-tune* feature searches for good values for *radius* and *window* within a certain parameter range. This is a time-consuming step, but it saves the analyst the task of tuning the parameters. Currently, the selected parameter is a trade off of computational time versus quality of the output. Turning off this feature allows more precise control. For more information on auto-tuning see the Elastic Radius module.

The *radius* parameter determines the width of the search space. The higher its setting, the larger the space the algorithm searches to find a good alignment. The *window* parameter determines how many samples are used when matching individual locations in two traces. If set to 0, only the sample at one location is used; if set to *x*, a window size of $2x+1$ samples is used. The *radius* and *window* parameters can be determined using the *auto-tune* feature or the Elastic Radius module.

The memory usage of the module is related to these parameters and the length of the trace set, and if the parameters are set too high, the module may switch to using a swap file for its calculations. This has a performance impact.

The length of a trace set may also be a limiting factor. This can be overcome by determining the alignment on a lower resolution representation of the trace set, and applying the alignment on the original trace set.

Note that the expansion and compression steps in the alignment algorithm respectively repeat and average and samples. This implies there is some loss of precision in the sample values after alignment, depending on the amount of elastic adjustment needed.

Experiments show that preprocessing a trace set may help the final elastic alignment. See PatternPad for more information. PatternPad works because the fundamental component in elastic alignment presumes the beginning sample and ending sample of each pair of traces are aligned.

Related tutorials

- **Elastic alignment tutorial** [../tutorials/sectionsElasticAlignTutorial.html] [PDF] [../tutorials/Tutorials.pdf#sectionsElasticAlignTutorial]

References

The full elastic alignment algorithm is designed based upon the research in [1].

[1] Jasper van Woudenberg, "Elastic Alignment of Power Traces for Differential Power Analysis of Smart Cards", to be published.

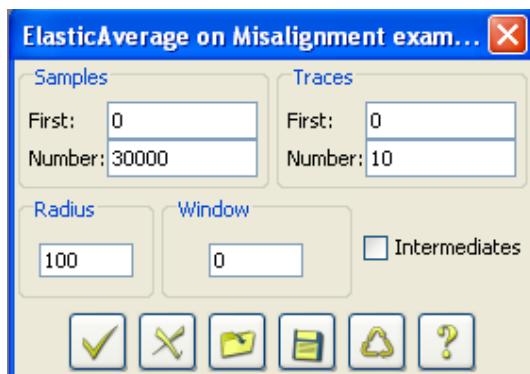
E.3.3 ElasticAverage

Version: SCA

Purpose

Produce an average trace for a trace set by performing repeated elastic alignment and averaging of trace pairs. The resulting average is not biased with respect to any single trace, and can therefore be used as reference trace to elastically align a larger set.

Dialog



Input

- *Radius*

is a parameter to the underlying trace matching algorithm. It is a trade off between speed (lower values) and quality (higher values) of the alignment.

- *Window*

determines the context window to use for matching pairs of traces to determine a proper alignment. At 0, only individual samples between two traces are matched; at e.g. 5 the five neighbouring samples to both the left and right are also matched. It is a trade off between speed (lower values) and quality (higher values) of the alignment.

- *Intermediates*

specifies whether intermediate averages should be output as well. This can be used for tracking the constituents that make up the final average.

Result

The resulting average is a trace whose time base is not biased towards a single trace. This module hierarchically averages traces; depending on the setting for *intermediates*, all intermediate averages are also output.

Background

The elastic alignment algorithm needs a reference time base needs to be specified. For the Elastic Align module, this time base is given by the reference trace, which implies there is a bias towards this trace's time base. This module tries to overcome this problem by generating an average trace which has no bias towards a particular trace.

This module calculates an average trace by performing repeated pairwise elastic aligning and averaging. As an example, consider a trace set with 4 traces. First, traces 0 and 1 are aligned and averaged, producing trace A01. Next, traces 2 and 3 are aligned and averaged to produce trace A23. The final result is obtained by repeating this process for A01 and A23, producing A0123.

See Elastic Align for a further explanation of the *window* and *radius* parameters.

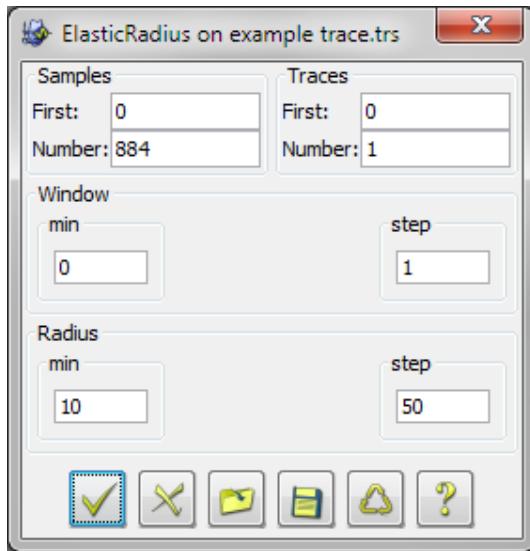
E.3.4 ElasticRadius

Version: SCA

Purpose

Determine suitable parameters for performing a proper Elastic Align or acquiring a proper Elastic Average.

Dialog



Input

This module expects a minimum value and a step size for both the *window* and *radius* parameters. These parameters will be iterated starting at the *min* parameter value, each time incremented with the *step* value, until the best pair of parameters is found. See Elastic Align for a further explanation of the *window* and *radius* parameters.

Result

The module outputs for each pair of traces a number of values given certain *window* and *radius* choices. For example:

```
Aligning Trace 0 and 1
Radius=010, window=000, correlation=0.931, time=4.140, criterion=3.722537e+05,
DTWdist=2.609690e+05, DTWpathlen=32664 (new best)
Radius=060, window=000, correlation=0.956, time=4.782, criterion=2.873631e+05,
DTWdist=1.943250e+05, DTWpathlen=31603 (new best)
Radius=110, window=000, correlation=0.960, time=7.250, criterion=2.937047e+05,
DTWdist=1.789890e+05, DTWpathlen=31416
Radius=160, window=000, correlation=0.950, time=9.844, criterion=3.552260e+05,
DTWdist=2.005450e+05, DTWpathlen=31828
Radius=010, window=002, correlation=0.871, time=2.719, criterion=4.227111e+05,
DTWdist=3.291860e+05, DTWpathlen=31809
Radius=060, window=002, correlation=0.889, time=6.984, criterion=4.741669e+05,
DTWdist=2.916790e+05, DTWpathlen=31975
Radius=110, window=002, correlation=0.926, time=11.406, criterion=3.951796e+05,
DTWdist=2.150360e+05, DTWpathlen=31465
Radius=160, window=002, correlation=0.928, time=16.031, criterion=4.296319e+05,
DTWdist=2.147120e+05, DTWpathlen=31459
Radius=010, window=004, correlation=0.861, time=3.156, criterion=4.449843e+05,
DTWdist=3.338570e+05, DTWpathlen=31931
Radius=060, window=004, correlation=0.920, time=16.750, criterion=4.500993e+05,
DTWdist=2.224870e+05, DTWpathlen=31365
```

```
Radius=110, window=004, correlation=0.944, time=16.609, criterion=3.582885e+05,
DTWdist=1.774790e+05, DTWpathlen=31044
Radius=160, window=004, correlation=0.925, time=27.000, criterion=4.726353e+05,
DTWdist=2.073410e+05, DTWpathlen=31497
Best tradeoff: radius=60, window=0
```

- *Correlation*

: the sample correlation between the reference trace and the elastically aligned trace.

- *Time*

: the wall-clock time elapsed during the alignment computation. Note this is influenced by other processes running on the system and is therefore may include some jitter.

- *Criterion*

: the auto-tune criterion. This is the result of a formula involving time and output quality. The lowest value of this criterion is defined to be the 'optimal' trade off.

- *DTWdist*

: the Dynamic Time Warping (DTW) distance between the reference trace and the aligned trace. The FastDTW algorithm is used internally for both elastic trace warping and determining the distance between the reference and the aligned trace, see Elastic Align.

- *DTWpathlen*

: the DTW warp path length. This value relates to the amount of stretching and compression necessary to align the traces.

- *Best tradeoff*

: the values for

window

and

radius

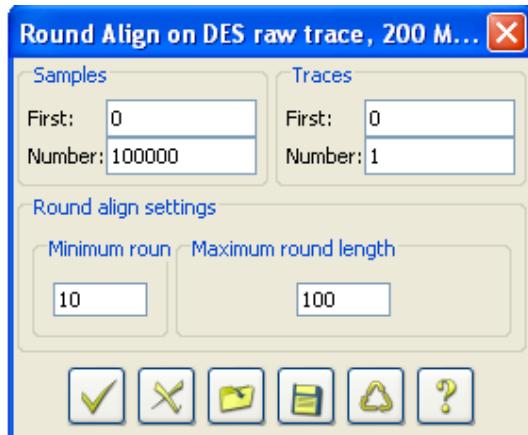
for which the criterion is minimal.

E.3.5 Round align

Purpose

To align multiple misaligned rounds in a trace set by inserting pauses at the right moments; rounds are detected by a pattern match.

Dialog



Inputs

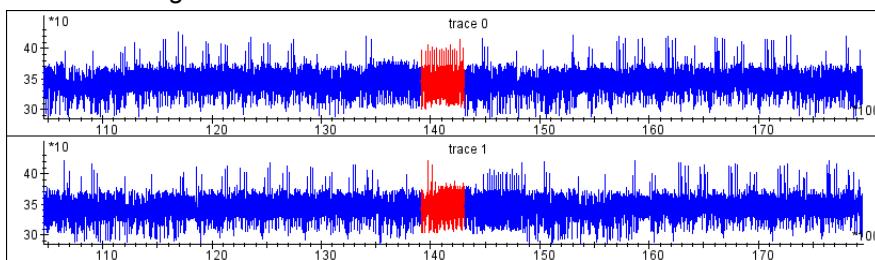
The user first selects a reference pattern that will be matched to the trace to find the start of each round. The "Minimum round length" and "Maximum round length" specify the minimum and maximum length of a round. Between these two bounds, the highest pattern match is used as the start of the round.

Result

The same round in each trace is placed at the same starting index by inserting a silence. If a round is detected at interval i , the next round is detected within the interval $[i + \text{minDist}, i + \text{maxDist}]$. The first round is defined to be within $[0, \text{maxDist} - \text{minDist}]$.

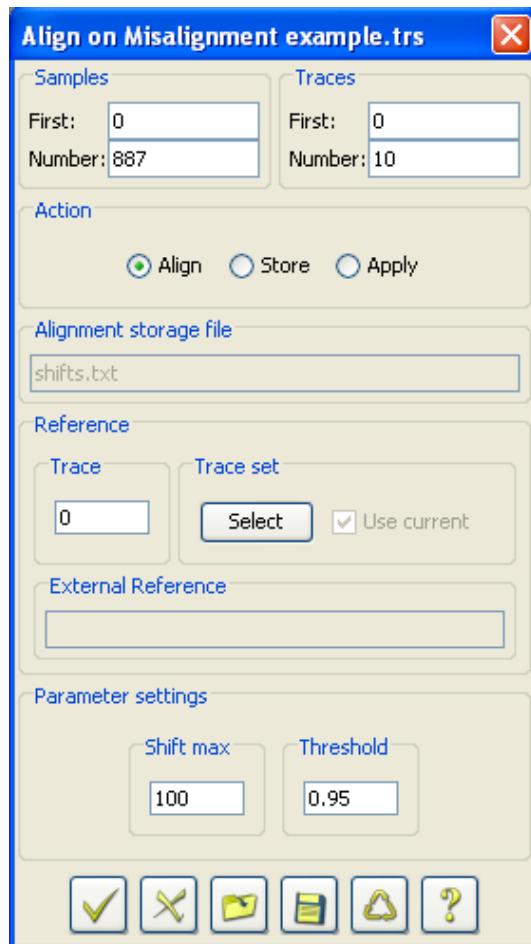
E.3.6 Static Align

Successful execution of this process is a prerequisite for statistical analysis. The module distinguishes between static and dynamic alignment. The former performs a single shift per trace to align it with a reference trace. The latter performs repeated shifts to achieve a continuous alignment with the reference trace.



The figure above shows two traces that need to be aligned. The selected part in the upper trace appears in the lower trace about 500 samples later.

Dialog



Input

The following parameters are relevant for static alignment:

- **Action:** The main action to be executed by the module.
 - *Align* aligns the trace set to the specified reference trace.
 - *Store* stores the alignment information of the trace set with respect to the specified reference trace to a file.
 - *Apply* uses the alignment information stored in a file to align the trace set.
- **Reference:** Reference trace selection options. Only applicable to *align* and *store* actions.
 - *Trace#* specifies the reference trace index in the reference trace set.
 - *Trace set* allows selecting between using a reference trace from the current trace set, or from an external trace set file.
 - *External reference* indicates which external trace set has been selected, or is empty if the current trace set is used.

- *Samples*: The part of the traces used for the static alignment, this must be a part of the entire trace to allow for left or right shifting.
- *Shift max*: The maximum number of samples to shift to the left or right.
- *Threshold*: The minimum correlation allowed for a match. Traces with correlation below the threshold are discarded.

Background

Static alignment uses a fragment (the selected trace part) to align traces with each other. Usually the analyst would take a part of the trace that is visually recognizable and is expected to be present and similar in all traces. During a static alignment the module correlates the selected part from the reference trace to any part in each of the traces to be aligned. The relative position where the best correlation appears is used as the shift value to align the traces.

With high values for the "Shift max" parameter the calculation of the correlation may take considerable time. There is an option in the Settings window called "Remove lag for reference correlation" which increases the speed but does not guarantee correctness. See Section 2.3, "Tweak settings".

Result

The Static Align process produces output detailing the results per trace.

```
Including trace 0, shift: 0, correlation: 1.0
Including trace 1, shift: -563, correlation: 0.990467
Including trace 2, shift: -462, correlation: 0.990083
Including trace 3, shift: -552, correlation: 0.987422
Including trace 4, shift: -204, correlation: 0.992715
Including trace 5, shift: -428, correlation: 0.994385
Excluding trace 6, correlation: 0.141
Including trace 7, shift: -867, correlation: 0.958981
Including trace 8, shift: -337, correlation: 0.994892
```

The output shows that trace 0 has a perfect correlation; this is because this trace was chosen as the reference.

Success is not always guaranteed if the traces are very noisy. In that case it is recommended to filter the traces prior to alignment. The filtering could be based upon moving average or spectral filtering. Even when signals are extremely noisy, a very low-pass filter will reveal a trend in the signal that can be used for alignment. However, the filtering process could lead to signal loss. Therefore the module allows the alignment shifts computed for a filtered signal to be stored by selecting the Store action. These shift values can be reused later with the Shift action to apply to the unfiltered signal. In this way the original signal can be aligned without losing spectral information.

Related tutorials

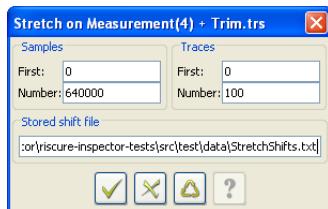
- **RSA High Order Analysis Tutorial** [\[../tutorials/sectionsRSAHighOrderAnalysisTutorial.html\]](#) [\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsRSAHighOrderAnalysisTutorial\]](#)

E.3.7 Stretch

Purpose

To scale traces which previously were statically aligned. This function enables unstable clocks or random process interrupts to be corrected.

Dialog



Input

The stretch dialog requires selection of a file with shift values for the end point of the stretch process. In the current trace set the user should select an area starting with the alignment point of the earlier static align at the left, and ending with the point where a static align at the right was applied (without saving traces) that resulted in the selected shift file.

Background

Static or dynamic alignment is suboptimal when a trace set contains traces whose misalignment steadily grows due to deviating clock speeds. In that case it would be beneficial to stretch or shrink traces slightly in order to achieve alignment. This would result in an aligned section much wider than just the section selected for a static alignment.

Traces misaligned due to different clock speeds that remain constant throughout the trace period can be aligned rather well using the Stretch module. The successfully aligned section will be much smaller if the clock speed varies considerably within a trace, or if the variations are caused by random process interrupts.

Before applying this module the analyst should prepare a trace set with the Align module. First a trace set should be created which is left-aligned at the beginning of the misalignment to be solved. Subsequently a second alignment must be performed at the end of the misalignment to be solved, and the alignment results should be stored. Then the Stretch module can be applied on the left-aligned traces, where the selection extends from the left-aligned point up to the point at the right that is to be aligned.

The result of the Stretch module will be a set of traces, where the selected area is stretched (or shrunk). The non-selected area at the left and right will just be copied.

E.4 Analysis

This section contains the descriptions for all the analysis modules.

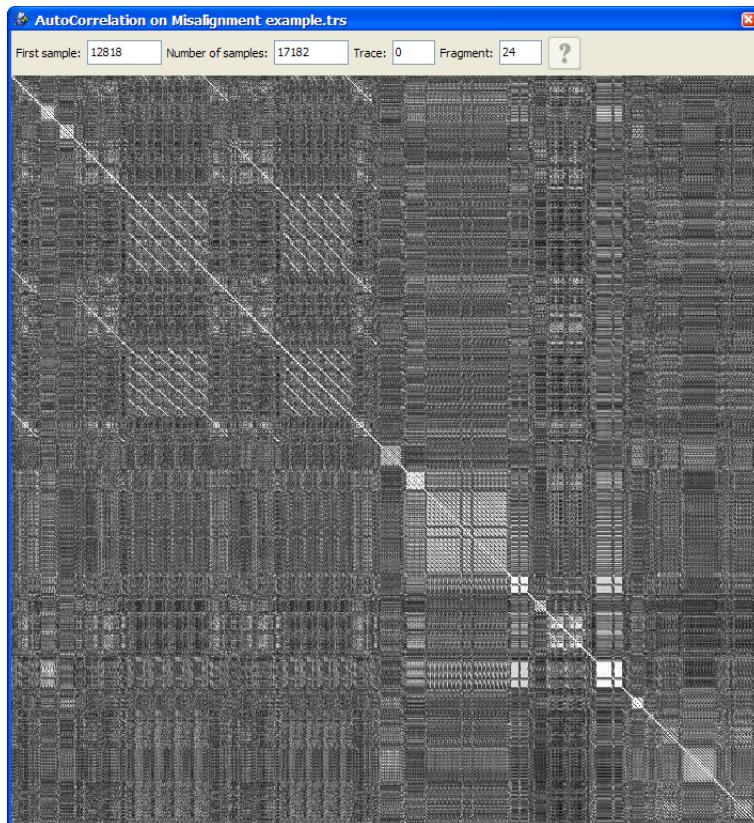
E.4.1 AutoCorrelation

Version: SCA

Purpose

To visualize repeating processes in a single trace by computing an auto-correlation graph. This analysis can be useful to recognize known program structures, such as permutations.

Dialog



Inputs

The First sample, Number of samples and Trace edit fields define the scope of the analysis. The Fragment edit field is computed automatically as the step size, but can be modified.

When the graph is generated, a copy function is available under the right mouse button. Also the location of fragments corresponding to a specific pixel can be shown in the title bar by clicking on the pixel.

Result

The module divides the selected trace part in N fragments of equal length, where N is defined as the width (in pixels) of the graphical area. Next a matrix is computed where every fragment is correlated to every other fragment. The first row in the matrix contains the correlation coefficients between the first fragment and all other fragments. The last row in the matrix

contains the correlation coefficients between the last fragment and all other fragments. Subsequently the matrix is visualized in the graph, by representing a good correlation by a light pixel, and a bad correlation by a dark pixel. Obviously, the diagonal from the upper left to lower right corner is always a white line, as this represents the correlation of fragments with themselves.

Background

An analyst who is aware of the implemented crypto algorithm may now be able to recognize specific algorithmic structures and gain knowledge about where the sensitive operations are executed. This knowledge can subsequently be used to mount a localized (differential) analysis using other modules. This graph may also help the analyst recognize some countermeasures against side channel analysis.

Related tutorials

- **Analysis of a DES implementation** [\[../tutorials/sectionsDESCardTutorial.html\]](#) [\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsDESCardTutorial\]](#)
- **Analysis of a DES implementation with random delays** [\[../tutorials/sectionsDESCardRndDelayTutorial.html\]](#) [\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsDESCardRndDelayTutorial\]](#)
- **Analysis of a DES implementation with S-box randomization** [\[../tutorials/sectionsDESCardRndSboxTutorial.html\]](#) [\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsDESCardRndSboxTutorial\]](#)
- **Analysis of an AES implementation** [\[../tutorials/sectionsAESCardTutorial.html\]](#) [\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsAESCardTutorial\]](#)
- **Analysis of an AES implementation with random delays** [\[../tutorials/sectionsAESCardRndDelaysTutorial.html\]](#) [\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsAESCardRndDelaysTutorial\]](#)
- **Analysis of an AES implementation with S-box randomization** [\[../tutorials/sectionsAesCardRndSboxTutorial.html\]](#) [\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsAesCardRndSboxTutorial\]](#)

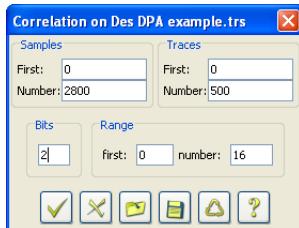
E.4.2 Correlation

Version: SCA

Purpose

To investigate an implementation's susceptibility to side channel analysis. It is possible to establish whether sensitive (crypto) data leaks from the implementation, and this may even allow reverse engineering of the implementation or the retrieval of specific secrets (e.g. keys).

Dialog



Input

The scope of the analysis is defined by setting the values in the Samples and Traces panels. In the Track panel the analyst can select the unit (bits or bytes) of the (crypto) input data to correlate with the samples. The Range panel defines the number of units to involve. One output trace will be generated for each unit.

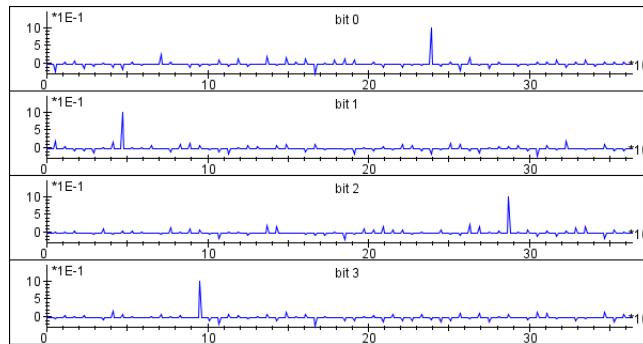
Background

Before correlation can be established, the input traces should be aligned, and they may also require other filtering to reduce noise. Various other modules can assist in this task.

Example

The file DESsimulation.trs contains 100 simulated power consumption traces, as well as the DES input (randomly chosen) and output data. Since the simulated traces do not contain noise, they clearly demonstrate the statistical properties. Note that real-life traces will contain (much) noise that will significantly hamper the analysis. An analyst will have to use many more traces and align them in order to achieve any practical results.

Running the Correlation module on DESsimulation.trs reveals clear correlation peaks at different times for the various input bits.



The peaks in this graph are caused by the reordering of the input bits during the IP permutation of the DES algorithm. Note that the temporal ordering of the bits may even facilitate a reconstruction of the permutation matrix, e.g. bit 1 is used before bit 3, etc.

Related tutorials

- **Second order AES Tutorial** [\[../tutorials/sectionsSecondOrderAEScard.html\]](#)[\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsSecondOrderAEScard\]](#)

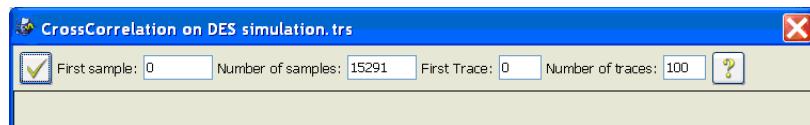
E.4.3 CrossCorrelation

Version: SCA

Purpose

To visualize correlation between samples in a trace set.

Dialog



Inputs

The *First sample*, *Number of samples*, *First Trace*, and *Number of traces* edit fields define the scope of the analysis.

When the graph is generated, a copy function is available under the right mouse button. In addition the location of samples corresponding to a specific pixel can be shown in the title bar by clicking on the pixel.

Result

The module considers the input trace set as a matrix where each column represents the same sample in different traces. It is important that the traces are well aligned, so samples in one column represent the same process. Subsequently, correlation coefficients are computed between all pairs of columns. These correlation coefficients are represented in a square matrix, where the value corresponds to the intensity of the representing dots. A positive correlation is shown in green, and a negative correlation is shown in red.

Background

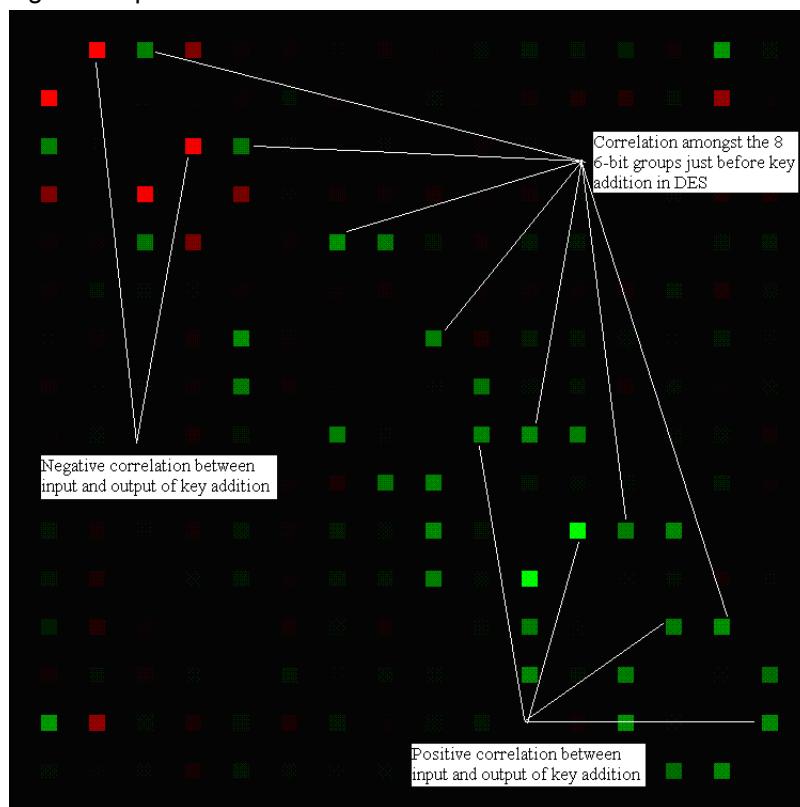
An analyst who is aware of the implemented crypto algorithm may now be able to recognize specific algorithmic structures and gain knowledge about where the sensitive operations are executed. This knowledge can subsequently be used to mount a localized (differential) analysis using other modules. Also, this graph may help the analyst recognize some countermeasures against side channel analysis.

This module also shows where values are inverted, since a negative correlation appears as a red dot. Thus it may be possible to recognize modulo-two key additions, which are common in symmetric algorithms. Since this module correlates multiple power consumption samples with each other, this type of analysis is referred to as high-order DPA.

Example

The picture below shows a graph taken from the key addition process in a DES simulation. The input is divided into 6 parts and added to the key. The key groups where input and output

have a negative correlation contain more than 3 bits set to 1, while the key groups that have a significant positive correlation have less than 3 bits set to 1.



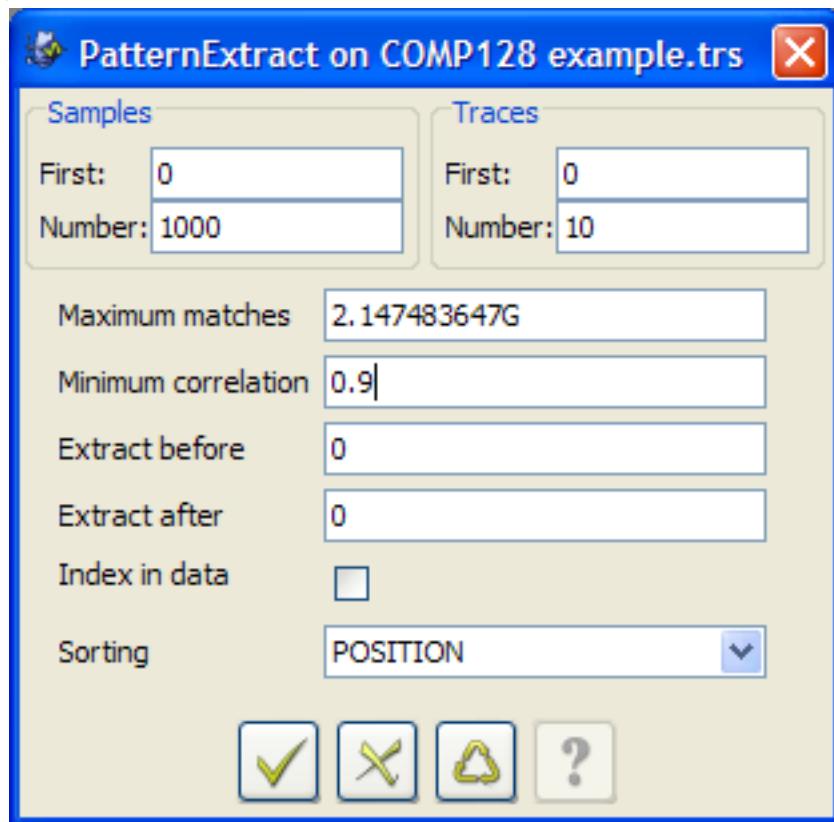
E.4.4 PatternExtract

Version: SCA

Purpose

PatternExtract is used to extract non-overlapping repeated patterns from traces. It is similar to the PatternMatch module used with the 'filter' option.

Dialog



Usage

- Select the pattern in a trace set that needs to be extracted.
- Run the module, choosing the desired options (mouse over for tooltip describing the individual options)

Input

After selecting a reference pattern manually before loading the module or entering the reference area in the *Samples* scope panel, an analyst can select several options:

- **Maximum matches:** This value limits the number of matches to extract from each trace. This is used when the analyst knows a maximum number of patterns is present or only a limited number of matches is wanted.
- **Minimum correlation:** The correlation threshold used to define matches. When correlation between a particular pattern and the reference pattern is higher than this value, a match has been found.
- **Extract before:** Number of samples to extract before the matched pattern. This same number of samples is skipped from the beginning of the input trace when performing the pattern match, assuming that at least so many samples should be present before any extracted pattern.

- **Extract after:** Similar to *Extract before*, a number of samples to extract after the matched pattern. Again, exactly this number of samples is skipped at the end of the trace for the pattern match.
- **Index in data:** Prepends the data of each extracted pattern trace with indexing information. This includes the original trace number (4 bytes) and the index within the matched patterns (2 bytes).
- **Sorting:** Allows selecting how extracted patterns are sorted. They can be sorted by *position* in the input trace or by *correlation* value. The former returns patterns ordered in time, while the latter returns patterns ordered by the quality of the match.

Background

This module computes the normalized cross-correlation of the selected pattern at each position in the input trace. The matches that are above the given threshold are sorted. In descending order of correlation values, pattern locations are returned if they do not overlap patterns returned earlier. Overlapping is determined by the length of the selected pattern, and the index of its match.

After this detection process, the matches patterns are returned as individual traces, possibly with an added header and tail length. These individual traces can be returned sorted according to correlation value, or order in the original trace. If desired, the original index in the trace can be added to the data field of the resulting traces.

Result

The result after the above steps is a trace set where each trace is an extracted pattern. The output window displays information for each match, e.g.:

```
Match 22 in input trace 0, position 581, length 20, correlation 0.43031707, class 0
Match 23 in input trace 0, position 609, length 20, correlation 0.61329377, class 0
Match 24 in input trace 0, position 637, length 20, correlation 0.36170387, class 0
Match 25 in input trace 1, position 5, length 20, correlation 0.29715848, class 0
Match 26 in input trace 1, position 26, length 20, correlation 0.48384708, class 0
Match 27 in input trace 1, position 53, length 20, correlation 0.36522982, class 0
```

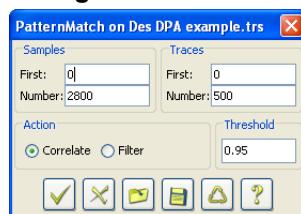
E.4.5 PatternMatch

Version: SCA

Purpose

To compare fragments and search for similarities.

Dialog

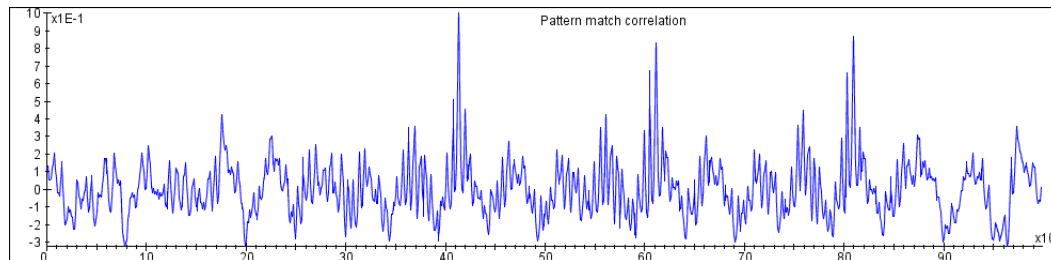


Input

Two methods are available:

Correlate

Compare the selected fragment with each trace at each offset. The result traces show the correlation.



Obviously, the selected fragment will have a perfect correlation (1.0) at its own offset, but other trace parts may also demonstrate a significant correlation. This method is very useful for finding repetitions or similarities in the software being examined.

Filter

With the filter method the input traces are tested for correlation with the selected fragment. Trace parts that exceed the given *threshold* are extracted and produced as result traces. The result traces will have the same length as the selected fragment, and will contain all trace parts similar to the selected fragment.

Related tutorials

- **Second order DES Tutorial** [[..../tutorials/sectionsSecondOrderDES_card.html](#)][PDF] [[..../tutorials/Tutorials.pdf#sectionsSecondOrderDES_card](#)]

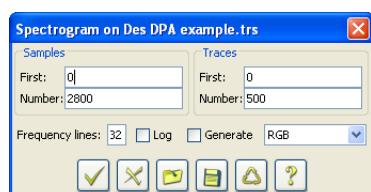
E.4.6 Spectrogram

Version: SCA + FI

Purpose

To present and analyse the evolution of the spectrum of a trace over the time.

Dialog



Input

The Spectrogram module takes the following inputs:

- The *frequency lines* text box specifies the number of frequency components to compute. If a number which is not power of two is supplied, the module will take the next power of two as the number of frequency lines.
- The *log* check box allows the analyst to select whether the spectral intensity will be calculated in a logarithmic scale or not.
- The *generate* check box tells the module to generate output traces. Each output trace represents the behavior for a certain frequency over time.
- A drop-down list is also available for choosing the kind of coloring map that will be used. Current options are black to green and RGB coloring schemes. In the black to green coloring scheme, black represents low intensity and green high intensity. In the RGB color map, low intensity is represented in blue and high intensity in red.

Results

The *Spectrogram* module presents a 2D plot showing the evolution of the signal's spectrum over the time. To this end, the module computes the fragment length as two times the number of frequency lines, and divides the trace into 50% overlapped fragments of the computed length. The last fragment is padded with zeros.

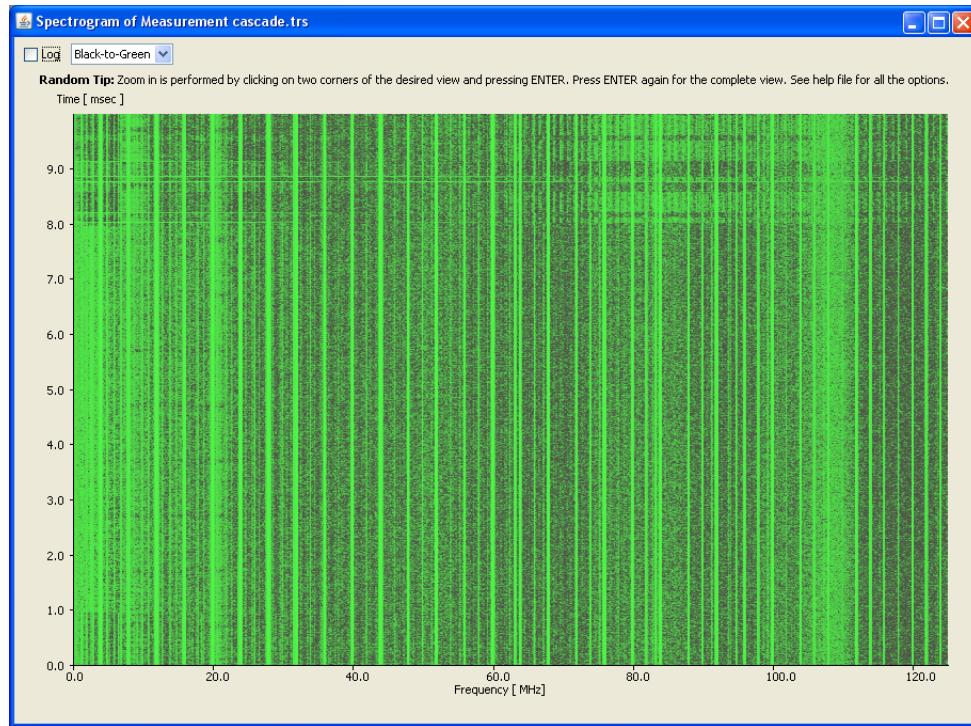
When several traces are selected for the analysis, the spectrum is calculated for all of them and an average is computed. This way it is possible to eliminate noise from the resulting signal.

In the 2D plot, the X axis represents the segment number while the Y axis represents the frequency value. In order to know the concrete values for a given point, it is possible to click on the desired pixel and the dialog's title will present the frequency, time and intensity values.

It is also possible to copy the 2D plot into the system clipboard by issuing a right click and pressing *Copy* in the pop up menu.

Furthermore, it is also possible to zoom into part of the image. To zoom in, click on a point of the spectrogram and it will be marked with a cross. Then click on some other point and a rectangle will show the selection. Pressing ENTER, the zoom will be performed. Further zooming is possible, and zooming out to the original view can be achieved by simply pressing ENTER when no area is selected.

Finally, the module shows the values for each point in the title bar when double-clicking over it. This action also removes any zoom selection.



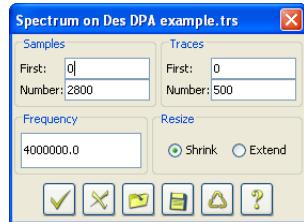
E.4.7 Spectrum

Version: SCA + FI

Purpose

To analyse the frequency spectrum of a trace. By observing the spectrum, the analyst can verify the clock frequency and its stability.

Dialog



Input

The *Samples* and *Traces* panels are used to set the scope of the frequency analysis.

The *Frequency* edit field is used to define the sampling speed of the input trace. By default this is the inverse of the x-scale of the input trace. If, for instance, the x-scale is 2 ns, the frequency is 500 MHz.

The *Resize* option relates to the fact that FFTs are computed for powers of two. Each signal to be analysed must therefore be either shrunk, or extended to a power of two. Shrinking has the

disadvantage of losing some information, but extension has the disadvantage of distorting the spectrum. Generally an analyst will prefer to use the nearest power of two.

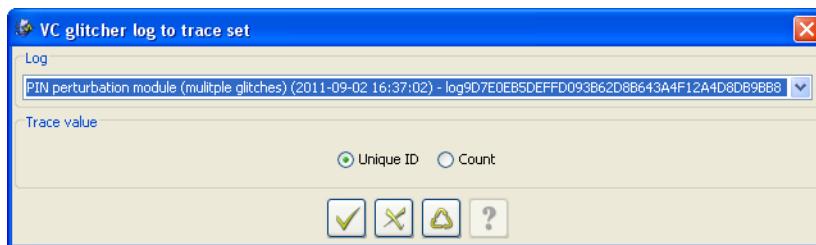
E.4.8 UniqueData

Version: SCA

Purpose

UniqueData is used to visualize different data responses of an optical FI log on the XY plane.

Dialog



Usage

- Perform XY scan using one of the perturbation modules.
- Start the module, select a log to analyze, and select the mode of operation.
- Use the AveragePlot module on the resulting traceset to visualize the XY scan.

Result

The result after the second step is a trace set, where each trace has one sample value. This sample is the result of the log analysis for one XY location. Depending on the mode of operation, each sample is either:

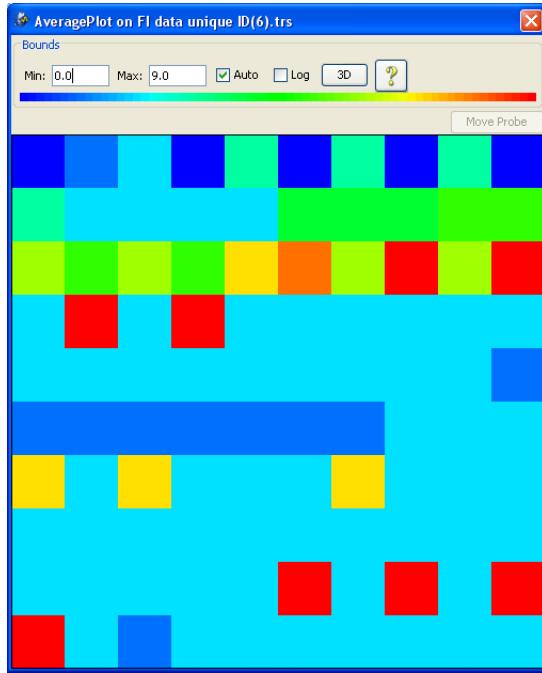
- The number of occurrences of the data in the whole perturbation run.
- A unique number, which is the same for each location where the data of the perturbation run is the same.

The output window displays all unique data strings, and, depending on the mode, the unique ID or the count:

```
FI data unique ID:
3be700ff8131fe45526973637572655a00005a01a000000bf000029000920040
0aa019000040 00000000f7: 1.0
3be700ff8131fe45526973637572655a00005a01a000000bf000029000920040aa019000040
00000000f700400269002b00005a01b00000be0000300900093: 5.0
3be700ff8131fe45526973637572655a00005a01a000000bf000029000920040aa019000040
00000000f700400269022900005a01b00000be0000300900093: 2.0
3be700ff8131fe45526973637572655a00005a01a000000bf000029000920040aa019000040
```

```
000000000f7004002690328000005a01b000000be00000300900093: 6.0
3be700ff8131fe45526973637572655a000005a01a000000bf00000290009200400aa0190000040
000000000f70040026985ae000304700018000034a9a1baa932ad2fe: 3.0
3be700ff8131fe45526973637572655a000005a01a000000bf00000290009200400aa0190000040
000000000f70040026986ad000005a01b000000be00000300900093: 7.0
3be700ff8131fe45526973637572655a000005a01a000000bf00000290009200400aa0190000040
000000000f7004002698fa4000005a01b000000be00000300900093: 4.0
3be700ff8131fe45526973637572655a000005a01a000000bf00000290009200400aa0190000040
000000000f700400ca019000040000000000900061000005a01b000000be00000300900093:
0.0
3be700ff8131fe45526973637572655a000005a01a000000bf00000290009200400aa0190000040
000000000f700405ca019000040000000000000000000000000000000000000000000000000000000
000000000a100400a0208404434010c26440062041806008c208a089228024009a108600820050
8e2803029084b001042242b2b40320104940a226900b800005a01b000000be00000300900093:
8.0
3be700ff8131fe45526973637572655a000005a01a000000bf00000290009200400aa0190000040
000000000f73be700ff8131fe45526973637572655a: 9.0 FI data frequency count:
3be700ff8131fe45526973637572655a000005a01a000000bf00000290009200400aa0190000040
000000000f7: 10.0
3be700ff8131fe45526973637572655a000005a01a000000bf00000290009200400aa0190000040
000000000f700400269002b000005a01b000000be00000300900093: 4.0
3be700ff8131fe45526973637572655a000005a01a000000bf00000290009200400aa0190000040
000000000f7004002690229000005a01b000000be00000300900093: 58.0
3be700ff8131fe45526973637572655a000005a01a000000bf00000290009200400aa0190000040
000000000f7004002690328000005a01b000000be00000300900093: 4.0
3be700ff8131fe45526973637572655a000005a01a000000bf00000290009200400aa0190000040
000000000f70040026985ae000304700018000034a9a1baa932ad2fe: 4.0
3be700ff8131fe45526973637572655a000005a01a000000bf00000290009200400aa0190000040
000000000f70040026986ad000005a01b000000be00000300900093: 4.0
3be700ff8131fe45526973637572655a000005a01a000000bf00000290009200400aa0190000040
000000000f7004002698fa4000005a01b000000be00000300900093: 3.0
3be700ff8131fe45526973637572655a000005a01a000000bf00000290009200400aa0190000040
000000000f700400ca019000040000000000900061000005a01b000000be00000300900093:
5.0
3be700ff8131fe45526973637572655a000005a01a000000bf00000290009200400aa0190000040
000000000f700405ca01900004000000000000000000000000000000000000000000000000000000
000000000a100400a0208404434010c26440062041806008c208a089228024009a108600820050
8e2803029084b001042242b2b40320104940a226900b800005a01b000000be00000300900093:
1.0
3be700ff8131fe45526973637572655a000005a01a000000bf00000290009200400aa0190000040
000000000f73be700ff8131fe45526973637572655a: 8.0
```

The 'trace set' that results from this module can be visualized using the AveragePlot module. This is useful when analyzing XY scans for results that are obtained in a certain area of the chip. An example of a 10x10 visualisation is the following:



E.5 Compress

This section contains the descriptions for several resample modules.

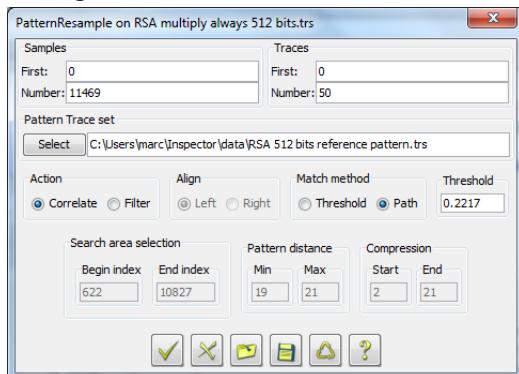
E.5.1 PatternResample

Version: SCA

Purpose

To find pattern matches and compress them. The module has two phases: 'Correlate', for detecting pattern occurrences; and 'Filter', for extracting the patterns.

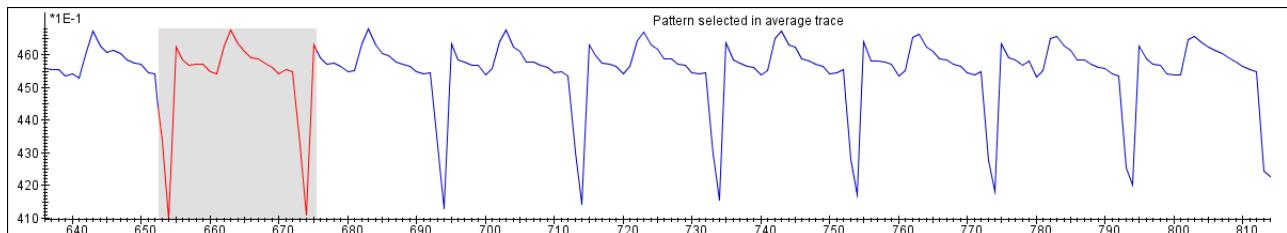
Dialog



Input

The *Samples* and *Traces* panels are used to set the scope of the analysis. The sample scope would normally not be restricted, or at least contain all detectable patterns. The trace scope would include a limited number of traces (e.g. 50) for the 'Correlate' phase, and all available traces (e.g. more than 1000) for the 'Filter' phase.

An essential input to the module is a *Pattern*. This needs to be prepared before the PatternResample module can be run. The pattern is created by aligning a trace set at the first occurrence of the patterns, and then computing an average trace. If the average trace shows a recognizable pattern (or multiple), then this part may be selected and saved. In the PatternResample module this pattern is selected by pushing the 'Select' button in the module dialog.



The *Action* panel is used to set the phase of the analysis. The 'Correlate' phase is for detecting pattern occurrences, while the 'Filter' phase is meant for extracting and compressing the patterns.

The *Align* panel is used to set the order of mapping found patterns. Selecting 'Left' will map patterns left-to-right. Selecting 'Right' will map patterns right-to-left. Ideally this should not make a difference, but when occasionally patterns are missed this may corrupt the mapping of subsequent patterns. To enable correct further processing it may be beneficial to perform mappings from left-to-right, and right-to-left and compare the results after subsequent cryptographic analysis.

The *Method* panel is used to set the method of identifying patterns. Selecting 'Threshold' will simply find all non-overlapping patterns for which the correlation exceeds the threshold. Selecting 'Path' will search for a chain of patterns that fit in a range of minimal to maximal distances. The first method is relatively fast, and works well when patterns are easily recognized by good correlation values. The second method works also on noisy correlation traces, but need predictable pattern distances. The method is effective when pattern distances are always less than twice the pattern length.

The *Threshold* panel is used to set a minimal correlation value for which patterns are detected. This threshold is only needed for the 'Threshold' method.

The *Search Area Selection* panel is used to restrict the area where patterns are searched. The same effect could be reached by selecting only part of the trace set when invoking the module. The panel also serves as a feedback to the user, as the result of the 'Correlate' phase when the module figures out the lower and upper bound of the area containing the patterns. An area selection defined by this panel is overruled by an active user selection of the trace set.

The *Pattern Distance* panel is used to restrict the minimal and maximal pattern distance. While patterns could be at any distance they may also be at very regular distances, and the pattern detection algorithm may benefit from that. During the 'Correlate' phase the module tries to figure out minimal and maximal distances and proposes them in this panel. These could be corrected by the analyst when proceeding to the 'Filter' phase.

The *Compression* panel is used to restrict the compressed sample range. If the entire pattern is to be compressed the bounds this panel should be set to 0, and the length of the search pattern. However, the search pattern may contain distinctive start and end features beneficial for recognition, but not to be included in the compression. During the 'Correlate' phase the module tries to figure out optimal compression bounds and proposes them in this panel. These could be corrected by the analyst when proceeding to the 'Filter' phase.

Application

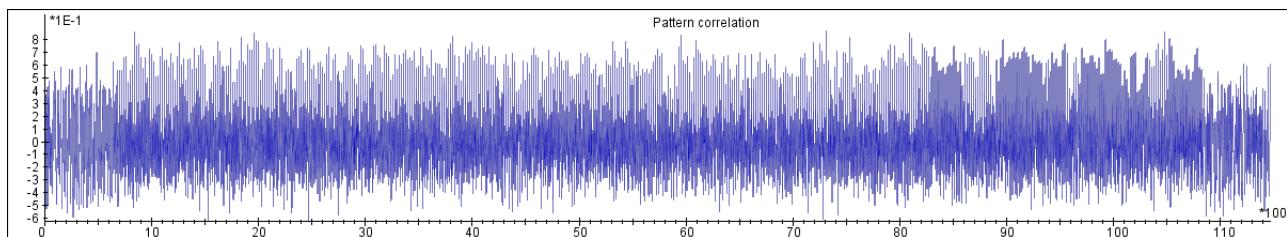
The module offers two phases of application: 'Correlate' and 'Filter'

The *Correlate* function should be used first to analyse the way patterns can be detected. This function should typically be invoked with enough traces (e.g. 50 should be good) to analyse the signal behaviour, but would not need a large trace volume. The module has built-in intelligence to suggest several parameters needed for the subsequent 'Filter' phase.

With the *Filter* method the input traces are one-by-one converted to compressed traces. Each input trace is tested for correlation with the selected fragment. Trace parts that match the given pattern are extracted and produced as result samples. The first filtered trace will set the size of the result traces. If this first trace contains N matches, each result trace will have N samples. The module reports for each input trace the number of patterns extracted. If these are more or less than N they will be mapped 'left-to-right', or 'right-to-left' (depending on the align selection) onto the output trace. The quality of the module output depends on the number of traces with the exact same amount of matched patterns.

Example

Load Trace set 'RSA multiply always 512 bits.trs'. Invoke the module and select pattern trace set 'RSA 512 bits reference pattern.trs'. Select phase 'Correlate' and method 'Path' and run the module on 50 traces. The module will produce the correlation traces shown below.



The correlation trace shows how patterns are detected at a regular distance, but that false correlation peaks can be observed at trace boundaries. The analysis further produces some useful settings for the subsequent filter phase, that can be used directly.

```
Pattern length: 23 (defined by loaded pattern)
Average distance between pattern matches: 19.980488, suggested distance between
pattern occurrences, minimal: 19, maximal: 21
Suggested threshold: 0.22170354, number of matches (exceeding this threshold): 511,
first match starts at: 632, final match starts at: 10818
```

Invoke the module again, select the 'Filter' phase and reset the trace scope to the entire volume of the input trace set. The module produces a nice set of compressed traces with 511 samples, which can be further processed with the RsaNeighborCorrelation module.

```

Pattern length: 23 (defined by loaded pattern)
Trace 0, found 511 patterns
Trace 1, found 511 patterns
Trace 2, found 511 patterns
...

```

E.5.2 RFResample

Version: SCA

Purpose

This module can be used to resample a Radio Frequency Analysis signal after passing through the sample and hold demodulator. The sample and hold demodulator takes two samples for each period of the RF carrier wave. Hence at a frequency of 27.12 MHz, equal to twice the carrier wave. *The RFResample* module synchronizes to the RFA demodulated signal and resamples the signal to a 27.12 MHz.

Dialog



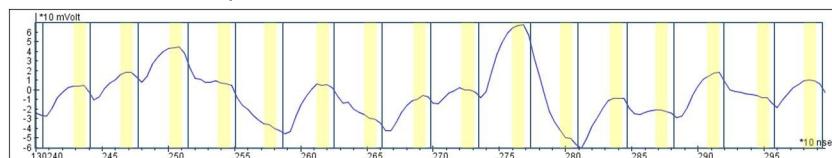
Input

The user may select part of a trace by choosing the corresponding samples or may select part of the trace set by selecting the corresponding traces. No other parameter settings are required.

Background

RFResample should be used in combination with the sample and hold demodulation device for Radio Frequency Analysis. Please refer to the section on the sample and hold demodulator for RFA for details about the demodulation process.

RFResample synchronizes based on the time derivative of the input signal (see vertical black lines in trace below) and cuts the input signal into periods of 1/27.12 MHz. The resample value is the average of the -40% to -15% interval (see yellow areas in trace below) calculated relative to the end of each period.



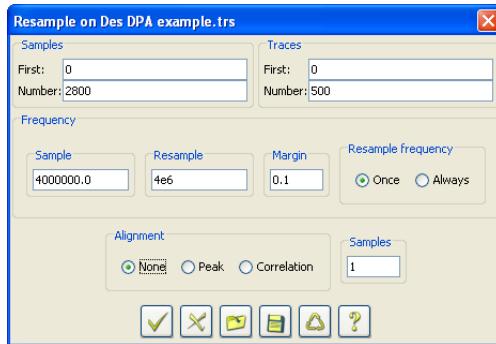
E.5.3 Resample

Version: SCA

Purpose

To resample a trace set with a lower frequency. A careful application of this function enables the analyst to strongly reduce the trace size without loosing too much information.

Dialog



Input

The frequency panel requires the following information:

- *Sample*: The original sample frequency used in the input trace set.
- *Resample*: The (estimated) resample frequency.
- *Margin*: The relative window to be used for finding the optimum resample frequency. A value of 0.1 means that the computed resample frequency must be in the range of 0.9 to 1.1 times the estimated resample frequency. A value of 0 means that the resample frequency is fixed.

The *Resample frequency* panel provides the following options:

- *Once*: The resample frequency is computed only for the first trace.
- *Always*: The resample frequency is computed for each trace.

The *Alignment* panel offers an intra-resample-period alignment: to perform small shifts (at maximum the length of the resample period) in order to align traces before resampling. The following methods are supported:

- *None*: no alignment is performed
- *Peak*: highest values within a central resampling period are aligned
- *Correlation*: central resampling periods are aligned by finding the best correlation value

The *Samples* panel offers setting the number of samples to retain for each resampling period. By default this value is 1, but it could be beneficial to set this value to 2 or 4 when different statistical effects are expected within one clock cycle.

Background

Resampling is often attempted on a dominant frequency, for example the external clock. As this frequency is often not a whole divisor of the sampling frequency, the module allows the resampling frequency to be estimated. The module uses the estimated resampling frequency while analysing the spectrum. The strongest frequency component within a given margin is selected as the resampling frequency.

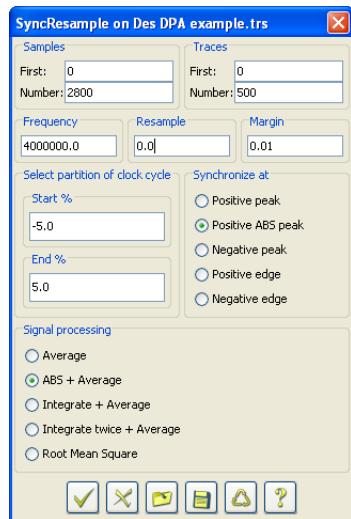
E.5.4 Sync Resample

Version: SCA

Purpose

To resample a trace to a lower frequency while each sample period is dynamically tuned to the wave shape.

Dialog



Input

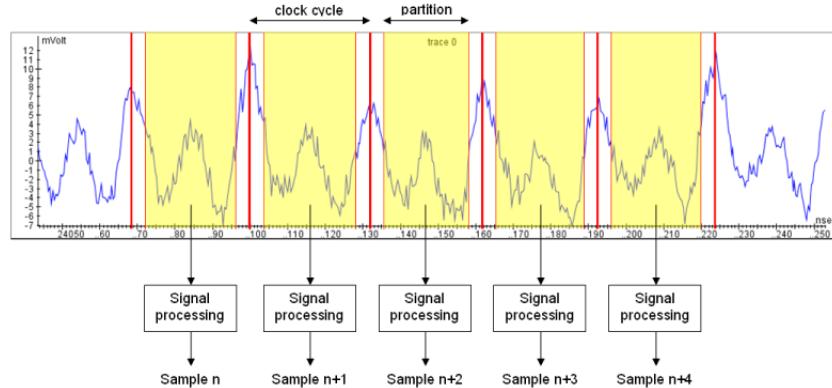
This dialog requires the following information:

- *Sample frequency*: the original sample frequency used in the input trace set
- *Resample frequency*: the (estimated) resample or clock frequency
- *Margin*: the relative window allowed for finding the optimal resample frequency. A value of 0.1 means that the computed resample frequency must be in the range 0.9 - 1.1 times the estimated resample frequency.
- *Start %*: start processing at the given percentage of each clock cycle. This percentage must be in the range -50 - 50. Note that end % - start % must be at least 10 %
- *End %*: stop processing at the given percentage of each clock cycle. This percentage must be in the range 50-150. Note that end % - start % must be at least 10 %

- *Synchronize at:* indicates if the clock cycles start at a Positive peak, Positive ABS peak (i.e. positive peak in ABS of signal), Negative peak, Positive edge or Negative edge
- *Signal processing:* the signal processing algorithm that is used before averaging. See next section for more details.

Processing techniques

The following figure shows how an input trace is processed. Note that in this example, the resampling process is synchronized at the positive peaks.



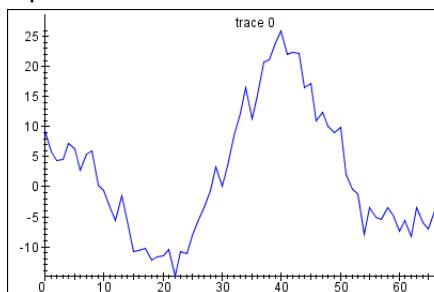
Average

When Average is selected, the samples within the selected partition of each clock cycle are averaged. Besides averaging, the data is not processed.

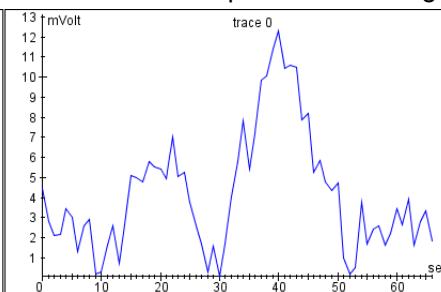
ABS + Average

When ABS + Average is selected, the samples below zero within the selected partition will be set to their positive equivalent. After that, the samples are averaged. The following figures show the input trace and the result before averaging respectively.

Input:



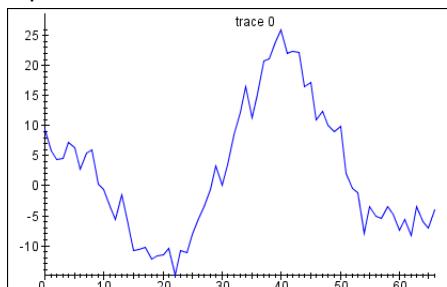
Output before averaging:



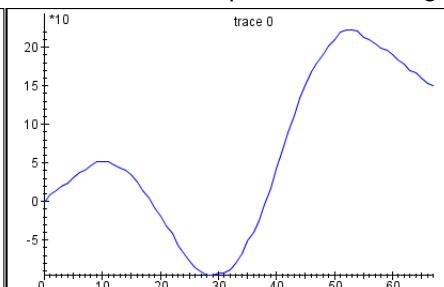
Integrate + Average

When ABS + Average is selected, the samples within the selected partition will be integrated. After that, the samples are averaged. The following figures show the input trace and the result before averaging respectively.

Input:



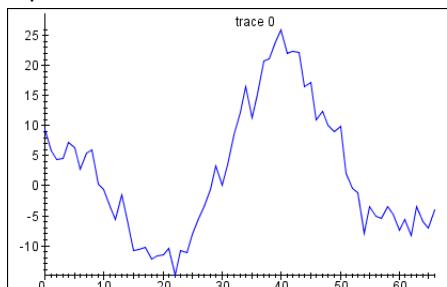
Output before averaging:



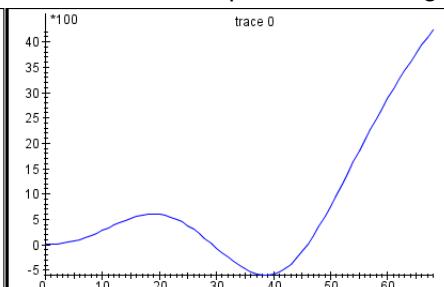
Integrate twice + Average

When ABS + Average is selected, the samples within the selected partition will be integrated twice. After that, the samples are averaged. The following figures show the input trace and the result before averaging respectively.

Input:



Output before averaging:



Root Mean Square

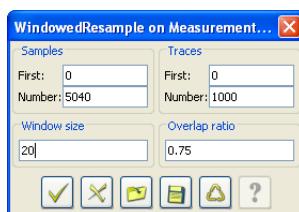
When Root Mean Square is selected, the square of the samples within the selected partition is computed. After that, the samples are averaged. Finally, the average value is square rooted.

E.5.5 WindowedResample

Purpose

To do a window based compression. Rather than converting from one frequency to another, this module simply compresses N samples into 1 (where N is the window size). Additionally, the compression may use overlapping windows.

Dialog



Input

The WindowedResample dialog requires two inputs:

1. The window size (compression factor). An integer number setting the number of samples compressed into one.
2. The overlap ratio. A real value in the range 0 to 1 setting the relative window shift, where 0 means no overlap, and 0.99 means 99% overlap.

Background

The WindowedResample module may bring two advantages:

1. A Signal-to-noise-ratio (S/N) improvement. By averaging several samples that carry the same signal the noise is reduced.
2. A performance improvement. The compression results in smaller traces which can be processed faster

If the analyst knows the boundaries of sample sequences that carry the sample signal (for instance a clock cycle), he may not need any overlap. If not, some overlap may prevent loss of S/N, at the cost of some performance.

E.6 Crypto

This section contains the descriptions for the main crypto modules.

E.6.1 AES Advanced Analysis

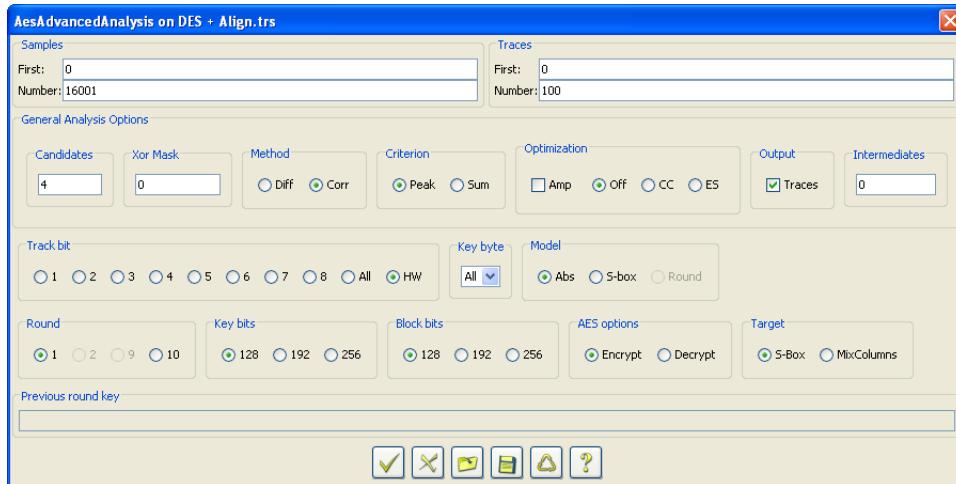
Version: SCA

Purpose

To retrieve an AES key using sophisticated statistical analysis methods.

The *AesAdvancedAnalysis* module provides a basis for sophisticated differential side channel analysis experiments on AES. The module extends *AdvancedDifferentialAnalysis* which implements the basic features that can also be used by other algorithm specific modules.

Dialog



Input

This help file explains advanced analysis settings. For a basic understanding of differential analysis read the AesAnalysis helpful.

General Analysis Options:

- **Candidates:** The number of candidates displayed in the list of best key candidates for each S-box.
- **Xor Mask:** A hex mask applied to the target before correlation. This allows for inversion of some target bits. For example hex mask A inverts bit 1 (msb) and 3 of an S-box output. This can be useful when seeking correlation with the hamming weight of all target bits while the bits have different polarity.
- **Method:** The computation method of result traces.
 - **Diff:** means a classical differential trace where all traces corresponding to a data bit set to 1 are collected in one group while all other traces are collected in the second group. The result is given by subtraction of the average trace of both groups. This method is relatively fast, but also sensitive to noise.
 - **Corr:** means a correlation trace where result traces are computed as a sequence of correlation coefficients of sample columns with data columns. This method is slightly more resource consuming, but produces better results in the presence of noise.
- **Criterion:** The way of selecting a good correlation.
 - **Peak:** Just search for the highest peak in the result trace.
 - **Sum:** Add all values in the selection that exceed a noise threshold given by $2/\sqrt{n}$, where n is the number of traces.
- **Optimization:**

- **Amp**: means an amplified correlation trace where the mathematical computation of correlation is amplified by a modified standard deviation. See DES Advanced Analysis for more detail.
- **Off**: No CC/ES applied.
- **CC**: cross-correlation. Here we take into account that incorrect candidates may also produce significant peaks, and incorporate the strength of various ghost peaks. See DES Advanced Analysis for more detail.
- **ES**: like CC, but uses Euclidean Similarity as a measure instead of correlation. See DES Advanced Analysis for more detail.
- **Output**: Produce differential/correlation result traces.
- **Intermediates**: Number of intermediate results to produce before the final output is to be presented
- **Track bit**: The S-box output bit to correlate the samples with. We can select each bit individually, or all together or take the hamming weight (HW). With all we actually compute 8 differential traces per candidate, and then sum the absolute values. With HW we compute one differential trace per candidate where we the power leakage corresponds to the hamming weight of the data.
- **Key Byte**: The key byte to retrieve, or "all" to specify to retrieve all of the key bytes.
- **Model**: A smart card may leak information related to the absolute value of intermediate data, or relative to some data earlier present in the process
 - **Abs**: Leakage is conform the absolute value of the target intermediate data
 - **S-box**: Leakage is relative to the S-box input value
 - **Round**: Leakage is relative to the round input value. This means the hamming distance between the output of the SubBytes and the input of the AddRoundKey functions is used. Note that attacking the first round with this method is not possible, since an additional AddRoundKey operation is performed.
- **Round**: The round to apply the analysis to. Initially the attack must be applied to the first or last round. With 128 bits keys, only one round key is sufficient to retrieve the key. With longer keys, also the penultimate/second round needs to be attacked.
- **Key bits**: The length of the key.
- **Block bits**: The length of input/output data.
- **AES options**: Specifies whether the target operation is an encryption or a decryption.
- **Target**: The intermediate data who's output is used for predicting key bits
 - S-box: use the output bits of the S-boxes directly
 - MixColumns: use the output of the MixColumns operation. Since a single key byte is used four times, four traces per candidate will be generated. Since the MixColumns operation is not performed in the final round, the last round of an encryption or the first round of a decryption cannot be attacked with this option.

- **Previous round key:** The previously found round key, in case multiple rounds need to be attacked.

E.6.2 AES DFA

Version: FI

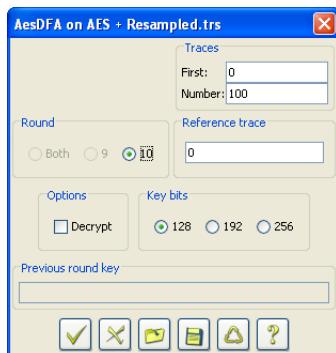
Purpose

Recover an AES key by analyzing fault-injected outputs.

Description

This module performs a Differential Fault Analysis on a trace set. Only data associated with the traces is processed, thus samples are ignored. The module expects the data to hold the 16 input bytes and the 16 output bytes. In case only 16 bytes are present, the module assumes only the output is stored with the traces. For the attack to work, at least one trace must contain the original, not fault-injected AES output, while the other traces must contain the same input with an injected fault.

Analysis Options



- **Round:** The round to attack. If faults in the current trace set are injected in a single round, select the targeted round. In case faults are spread in both the rounds (or even in the full 10-14 rounds), select the *both* option.
- **Reference Trace:** The trace that contains the original, correct output.
- **Key Bits:** The size of the key.
- **Decrypt:** If selected, attacks a AES decrypt operation.
- **Previous round key:** The round key found for the first round and used for the second round. Only useful when attacking one round at a time.

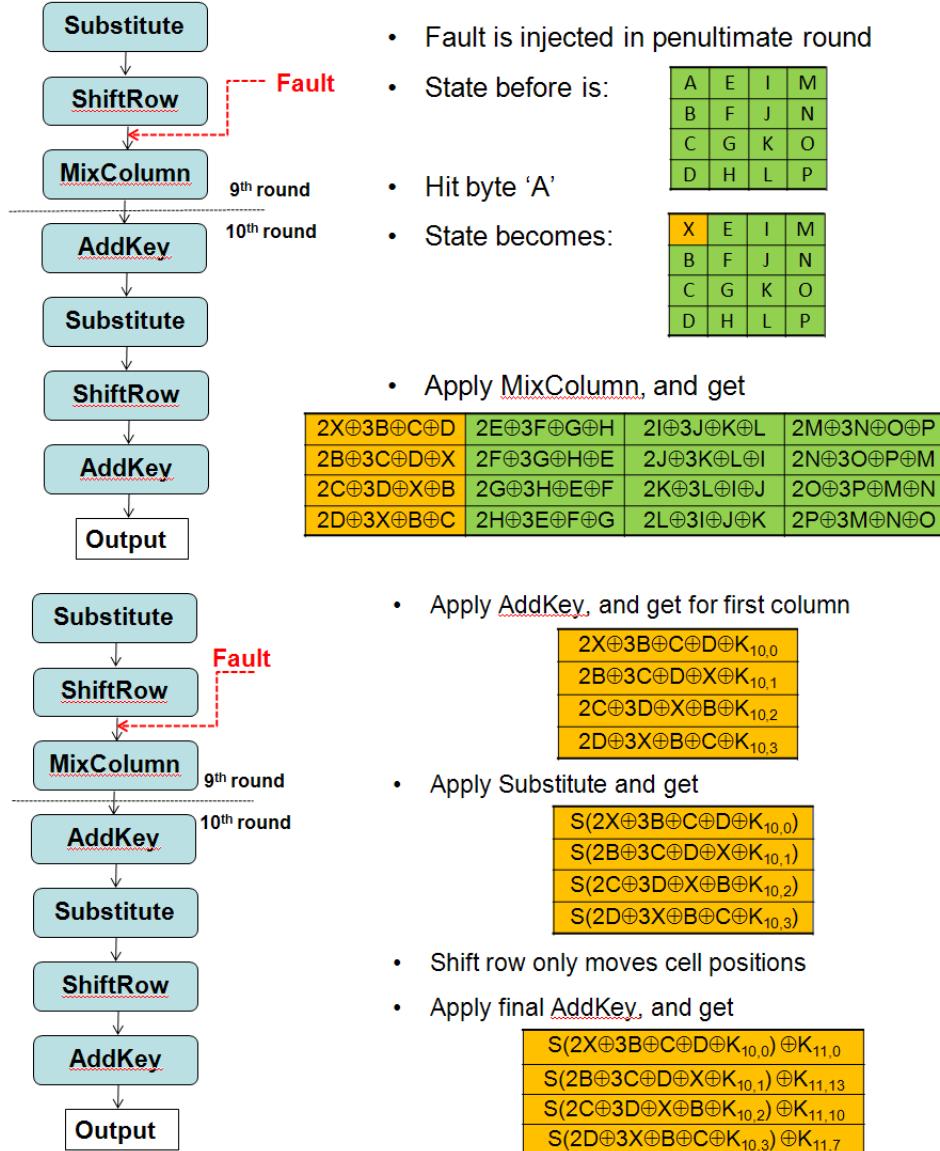
Results

The module will output the results for the most likely S-Box candidates. The results are normalized such that the best candidate always has the value "1". Furthermore, the module

generates a trace set which contains the same information in a trace form. The highest peak will be the most likely candidate for an S-Box.

Background

The idea behind the attack is to corrupt one entry in the "state" just before penultimate-round MixColumn. The following two pictures show the attack result



Next, we compare the correct result with the faulty one:

$$S(2A \oplus 3B \oplus C \oplus D \oplus K_{10,0}) \oplus K_{11,0} = O_0$$

$$S(2X \oplus 3B \oplus C \oplus D \oplus K_{10,0}) \oplus K_{11,0} = O'_0$$

Xor both formulas:

$$S(2A \oplus 3B \oplus C \oplus D \oplus K_{10,0}) \oplus S(2X \oplus 3B \oplus C \oplus D \oplus K_{10,0}) = O_0 \oplus O'_0$$

Which can be simplified as:

$$S(Y_0) \oplus S(2Z \oplus Y_0) = U_0$$

where $Z = A \oplus X$, $Y_0 = 2A \oplus 3B \oplus C \oplus D \oplus K_{10,0}$ and $U_0 = O_0 \oplus O'_0$

Only a subset of values for Z matches with Y_0 . The intersection of the joined formulas for 4 cells yields a sub set for Z

When Z is restricted, matching ranges of Y_0, Y_1, Y_2, Y_3

$$S(Y_0) \oplus K_{11,0} = O_0 \text{ provides a range for } K_{11,0}$$

Similarly, ranges for $K_{11,13}, K_{11,10}$ and $K_{11,7}$ are found. Four subsets of round key bytes are found with one byte. The process is iterated until all the sub sets only have one element, and all the round key bytes are found

Note



If the number of bytes in the communication data of a trace is odd then the last byte is interpreted as a timeout flag. If this timeout byte has value 1 the data of this trace is ignored.

E.6.3 AES Known Key Analysis

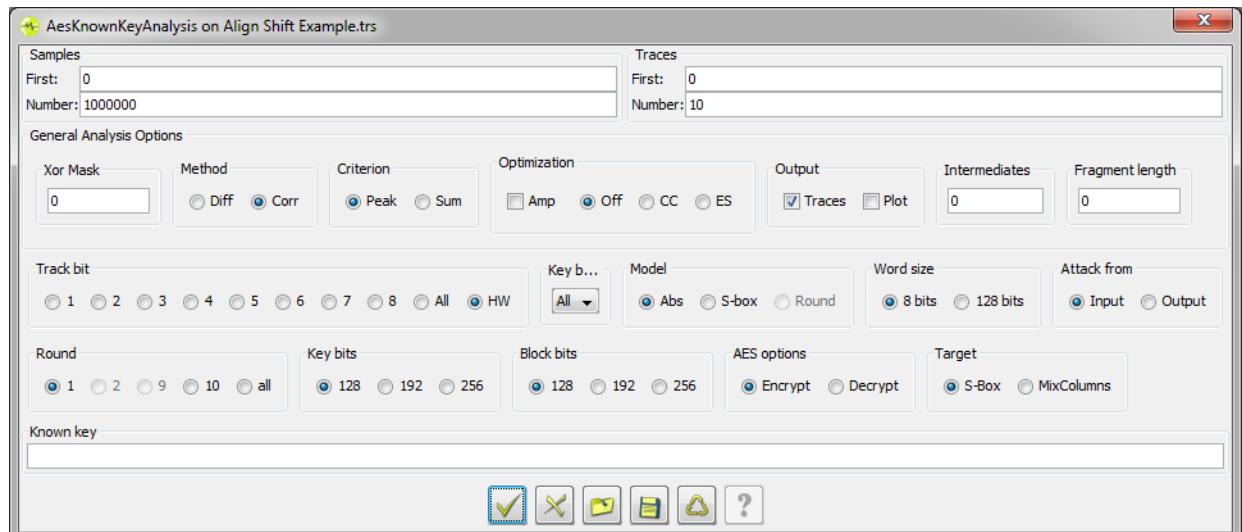
Version: SCA

Purpose

To determine data leakage of intermediate AES results using a known key.

The *AesKnownKeyAnalysis* module provides functionality to determine the location of data leakage related to intermediate results of the AES rounds inside a trace. The module extends the *AesAdvancedAnalysis* module, thus offering the same side channel distinguishers.

Dialog



Input

This help file explains known key analysis results. The module settings inherited from AesAdvancedAnalysis can be found in its own help file.

Known Key Analysis Specific Options:

- **Round:** The round to be analysed. It can be one round in the set 1,2,9-14 (depending on the key length) or the complete AES. The module will compute the results of the selected rounds and obtain its leakage. Note that when all the rounds are attacked, one more round will be displayed in the results (so for AES-128, 11 rounds will be displayed). This is because AES use one more round key. When absolute model is used, the first or the last round output (depending on whether the target AES is in encrypt or decrypt mode) will correlate with the input or output data.
- **Attack from:** This checkbox is used to set whether to perform the attack starting from the input or from the output. Since the key is known, both attacks are always possible. Attacking a round from the input or from the output will have different final results, because different intermediates are used. Note that in order to obtain the same results as in the AesKnownKeyAnalysis module, attacks should be performed using the output for the last rounds and the input for the first rounds.
- **Fragment length:** The module will divide the search interval in several fragments, providing its output per fragment. In this way, it's possible to determine in which fragment a certain round is located.
- **Plot:** Checking this option will instruct the module to generate a 2D representation of the results in a new dialog window.
- **Word size:** When set to 8 bits, the module analyses only the bits related to a given S-box. When set to 128 bits, the module looks at 128 bits hamming weights, setting the bits not related to the analysed S-box to their correct value. This is useful to maximize the correlation when all the target results are being computed at the same time.

Result

The module produces two kinds of results: textual and graphical. For a detailed description of the results, see the DesKnownKeyAnalysis help file.

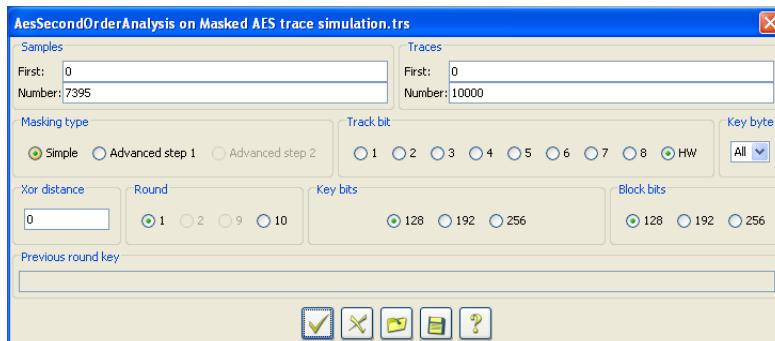
E.6.4 AES Second Order Analysis

Version: SCA

Purpose

To investigate a masked AES (Advanced Encryption Standard) implementation's susceptibility to second order side channel analysis.

Dialog



Inputs

An analyst can mount an attack on AES by first collecting a large set of traces including samples and crypto data (the data which is input or output for the algorithm). It is important that the crypto data be variable in order to establish sufficient variation in the intermediate data to observe correlation. The trace set may need preprocessing to reduce noise and establish alignment. Also the Correlation module could be applied to verify data leakage.

The analyst selects a part of the trace where intermediate data may be observed, during the substitution step (see 'background' below). The following fields have been added, on top of the others already present in the standard AES Analysis

- Xor distance: Distance between the values to XOR together in order to remove the masking. Can be either a single value if the correct distance is known or a range (e.g. '3-150')
- Masking type: This module supports side channel analysis of two different kind of masking, see 'background' for more informations

Background

The AES algorithm was developed just before the year 2000, and is becoming the world's leading symmetric security algorithm, replacing the popular variations of the DES algorithm. The AES algorithm is used, for example, in the authentication scheme for 3G mobile networks (Milenage).

The AES algorithm can take keys and block sizes of either 128, 192 or 256 bits. The number of rounds is either 10, 12 or 14, depending on the key and block sizes. As a first step a key schedule is created where the full key is used to derive N+1 round keys for N rounds.

All rounds (except for the last one) have the following structure:

- **key addition**

The round key is added modulo-two to the data.

- **substitute**

Each data byte is replaced by the value given by an 8-bit substitution table.

- **shift row**

The order of bytes in the data array is changed.

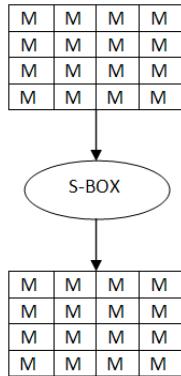
- **mix column**

Data bytes are mixed according to a mathematical function.

The last round is a little different because it skips the mix column function, but adds an additional key addition step.

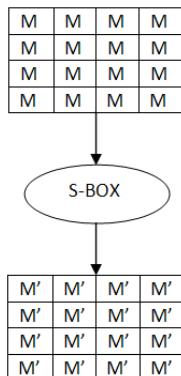
When masking is used, two different approaches can be followed:

- Simple masking:



In this case the same boolean mask value M is used for both inputs and output. Key bytes are guessed by simply assuming a value (a candidate) and then computing intermediate data after data addition and substitution. The intermediate values before and after the substitution are then XOR-ed, such that the masking is removed. The traces are then preprocessed so that every resulting sample represents the approximation of the XOR of two intermediate samples. The correlation of the intermediate data with the preprocessed traces is computed and the candidate resulting in the best correlation is assumed to be correct.

- Advanced masking:



In this case substitution input and output values are masked with a different value, therefore the previous method will not work. This method is divided in two steps. The first step tries to find the value of two key bytes at the same time. One of the two is always the first one, while the second one can be chosen with the "Key byte" field. By default the fifth key byte is chosen. These bytes are guessed after the substitution has been performed and then XOR-ed together, so that the M' mask is removed. At the end of the first step some intermediate informations are printed. Example:

```

Step 1 of Advanced masking second order AES analysis finished
Keybyte 0: 0x0F at position: 632
  
```

Keybyte 4: 0x47
Ratio: 0.2648

The second step can now be started: all the remaining key bytes are guessed and XOR-ed with the known first byte. Since the position of the first byte is known (and automatically set by the module in the 'First' field; be careful not to change it otherwise the module will not give correct results), this step requires a relatively small amount of time to be performed.

Related tutorials

- **Second Order AES Analysis Tutorial (simulation)** [\[../tutorials/sectionsSecondOrderAES.html\]\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsSecondOrderAES\]](#)
- **Second Order AES Analysis Tutorial** [\[../tutorials/sectionsSecondOrderAEScard.html\]\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsSecondOrderAES\]](#)

E.6.5 DES Advanced Analysis

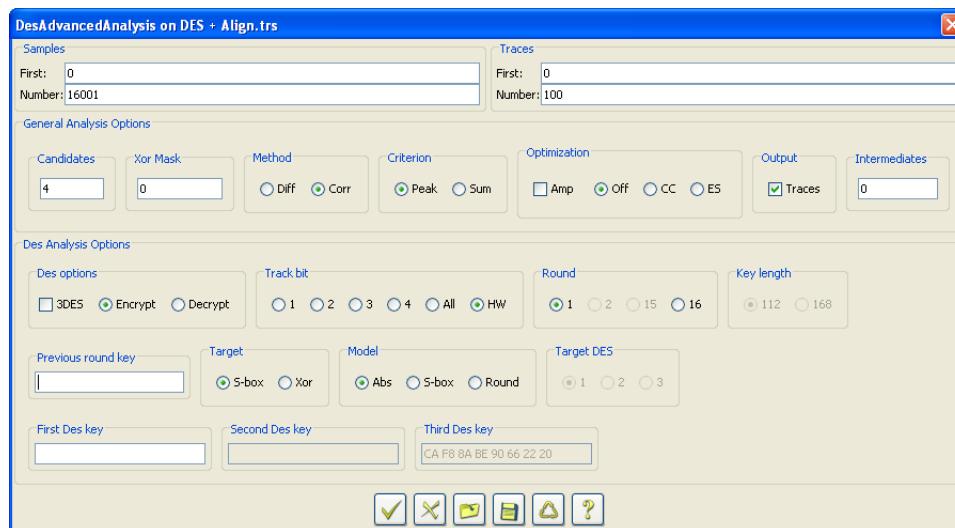
Version: SCA

Purpose

To retrieve a DES key using sophisticated statistical analysis methods.

The *DesAdvancedAnalysis* module provides a basis for sophisticated differential side channel analysis experiments on Des. The module extends AdvancedDifferentialAnalysis which implements the basic features that can also be used by other algorithm specific modules.

Dialog



Input

This help file explains advanced analysis settings. For a basic understanding of differential analysis read the *DesAnalysis* help file.

General Analysis Options:

- **Candidates:** The number of candidates displayed in the list of best key candidates for each S-box.
- **Xor Mask:** A hex mask applied to the target before correlation. This allows for inversion of some target bits. For example hex mask A inverts bit 1 (msb) and 3 of an S-box output. This can be useful when seeking correlation with the hamming weight of all target bits while the bits have different polarity.
- **Method:** The computation method of result traces.
 - **Diff:** means a classical differential trace where all traces corresponding to a data bit set to 1 are collected in one group while all other traces are collected in the second group. The result is given by subtraction of the average trace of both groups. This method is relatively fast, but also sensitive to noise.
 - **Corr:** means a correlation trace where result traces are computed as a sequence of correlation coefficients of sample columns with data columns. This method is slightly more resource consuming, but produces better results in the presence of noise.
- **Criterion:** The way of selecting a good correlation.
 - **Peak:** Just search for the highest peak in the result trace.
 - **Sum:** Add all values in the selection that exceed a noise threshold given by $2/\sqrt{n}$, where n is the number of traces.
- **Optimization:**
 - **Amp:** means an amplified correlation trace where the mathematical computation of correlation is amplified by a modified standard deviation. See below for more detail.
 - **Off:** No CC/ES applied.
 - **CC:** cross-correlation. Here we take into account that incorrect candidates may also produce significant peaks, and incorporate the strength of various ghost peaks. See below for more info.
 - **ES:** like CC, but uses Euclidean Similarity as a measure instead of correlation. See below for more info.
- **Output:** Produce differential/correlation result traces.
- **Intermediates:** Number of intermediate results to produce before the final output is to be presented
- **Track bit:** The S-box output bit to correlate the samples with. We can select each bit individually, or all together or take the hamming weight (HW). With all we actually compute 4 differential traces per candidate, and then sum the absolute values. With HW we compute one differential trace per candidate where we the power leakage corresponds to the hamming weight of the data.
- **Round:** The round to apply the analysis to. Initially the attack must be applied to round 1 or round 16. If these rounds yield a correct round key it is possible to proceed to round 2 or 15.

- **Key length:** In case of a Triple DES implementation, select whether the implementation uses two or three different keys.
- **Target DES:** In case of a Triple DES implementation, select which DES operation is being analyzed.
- **3DES:** The attack is applied to a Triple DES implementation. If the box is checked, 3DES specific options will be enabled.
- **Encryption/Decryption** (DES) or **EDE/DED** (3DES): Whether the analysed DES is doing an encryption or decryption.
- **Target:** The intermediate data who's output is used for predicting key bits
 - S-box: use the output bits of the S-boxes directly
 - Xor: use the intermediate data after exclusive-or with the left half of the data L, at the end of the round. Note that when attacking from the end, earlier intermediates will be used for attacking a round key. For example, if round is used in combination with xor, the module will attack the result of $R15 \wedge L15$ in order to retrieve subkey 16. This differs from DesKnownKeyAnalysis, because in that case the key is known and attacking the last round is performed by computing the algorithm from the beginning, thus using the input. Therefore when using "xor", the DesKnownKeyAnalysis module will use the value of L before the xor instead of after, attacking $R16 \wedge L16$ instead of $R15 \wedge L15$. Even though the intermediates are different, the modules should return the same correct subkey candidates.
- **Previous round key:** The round key found for the first round and used for the second round
- **First Des key:** The DES key found for the first DES. This key will be used to preprocess the data when attacking the middle DES.
- **Second Des key:** The DES key found for the second DES. This key will be used to preprocess the data when attacking a 168 bits 3DES implementation.
- **Third Des key:** The DES key found for the third DES. This key will be used to preprocess the data when attacking the middle DES.
- **Model:** A smart card may leak information related to the absolute value of intermediate data, or relative to some data earlier present in the process
 - **Abs:** Leakage is conform the absolute value of the target intermediate data
 - **S-box:** Leakage is relative to the S-box input value
 - **Round:** Leakage is relative to the round input value.

Amp method

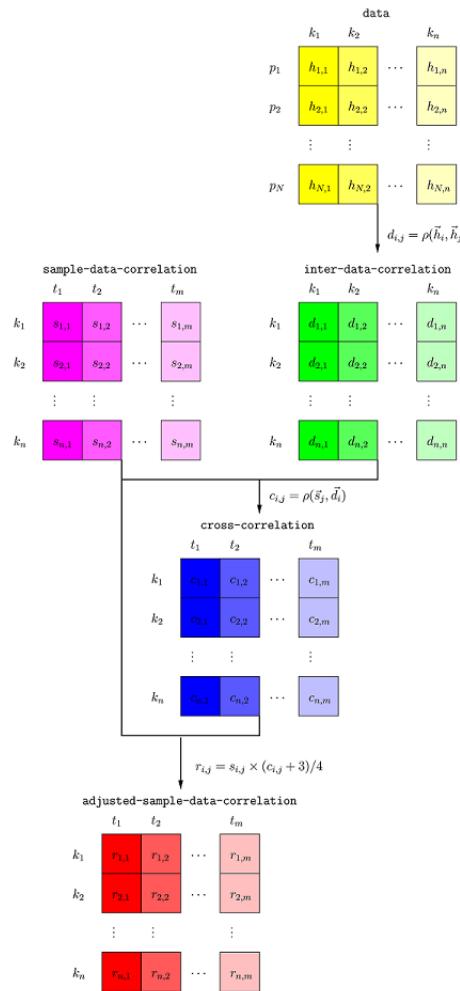
In the normal correlation computation the covariance is divided by the product of standard deviation of samples and standard deviation of data. The standard deviation is normally computed as the square root of the variance. We replace the standard deviation of the samples by the modified standard deviation, which is computed as the square root of the sum of the variance for samples taken for a data bit set to one and samples taken for a data bit set to zero. This modified standard deviation should be slightly less for an optimal division of sample groups.

Normally correlation is computed as $\text{cov}(P,D) / \text{std}(P) * \text{std}(D)$, with P the power measurements and D the hypothetical power consumption (hamming weight). Now we replace $\text{std}(P)$ with $\sqrt{\text{sum}(N_h * \text{var}(P_h)) / \text{sum}(N_h)}$. The function $\text{var}(P_h)$ calculates the variance in power consumption for the set of traces that have hamming weight h, and N_h denotes the number of traces in this set.

Note that since a modified correlation formula is used, the resulting correlation values can be larger than 1 (especially for traces with very low noise, such as simulations).

For more information please refer to Appendix A of "Statistic and secret leakage" (Coron, Naccache and Kocher, ACM TECS 04, FC 00)

CC method



The Cross-Correlation (CC) method is an optimizer of Differential Power Analysis (DPA). Flattening the ghost peaks, the CC method enhances a DPA attack purely based on properties that are inherited from the cryptographic algorithm. Possible correlation between different key candidates that is originally the cause of the ghost peaks are translated in this method to be of avail to the attack.

CC method adds three steps to a basic DPA attack, which are illustrated in the figure above.

Step 1: Correlating the data vectors. First, the data vectors (the yellow grid) are correlated with each other. For each candidate an inter-data-correlation vector (the green grid) is obtained representing the correlation of this candidate with all different candidates.

Step 2: Comparing the inter-data-correlation and the sample-data-correlation. Next, the sample-data-correlation vectors (the magenta grid) are treated and a cross-correlation vector (the blue grid) is computed for each time sample, containing the correlation between the sample-data-correlation vector at this time sample and all inter-data-correlation vectors. These sample-data-correlation vectors represent the correlation of the actual samples with all candidates. When they are compared with the inter-data-correlation vectors, the best match is expected to occur for the correct candidate and the correct time sample.

Step 3: Adding the cross-correlation into the sample-data-correlation. Finally, the cross-correlation is used to adjusted the sample-data-correlation. Every sample-data-correlation value is tuned down by an amount that is inversely proportional to the corresponding cross-correlation value with a maximum of 50% and a minimum of 0.

The final result of the attack is then determined based on the adjusted-sample-data-correlation vectors (the red grid).

ES method

The ES method is very similar to CC. It does not cross-correlate candidates, but rather uses Euclidean Similarity to determine the best candidates. Based on our experiences, both CC and EDPA have advantages and disadvantages over each other depending on the implementation under attack. While EDPA is good at locating the correct sample(s) (which corresponds to the execution of the targeted intermediate value), CC is good at eliminating ghost peaks that occur at the correct sample(s). As a rule of thumb, we suggest to use CC to attack software implementations and ES to attack hardware implementations.

See "Improving DPA by Peak Distribution Analysis" at <http://www.riscure.com/news-events/improving-dpa-by-peak-distribution-analysis> for details.

Related tutorials

- **DES Advanced Analysis tutorial** [\[../tutorials/sectionsAdvancedDES.html\]](#) [\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsDESCardTutorial\]](#)
- **Analysis of a DES implementation** [\[../tutorials/sectionsDESCardTutorial.html\]](#) [\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsDESCardTutorial\]](#)
- **Analysis of a DES implementation with random delays** [\[../tutorials/sectionsDESCardRndDelayTutorial.html\]](#) [\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsDESCardRndDelayTutorial\]](#)
- **Analysis of a DES implementation with S-box randomization** [\[../tutorials/sectionsDESCardRndSboxTutorial.html\]](#) [\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsDESCardRndSboxTutorial\]](#)

E.6.6 DES DFA

Version: FI

Purpose

Recovering a DES key by analyzing fault-injected outputs

Description

This module performs a Differential Fault Analysis on a trace set. Only data associated with the traces is processed, thus samples are ignored. The module expects the data to hold the 8 input bytes and the 8 output bytes. In case only 8 bytes are present, the module assumes only the output is stored with the traces. For the attack to work, at least one trace must contain the original, not fault-injected DES output, while the other traces must contain the same input with an injected fault.

Analysis Options



- **Round:** The round to attack. If faults in the current trace set are injected in a single round, select the targeted round. In case faults are spread in both the rounds (or even in the full 16 rounds), select the *both* option.
- **Reference Trace:** The trace that contains the original, correct output.
- **T-Des:** If selected, attack the second DES of a Triple DES operation.
- **Decrypt:** If selected, attacks a DES decrypt operation (or 3DES DED).
- **Attenuate:** When attacking the second DES of a Triple DES operation, if faults are evenly injected in the whole 48 rounds, because of an algorithmic property both the last round key of the second DES and the first round KEY of the last DES will be identified as correct candidates. To overcome this issue, if this option is selected the module will pick the last round key of the second DES over the first round key of the third DES in case the two candidates have a close likeliness to be the right one.
- **Detect round:** If faults are injected on multiple rounds, the module will try to detect for every trace whether the fault has been injected in the last round (or the fifteenth in case the analyst is attacking the penultimate round). The specified value is the hamming distance between the expected output and the glitched output. If an early round is glitched, the hamming

distance of both the halves is expected to be high. On the other hand, if only the last round is glitched, the hamming distance should be low, depending on the glitch effect and the algorithm implementation. By setting this field, the module will only use data with a hamming distance of the L part lower than the specified parameter. Usually values ranging from 8 to 16 yield the best results.

- **Previous round key:** The round key found for the first round and used for the second round. Only useful when attacking one round at a time.
- **Previous DES key:** The DES key found for the first DES. This key will be used to preprocess the data when attacking the middle DES in a 3DES operation.

Results

The module will output the results for the most likely S-Box candidates. The results are normalized such that the best candidate always has the value "1". Furthermore, the module generates a trace set which contains the same information in a trace form. The highest peak will be the most likely candidate for an S-Box.

Background

The idea behind the attack is to first inject faults on the fifteenth round. Such faults will corrupt the outcome of the round.

$$R_{16} = F(R_{15}, K_{16}) \oplus L_{15}$$

$$R'_{16} = F(R'_{15}, K_{16}) \oplus L_{15}$$

If the correct and the fault-injected R part of the sixteenth round (R_{16} and R'_{16}) are Xor-ed together, the result is the following: $R_{16} \oplus R'_{16} = F(R_{15}, K_{16}) \oplus F(R'_{15}, K_{16})$

This means that only K_{15} is unknown as R_{15} and R'_{15} are equal to L_{15} and L'_{15} .

The equation is then solved per S-box: K_{16} is split into 8 parts K_{16i} of 6 bits.

If R_{15} and R'_{15} differ for that S-box a single or few values of K_{16i} will match.

$$R_{16} \oplus R'_{16} = F(R_{15}, K_{16}) \oplus F(R'_{15}, K_{16}) \Rightarrow R_{16} \oplus R'_{16} = P(S(E(R_{15}) \oplus K_{16})) \oplus P(S(E(R'_{15}) \oplus K_{16}))$$

In terms of individual S-box results: $(P^{-1}(R_{16} \oplus R'_{16})) = S_i(E(R_{15}) \oplus K_{16i}) \oplus S_i(E(R'_{15}) \oplus K_{16i})$

Experiments show that on average 8 of 64 candidates will match per S-box, so 3 bits entropy loss per S-box can result in 24 bits key space reduction if all S-boxes are affected. The attack can be repeated until all the bits are found. Once K_{16} is known, the attack can be iterated by injecting a fault on round 14, thus obtaining K_{15}

In case the trace set contains data with faults injected in different rounds, this data will increase the overall noise. However, with a large enough data set, the attack can be successfully performed, even if the faults are injected in unknown rounds.

Note



If the number of bytes in the communication data of a trace is greater than 16 then the 17th byte is interpreted as a timeout flag. If this timeout byte has value 1 the data of this trace is ignored.

Related tutorials

- **DFA attack on a DES implementation** [../tutorials/sectionsDESDFTutorial.html] [PDF] [../tutorials/Tutorials.pdf#sectionsDESDFTutorial]

E.6.7 DES Known Key Analysis

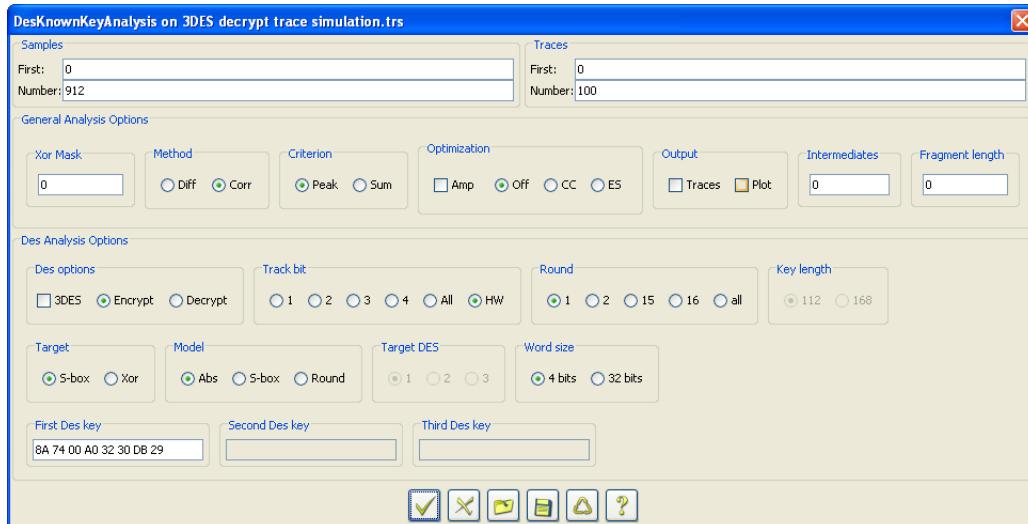
Version: SCA

Purpose

To determine data leakage of intermediate DES results using a known key.

The *DesKnownKeyAnalysis* module provides functionality to determine the location of data leakage related to intermediate results of the DES rounds inside a trace. The module extends the DesAdvancedAnalysis module, thus offering the same side channel distinguishers.

Dialog



Input

This help file explains known key analysis results. The module settings inherited from DesAdvancedAnalysis can be found in its own help file.

Known Key Analysis Specific Options:

- **Round:** The round to be analysed. It can be one round in the set 1,2,15 and 16 or the complete DES. The module will compute the results of the selected rounds and obtain its leakage.
- **Fragment length:** The module will divide the search interval in several fragments, providing its output per fragment. In this way, it's possible to determine in which fragment a certain round is located.
- **Plot:** Checking this option will instruct the module to generate a 2D representation of the results in a new dialog window.

- **Word size:** When set to 4 bits, the module analyses only the bits related to a given S-box. When set to 32 bits, the module looks at 32 bits hamming weights, setting the bits not related to the analysed S-box to their correct value. This is useful to maximize the correlation when all the target results are being computed at the same time. Since 32 bits are used, all the correct sub-key candidates will have the same correlation value. When the correlation value happens to be very low compared to the wrong candidates, vertical blue lines can appear in the plot generated by the module.

Result

The module produces two kinds of results: textual and graphical. On the textual side, for each analysed round and fragment the ranking for each of the known candidates is provided.

Furthermore, at the end of the detailed output, the module provides a summary with the average ranking per round and fragment, and an average for all rounds per fragment in case all the rounds are analysed.

The following listing provides the output relative to the first round when dividing a trace into several fragments and analysing the complete DES:

```
Analysing round 1 (round key: 88 05 BC 20 C8 12)
Best correlation S-Box 1 in the range 0 till 20:
0: sub key: 34, value: 0.5564, at position: 2
Best correlation S-Box 1 in the range 20 till 40:
40: sub key: 34, value: 0.1364, at position: 22
Best correlation S-Box 1 in the range 40 till 60:
35: sub key: 34, value: 0.1509, at position: 59
Best correlation S-Box 1 in the range 60 till 76:
42: sub key: 34, value: -0.1312, at position: 71
Best correlation S-Box 2 in the range 0 till 20:
0: sub key: 0, value: 0.5206, at position: 2
Best correlation S-Box 2 in the range 20 till 40:
22: sub key: 0, value: -0.1762, at position: 33
Best correlation S-Box 2 in the range 40 till 60:
15: sub key: 0, value: -0.1671, at position: 42
Best correlation S-Box 2 in the range 60 till 76:
5: sub key: 0, value: 0.2042, at position: 70
Best correlation S-Box 3 in the range 0 till 20:
0: sub key: 22, value: 0.5003, at position: 2
Best correlation S-Box 3 in the range 20 till 40:
57: sub key: 22, value: 0.1069, at position: 27
Best correlation S-Box 3 in the range 40 till 60:
19: sub key: 22, value: -0.1684, at position: 55
Best correlation S-Box 3 in the range 60 till 76:
22: sub key: 22, value: -0.1599, at position: 71
Best correlation S-Box 4 in the range 0 till 20:
0: sub key: 60, value: 0.4687, at position: 2
Best correlation S-Box 4 in the range 20 till 40:
28: sub key: 60, value: 0.1431, at position: 23
Best correlation S-Box 4 in the range 40 till 60:
39: sub key: 60, value: 0.1351, at position: 50
Best correlation S-Box 4 in the range 60 till 76:
34: sub key: 60, value: -0.1193, at position: 72
Best correlation S-Box 5 in the range 0 till 20:
0: sub key: 8, value: 0.4986, at position: 3
Best correlation S-Box 5 in the range 20 till 40:
54: sub key: 8, value: 0.1078, at position: 23
```

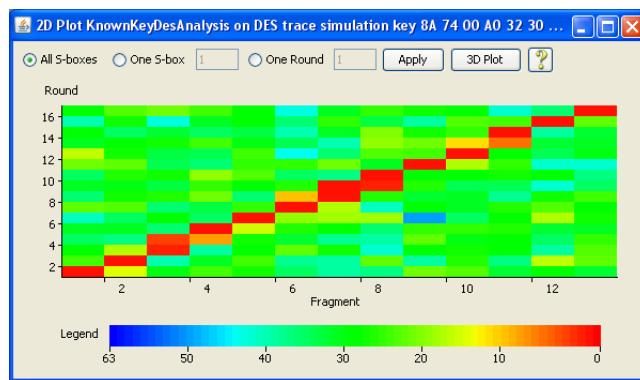
```

Best correlation S-Box 5 in the range 40 till 60:
34: sub key: 8, value: 0.1399, at position: 50
Best correlation S-Box 5 in the range 60 till 76:
61: sub key: 8, value: -0.0803, at position: 67
Best correlation S-Box 6 in the range 0 till 20:
0: sub key: 12, value: 0.4306, at position: 3
Best correlation S-Box 6 in the range 20 till 40:
33: sub key: 12, value: 0.1354, at position: 27
Best correlation S-Box 6 in the range 40 till 60:
54: sub key: 12, value: -0.1146, at position: 57
Best correlation S-Box 6 in the range 60 till 76:
61: sub key: 12, value: 0.0918, at position: 68
Best correlation S-Box 7 in the range 0 till 20:
0: sub key: 32, value: 0.6120, at position: 3
Best correlation S-Box 7 in the range 20 till 40:
61: sub key: 32, value: -0.1022, at position: 26
Best correlation S-Box 7 in the range 40 till 60:
3: sub key: 32, value: -0.1973, at position: 49
Best correlation S-Box 7 in the range 60 till 76:
7: sub key: 32, value: 0.2006, at position: 71
Best correlation S-Box 8 in the range 0 till 20:
0: sub key: 18, value: 0.4595, at position: 3
Best correlation S-Box 8 in the range 20 till 40:
3: sub key: 18, value: 0.2101, at position: 37
Best correlation S-Box 8 in the range 40 till 60:
32: sub key: 18, value: -0.1376, at position: 49
Best correlation S-Box 8 in the range 60 till 76:
33: sub key: 18, value: 0.1270, at position: 64

```

One line is presented per each S-box in a given fragment. The first number is the ranking the correct candidate achieved. If this number is zero, it means that the correct candidate obtained the best result. If it is 63, it means it obtained the worst result. Next to the index, the correct sub key, the correlation and the position in the trace where the value is obtained are printed.

However, this output can be very overwhelming and difficult to examine when analysing a complete DES with several fragments. To assist the analyst in this task, the module also provides a graphical output in the form of a 2D plot:



This 2D plot provides a representation of the analysis results. The X axis represents the different fragments analysed, and the Y axis represents either the different rounds or the different S-boxes.

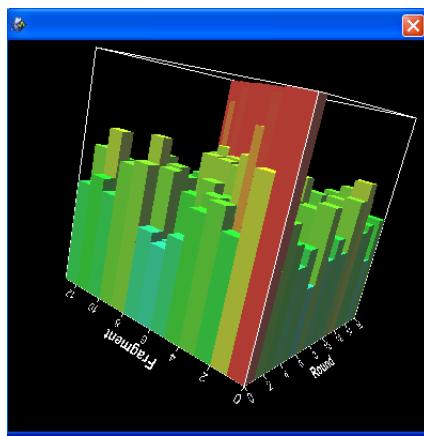
When all the rounds are analysed, the default plot shows the average results per row. It is also possible to select the results for one S-box and all the rounds, or for one round and all S-boxes. Selecting them in the appropriate radio button and pressing the Apply button repaints

the new view. When only analysing one round, the only available view is for all the S-boxes of the given round.

A legend is shown for the interpretation of the results. A pure red spot means that the correct result was ranked in the first position, and a pure blue spot means that it was ranked in the last position.

The existence of a DES implementation is easy to recognize in this case, seeing the *stairs-like* figure drawn by the red spots. This indicates the presence of all the rounds one after each other.

Finally, it is also possible to obtain a 3D view of this plot, representing higher bars for the best result and lower bars for the worst one:



E.6.8 DES Masked Input Analysis

Version: SCA

Purpose

Recovering a DES key when the input is partially unknown

Description

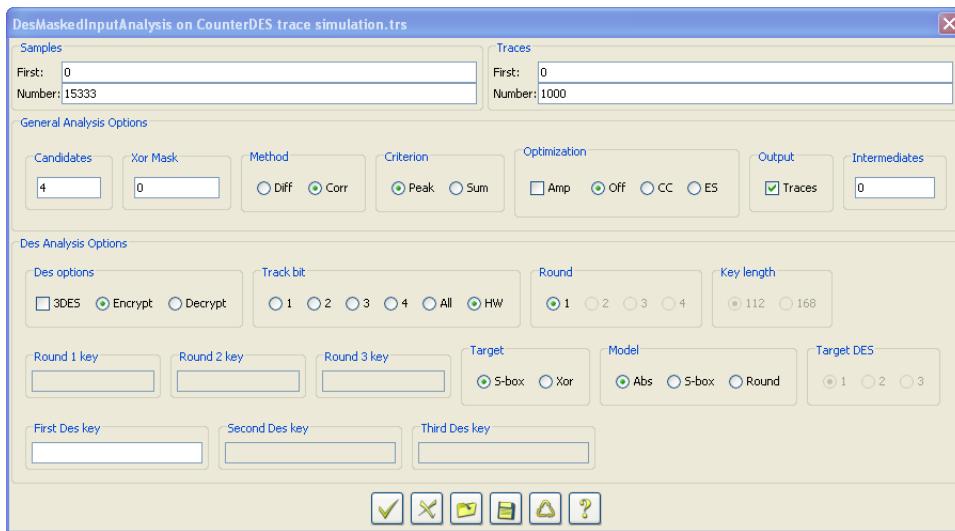
The standard Inspector DES modules assume the analyst knows the full plain text. This module is for use in situations where part of the plain text is unknown.

The plain text is represented as d^m where d is the known part and m an unknown mask (and \wedge denotes XOR). The input to the S-boxes in round 1 can be written $K1 \wedge E(RH(IP(d^m)))$ where $K1$ is the round key, E is the DES E-permutation, RH takes the right half and IP is the DES initial permutation. But this is the same as $(K1 \wedge E(RH(IP(m)))) \wedge E(RH(IP(d)))$ - so we can apply side channel analysis as if the plain text were just d , recovering a round key which is the real round 1 key XOR-ed with $E(RH(IP(m)))$.

Similarly, the input to the S-boxes in round 2 can be written $K2 \wedge E(LH(IP(d^m))) \wedge (\text{result of } f \text{ in round 1})$, where $K2$ is the round key and LH takes the left half. So we can apply side channel analysis as if the plain text were just d , recovering a round key which is the real round 2 key XOR-ed with $E(LH(IP(m)))$.

After analyzing 4 rounds, we have sufficient information to reduce the number of possible key values to 2. This cannot normally be reduced any further because of the DES duality ($\sim\text{DES}(k,d) = \text{DES}(\sim k, \sim d)$). However, if both input and output are known, the module can verify which of the two keys is the correct one.

Input



All fields in the input dialog are the same as for DES Advanced Analysis (see Section E.6.5, “DES Advanced Analysis”), except that the analysis is carried out on rounds 1,2,3 and 4 instead of 1 and 2 or 15 and 16. Thus, there are 3 fields for entering the keys of previous rounds.

Results

The results are also as for DES Advanced Analysis except that, at the end of round 4, the 2 possible key values and corresponding input mask values are also calculated.

Related tutorials

- **DES Masked input tutorial** [[..../tutorials/sectionsDESMaskedInput.html](#)][PDF] [[..../tutorials/Tutorials.pdf#sectionsDESMaskedInput](#)]

E.6.9 DES Partly Constant Analysis

Version: SCA

Purpose

Recovering a DES key when the input is partly constant

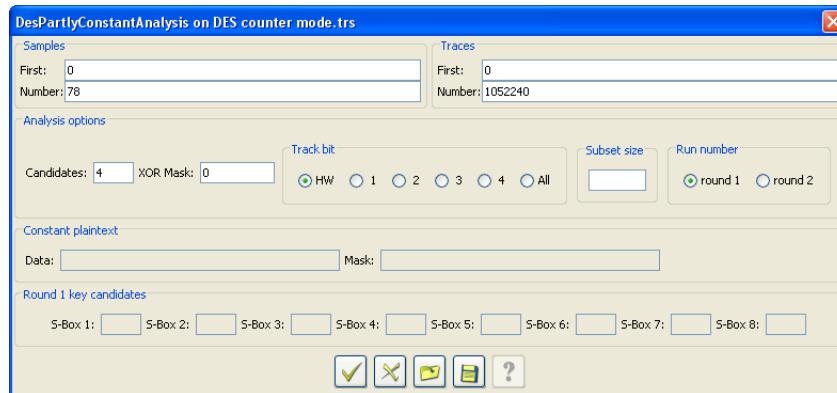
Description

The standard Inspector DES modules assume the analyst has full control over the plain text and can make it random. This module is for use in situations where the full plain text is known, but part of it is constant, causing the usual correlation attack to fail (for example, with DES in counter mode).

Precisely how many inputs bits are constant and which they are is variable: the module assumes only that the analyst knows the full input and has stored it as the first 8 bytes of data with each trace.

The module is normally run twice: the first run determines which input bits are constant and their values, and performs standard DPA on 1st round S-boxes with 3 or more varying inputs. The second run performs DPA on 2nd round S-boxes, using hypotheses built from constant bits and key candidates supplied by the 1st run and bit functions for the rest, resulting in a correlation list of function choices for each 2nd round S-box. These and the 1st run results are then used to calculate possible key values.

Input



- **Samples, Traces:** As in all modules which read in traces, these fields give which traces to process and which sample range within them
- **Round number:** Which round processing to perform (see description above)
- **Subset size:** By setting a value in this field, a random trace set subset of the specified size will be used instead of the whole trace set. It is useful when attacking DES in counter mode. See below for an explanation on how to use this field.
- **Candidates:** The number of key candidates (round 1) or function choices (round 2) to display as output
- **XOR Mask:** If "track bit" is set to HW, a 32-bit mask can be given (in hexadecimal, 4 bits per S-box) which is XOR-ed with the S-box output before taking the Hamming weight, for chips which leak the Hamming distance from a constant value.
- **Track bit:** Correlation can be performed between the sample values and the Hamming weight of each S-box output (HW), or a specified bit of each S-box output (1-4) or with all S-box output bits individually (All) - in which case the correlations are summed for each candidate (so the maximum value is 4 instead of 1).
- **Constant plaintext (round 2 only):** The mask field gives which input bits are constant and the data field specified what those constant values are. These fields are filled in automatically after the 1st round, but can be amended by the user. Both are 64 bit hexadecimal values.
- **Round 1 key candidates (round 2 only):** For each S-box with 3 or more varying inputs (as determined by the constant plain text mask above), 1 or 2 key candidates from the first round are carried over for processing in the 2nd round. These fields are filled in

automatically, but can be amended by the analyst. The values are in hexadecimal with 2 digits per candidate.

Results

Round 1 analysis results in a correlation list of key candidates for each S-box, along with the constant plain text data and mask. Round 2 analysis gives a correlation list of function choices for each S-box, and the possible key values consistent with all choices giving the highest correlation.

Usage with DES in counter mode

DES in counter mode can be represented as partly constant input DES, where the non-constant part is the counter. However due to how DES works, it is required that several bits are varying in the input. A minimum of 12 bits of entropy is required for the attack to work. In case of counter mode this means at least 4.096 traces. However the more the entropy bits are the more efficient the attack is, since the first round attack will discard more candidates. For Example, when 24 bits of entropy are present compared to 16, the time for the second round attack (with an equal number of traces) can speed up by a factor of up to 1000. For this reason, we suggest to acquire a large amount of traces when possible (even several millions), and then to reduce their number by using the *Subset size* field. This way a smaller subset of the large trace set can be selected without reducing the total entropy.

Related tutorials

- **DES Partly constant input tutorial** [[..../tutorials/sectionsDESPartlyConstant.html](#)][PDF] [[..../tutorials/Tutorials.pdf#sectionsDESPartlyConstant](#)]

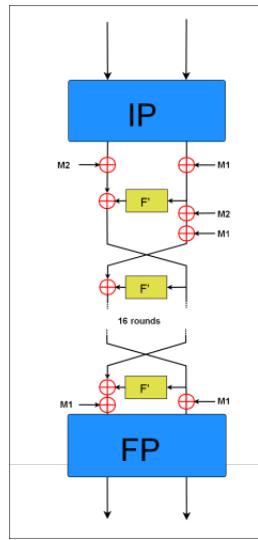
E.6.10 DES Second Order Analysis

Version: SCA

Purpose

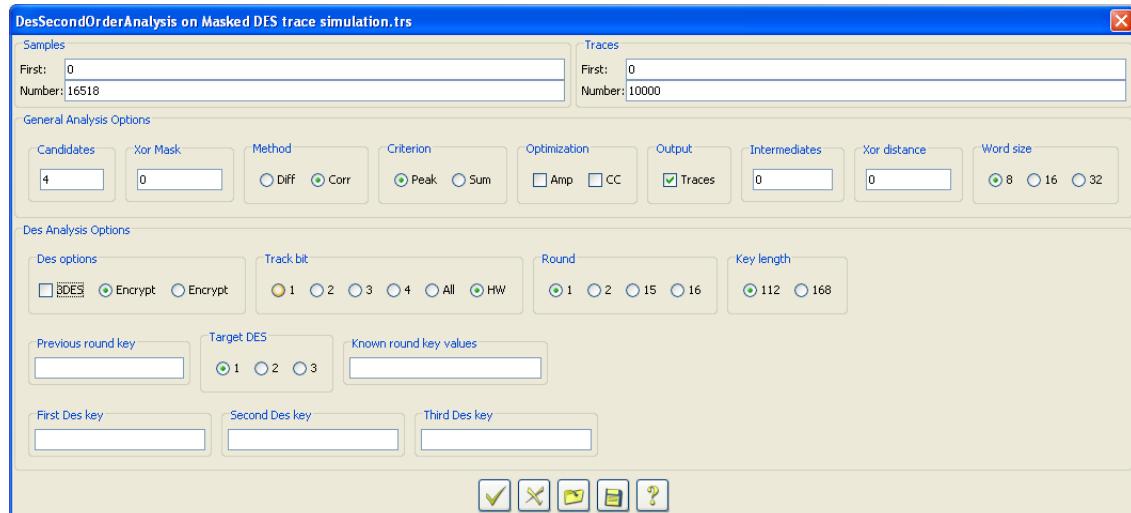
To retrieve a DES key used in a masked implementation.

The *DesSecondOrderAnalysis* module provides a basis for second order differential side channel analysis experiments on DES. The module extends *AdvancedDifferentialAnalysis* which implements the basic features that can also be used by other algorithm specific modules. The masking scheme of a single round is represented in the following image.



The module operates by trying to guess the XOR-ed value of the R input of two consecutive rounds. Since both of the inputs are XOR-ed with the same mask M1, the result of their XOR will remove the mask. A drawback of this method is that any R byte is influenced by all of the S-Boxes, therefore since the module still tries to guess one sub key at a time, the resulting noise will be higher, thus requiring a higher amount of traces to yield the same results as the first order DES Analysis module on non-masked implementations.

Dialog



Input

This help file explains advanced analysis settings. For a basic understanding of differential analysis read the DesAnalysis help file.

General Analysis Options:

- Candidates:** The number of candidates displayed in the list of best key candidates for each S-box.

- **Xor Mask:** A hex mask applied to the target before correlation. This allows for inversion of some target bits. For example hex mask A inverts bit 1 (msb) and 3 of an S-box output. This can be useful when seeking correlation with the hamming weight of all target bits while the bits have different polarity.
- **Method:** The computation method of result traces.
 - **Diff:** means a classical differential trace where all traces corresponding to a data bit set to 1 are collected in one group while all other traces are collected in the second group. The result is given by subtraction of the average trace of both groups. This method is relatively fast, but also sensitive to noise.
 - **Corr:** means a correlation trace where result traces are computed as a sequence of correlation coefficients of sample columns with data columns. This method is slightly more resource consuming, but produces better results in the presence of noise.
- **Criterion:** The way of selecting a good correlation.
 - **Peak:** Just search for the highest peak in the result trace.
 - **Sum:** Add all values in the selection that exceed a noise threshold given by $2/\sqrt{n}$, where n is the number of traces.
- **Optimization:**
 - **Amp:** means an amplified correlation trace where the mathematical computation of correlation is amplified by a modified standard deviation. In the normal correlation computation the covariance is divided by the product of standard deviation of samples and standard deviation of data. The standard deviation is normally computed as the square root of the variance. We replace the standard deviation of the samples by the modified standard deviation, which is computed as the square root of the sum of the variance for samples taken for a data bit set to one and samples taken for a data bit set to zero. This modified standard deviation should be slightly less for an optimal division of sample groups.
 - **CC:** cross-correlation. Here we take into account that incorrect candidates may also produce significant peaks. We correlate each of the data vectors (corresponding to key candidates) with each other. For each data vector we get a new data-cross-correlation vector (size: number of candidates) representing the correlation of one candidate with all different candidates. Next we take the correlation results from the sample vector with each of the data vectors. This sample-data-correlation vector (size: number of candidates) represents the correlation of the actual samples with each of the candidates. When we then correlate the sample-data-correlation vector with the data-cross-correlation vector we hope to get the best match. In fact we now do not only use the best results for the correct candidate, but we also incorporate the strength of various ghost peaks.
- **Distance:** The distance between the computation of the R part in two consecutive rounds. The values will be XOR-ed together to remove the masking. Can be either a single value if the correct distance is known or a range (e.g. 3-150')
- **Word size:** The number of bits that are processed simultaneously by hardware.
- **Track bit:** The S-box output bit to correlate the samples with. We can select each bit individually, or all together or take the hamming weight (HW). With all we actually compute 4 differential traces per candidate, and then sum the absolute values. With HW we compute

one differential trace per candidate where we the power leakage corresponds to the hamming weight of the data.

- **Round:** The round to apply the analysis to. Initially the attack must be applied to round 1 or round 16. If these rounds yield a correct round key it is possible to proceed to round 2 or 15.
- **3DES:** The attack is applied to a Triple DES implementation. After retrieval of the DES key for the first (or third) DES the analyst checks this box and proceeds with the middle DES. The input or output data is first encrypted with the earlier found DES key before being used for the analysis of the middle DES.
- **Previous round key:** The round key found for the first round and used for the second round
- **Previous Des key:** The DES key found for the first or last DES of a triple DES. This key will be used to preprocess the data when attacking the middle DES.
- **Known key values:** This module can be used in multi-pass mode, by just re-running it without changing any parameters. For every pass the module computes the s-box entry which is more likely to be correct. In the following execution this value will be kept constant: if the guess is right the correlation values will increase and lead to better results. Key values still not found are represented as "??". Note that the value in this field will look different from the content of the "Previous round key" field. This is because here the key is represented as eight 6-bits long values, while in the "Previous round key" field it's represented as six 8-bits long values.

Related tutorials

- **Second Order DES Analysis Tutorial (simulation)** [\[../tutorials/sectionsSecondOrderDES.html\]\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsSecondOrderDES\]](#)
- **Second Order DES Analysis Tutorial** [\[../tutorials/sectionsSecondOrderDES_card.html\]\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsSecondOrderDES_card\]](#)

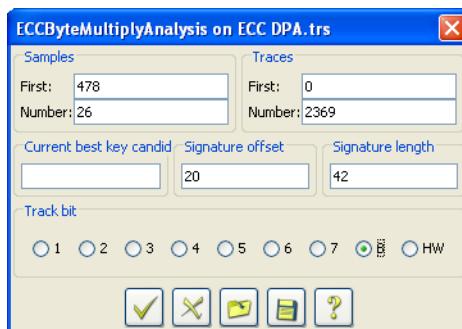
E.6.11 ECCByteMultiplyAnalysis

Version: SCA

Purpose

To investigate an ECC signature implementation's susceptibility to side channel analysis.

Dialog



- The *Current best key candidate* shows the current partial key candidate that has the best correlation. Note that a list of best candidates is stored internally.
- The *Signature offset* field can be used to specify the index of the signature in the trace set's data. The length of the signature (in bytes) should be entered in the *Signature length* field. Note that only half of this number will be used for correlation.
- You can specify the bit that is used for correlation, where bit 8 is the most significant bit and bit 1 is the least significant bit. In addition, the hamming weight (*HW*) of all bits can be used for correlation.

Input

This module requires a trace set that contains the multiplication of c by s . The signature output by the algorithm should be stored in the traces. The analyst should select each iteration of the multiplication loop.

Result

This module results in:

1. A trace set that shows the correlation between the selected part of the trace set and the intermediate data of the multiplier;
2. A list of the best candidates for the last iteration in the output window.

Background

Most ECC signature algorithms, such as ECDSA and ECNR, multiply the first part of the output signature c by the private key s to obtain the second part of the signature d . Suppose that the multiplication of c and s is performed byte-wise, as shown in the following figure.

| Bytes of C | | C_2 | C_1 | C_0 |
|-----------------------|-----|--|--|-------------------------|
| Bytes of S | | S_2 | S_1 | S_0 |
| Intermediates | | $C_2 \times S_0$ | $C_1 \times S_0$ | $C_0 \times S_0$ |
| Intermediates | | $C_2 \times S_1$ | $C_1 \times S_1$ | $C_0 \times S_1$ |
| Intermediates | | $C_2 \times S_2$ | $C_1 \times S_2$ | $C_0 \times S_2$ |
| Bytes of $C \times S$ | ... | LSB(MSB($C_1 \times S_0$) + MSB($C_0 \times S_1$) + LSB($C_2 \times S_0$) + LSB($C_1 \times S_1$) + LSB($C_0 \times S_2$) + carries) | LSB(MSB($C_0 \times S_0$) + MSB($C_1 \times S_1$) + LSB($C_1 \times S_0$) + LSB($C_0 \times S_1$)) | LSB($C_0 \times S_0$) |

First of all we assume that c is known as it is part of the output signature. In the first iteration of the multiplication algorithm, C_0 is multiplied by S_0 . We perform a correlation between the selected part of the trace and the Hamming weight of the least significant byte of the product $C_0 \times S_0$. We use the known value of C_0 and use a hypothesis for the value of S_0 , $0 \leq S_0 \leq 255$. Instead of the Hamming weight, it is also possible to use the individual bits of the product for correlation which may yield different leakage characteristics.

For a simulated trace set, this step results in a list of candidates for S_0 and their corresponding correlation value:

Best correlation:

```
0, candidate = 0x01, correlation value: 0.4759, at position: 26
1, candidate = 0x21, correlation value: 0.4570, at position: 26
2, candidate = 0x02, correlation value: 0.4545, at position: 26
```

```
3, candidate = 0x84, correlation value: 0.4530, at position: 29
4, candidate = 0x04, correlation value: 0.4387, at position: 26
5, candidate = 0x82, correlation value: 0.4291, at position: 26
6, candidate = 0x08, correlation value: 0.4264, at position: 29
7, candidate = 0x42, correlation value: 0.4255, at position: 26
8, candidate = 0xC2, correlation value: 0.4230, at position: 29
Partial key estimate: 1
```

For this experiment, we know that S_0 is 0x84. This value occurs in the list at position 3. As we perform differential analysis on a multiplication algorithm, some other values will also show a good correlation (e.g. powers of two). Therefore, we cannot assume that the best candidate is always the correct key byte and consequently less good candidates should also be considered as candidates for the next iteration.

In the next iteration, we compute the correlation between the selected part of the trace and the Hamming weight of the least significant byte of $MSB(C_0 \times S_0) + LSB(C_1 \times S_0) + LSB(C_0 \times S_1)$ for $0 \leq S_1 \leq 255$ and all best candidates for S_0 from the first iteration.

This results in a list of candidates for $S_1 S_0$ and their correlation value.

```
Best correlation:
0, candidate = 0xB784, correlation value: 0.4980, at position: 551
1, candidate = 0x8001, correlation value: 0.4604, at position: 602
2, candidate = 0x6F08, correlation value: 0.4473, at position: 551
3, candidate = 0x3784, correlation value: 0.4309, at position: 551
4, candidate = 0xDB2C, correlation value: 0.4197, at position: 551
5, candidate = 0xC001, correlation value: 0.4107, at position: 602
6, candidate = 0xF784, correlation value: 0.4051, at position: 551
7, candidate = 0x8002, correlation value: 0.4001, at position: 602
8, candidate = 0x5BC2, correlation value: 0.3989, at position: 551
Partial key estimate: B7 84
```

In contrast to the first iteration, the best candidate is the correct one. This process can be continued until the complete key is determined.

Related tutorials

- **ECC DPA Tutorial (simulation)** [\[../tutorials/sectionsECCDPASimTutorial.html\]\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsECCDPASimTutorial\]](#)
- **ECC DPA Tutorial** [\[../tutorials/sectionsECCDPATutorial.html\]\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsECCDPASimTutorial\]](#)

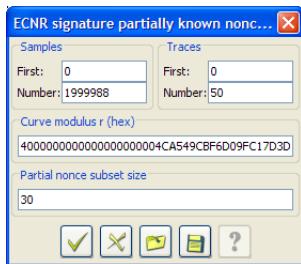
E.6.12 ECNRPartialNonceAnalysis

Version: SCA

Purpose

This module implements private key recovery using partially known nonces extracted from an ECNR signature implementation. The partial nonces in each trace represent the most significant bits of the nonce as used in the calculation of the ECNR signature. This module must be used in the final phase of an SPA attack on the ECNR signature scheme. For more details see the **ECC SPA Tutorial** [\[../tutorials/sectionsECCSPATutorial.html\]\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsECCSPATutorial\]](#).

Dialog



Input

- In the input traces each sample represents a bit from the partially known nonce, and therefore has value 0 or 1. The most significant bit is located at the start of the trace; the least significant bit of the partial nonce is at the end. This format is chosen such that the output of earlier trace processing can be used as input to this module.
- The signature c,d for each trace is assumed to be at the end of the entire data field of each trace, and are both encoded from MSB to LSB using the same number of bytes. For e.g. an 167-bit signature the last 42 bytes are considered the signature; the first 21 bytes of data are c, and the last 21 bytes of data are d.
- The module requires the modulus r of the curve to be specified in hex.
- The partial nonce subset size is used in the final processing stage of the module. It determines the number of partial nonces that are used in tandem to perform the attack.

Result

For each subset of partial nonces, the module outputs the private key guess. If these are (almost) all the same, the private key is found. Otherwise, the input needs to be tuned.

Background

In "The insecurity of the elliptic curve Digital Signature Algorithm with partially known nonces", Nguyen and Shparlinski describe an attack on ECDSA that, with fair probability, can retrieve the private key s based on a number of partially known nonces for a number of known signatures. We refer to the original paper for the details. The basic principle underlying the nonce recovery is that with the known signatures, we can span a lattice L of points Pi that contains the information about valid signatures with their complete nonces. The lattice L is fully known; however, the complete nonce information is hidden.

On the other hand, we have valid signatures and partial nonces. We can use this information to calculate a point Q, which is very close to some lattice point Pi. The only difference between Q and Pi is caused by the missing nonce bits. Since these are the lower order bits of the nonce, the distance is very small. The point Pi can be found by solving the CVP (Closest Vector Problem). The difference between Q and Pi now gives the missing nonce bits. Now we have one full nonce, from which the private key s can be directly calculated.

The success of the attack depends on the characteristics of the signatures. In some cases the CVP solver does not yield the point Pi we are interested in. Fortunately, we can retry the attack with a different set of signatures to obtain a new chance of getting the right key.

The partial nonce subset size is used in processing stage of the module to determines the number of partial nonces that are used in tandem to perform the attack. An attack may not always succeed. The subset size determines the balance between computation time and the probability of success: the smaller the size, the smaller the probability of success but the faster the attack. To give an idea of the range for the subset: we need a minimum subset size of 35 to attack partial nonces of 7 bits, and we need a minimum subset size of 29 to attack partial nonces of 30 bits.

Because we may have more partial nonces than the subset size, we iterate the attack with different subsets of partial nonce and signature data. For each iteration, we output the private key guess. If (almost) all keys are equal, we know we have found the right key. Otherwise, the input parameters need to be tuned.

References and license

The following components are used to perform the lattice attack:

fplll-3.0.12 [<http://perso.ens-lyon.fr/damien.stehle/downloads/libfplll-3.0.12.tar.gz>]

mingw [<http://www.mingw.org>]

The GNU MP Bignum Library [<http://gmplib.org>]

The MPFR library [<http://www.mpfr.org>]

Related tutorials

- **ECC SPA Tutorial** [[..../tutorials/sectionsECCSPATutorial.html](#)][PDF] [[..../tutorials/Tutorials.pdf#sectionsECCSPATutorial](#)]

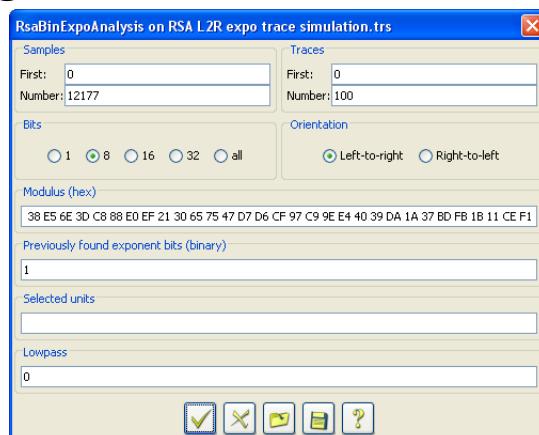
E.6.13 RsaBinExpoAnalysis

Version: SCA

Purpose

To recover the exponent used in a binary exponentiation implementation of the RSA algorithm.

Dialog



Inputs

The *Samples* and *Traces* panels are used to set the scope of the analysis. The sample scope should include at least one modular operation of algorithm.

The *Bits* panel selects the number of bits that are processed simultaneously in the hardware.

The *Orientation* panel selects whether the exponentiation is performed left-to-right (MSB first) or right-to-left (LSB first).

The *Modulus* edit field contains the modulus used for the RSA computation. This value is needed to compute intermediate values. Note that the input traces for this module are expected to contain both input and output data whose length equals the length of the modulus.

The *Previously found exponent bits* edit field contains key bits that were found up to now. This field is updated after each analysis round.

The *Selected units* field contains a list of units that the analyst wants to display. If nothing is inserted, all the units will be displayed. Often it's the case that only some units leak, while others only contain noise. To improve the results, only the leaking ones can be selected. The list can be dash separated to specify a range, or comma separate to specify single units.

The *Lowpass* field contains the weight of the low pass filter to be applied to the correlation results. As in RSA it's likely that several adjacent samples will leak, a low pass will decrease the noise in the other areas of the correlation traces without cancelling the peak.

Result

The analysis method must be invoked repeatedly to retrieve the exponent bit by bit. For long RSA keys this method is not very efficient, but it may be serve well for proving the principle of RSA key extraction.

At each round a comparison is made between a square and a multiply operation, and depending on the correlation results the key exponent is updated with a zero or one. The most likely operation is determined by comparing correlation values of two different intermediate data values, whose occurrence is dependent on the actual exponent.

The module estimates the probability by computing a quality indicator defined as the quotient of the best correlation value and the sum of both correlation values.

Example

The module can be tested with data file *RSA expo trace simulation L2R*. This simulated trace set has no noise and requires only little traces to succeed.

After two rounds the result may look like:

```
Best correlation for key unit 0:  
0, key unit: 0 (0x00), value: 1.0000, at position: 99  
1, key unit: 1 (0x01), value: 0.3113, at position: 443  
..  
..  
Best correlation for key unit 31:
```

```
0, key unit: 0 (0x00), value: 1.0000, at position: 130
1, key unit: 1 (0x01), value: -0.2820, at position: 404
Next operation probably square (probability: 75%)
Key exponent now (binary): 100
```

For a left-to-right exponentiation the input is raised to the power of the known input bits, followed by either a square or a multiply. The results of these alternatives are correlated for all 'key units', where each unit represents a number of bits as selected in the user dialog. A right-to-left exponentiation works a little different, which is explained in the background section.

In case the analyst is able to complete the repeated execution of the module to retrieve all key bits, the module will check this with the known input and output values, and in case of success report:

```
Key complete, exponent: 13 90 7C 8C 53 CC 73 EE 23 AE 48 70 1E 31 CA...
```

Background

Binary exponentiation offers a relatively simple RSA implementation. This can be done in two functionally equivalent ways: left-to-right (L2R) or right-to-left (R2L).

Consider the RSA computation: $M = C^d \bmod N$

The L2R variant is performed as follows:

```
M = 1
for each bit di
    M = (M * M) mod N
    if (di == 1) M = (M * C) mod N
return M
```

Note that each round includes a square of the intermediate result M , and an optional multiplication of M with C (depending on the value of the corresponding key bit).

The R2L variant is performed as follows:

```
M = 1
for each bit di
    if (di == 1) M = (M * C) mod N
    C = (C * C) mod N
return M
```

The R2L variant looks rather similar to L2R, although it does not square the intermediate result M , but the cipher text C .

With side channel analysis the L2R variant is slightly easier to attack: At a stage i in the process we try to correlate two candidate intermediate values with the samples taken from the modular operation. One candidate value would be $M'_i = (M_i * C) \bmod N$, the other would be $M''_i = (M'_i * M_i) \bmod N$. Interestingly, M'_i is only computed if the corresponding key bit is one, while M''_i would only be computed if the key bit is zero, otherwise the square operation would compute: $M'''_i = (M'_i * M_i) \bmod N$. This observation makes it relatively easy to figure out the value of the

next bit: just compare the correlation values of M'_i with correlation values of M''_i . The strongest correlation tells us whether there is a square or a multiply at that stage.

The R2L variant is more difficult to automate: The multiplication $M'_i = (M_i * C) \bmod N$ is only executed for key bits set to one, but the squaring $C'_i = (C_i * C) \bmod N$ is always executed. We can therefore not state that only one of the intermediate values can correlate. If C'_i correlates stronger than M'_i it could mean that M'_i is not computed, or that it just correlates slightly less (e.g. due to noise) than C'_i . The module hardly solves this dilemma: it slightly amplifies the M'_i correlation to promote the probability that a genuine correlation of M'_i results in the identification of a key bit set to one. It is important for the analyst to carefully compare correlation values of both candidates in subsequent analysis rounds and make sure the correct choices for the key exponent bits are used.

Related tutorials

- **RSA BinExpo Tutorial (simulation)** [\[../tutorials/sectionsRSABinExpoSimTutorial.html\]](#) [\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsRSABinExpoSimTutorial\]](#)
- **RSA BinExpo Tutorial** [\[../tutorials/sectionsRSABinExpoTutorial.html\]](#) [\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsRSABinExpoTutorial\]](#)

E.6.14 RSA CRT DFA

Version: FI

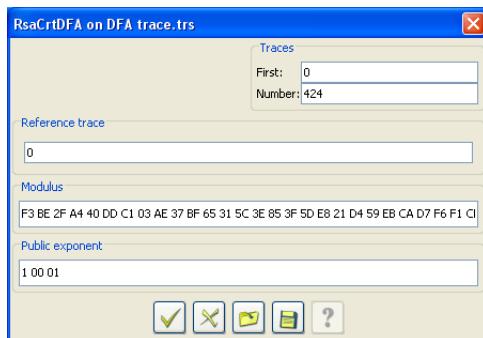
Purpose

Recovering a RSA private key by analyzing fault-injected outputs

Description

This module performs a Differential Fault Analysis on a trace set. Only data associated with the traces is processed, thus samples are ignored. The module expects the data to hold the output bytes. For the attack to work, at least one trace must contain the original, not fault-injected RSA output, while the other traces must contain the same input with an injected fault.

Analysis Options



- **Reference Trace:** the trace that contains the original, correct output.

- **Modulus:** the modulus used for the RSA computation. Note that the input traces for this module are expected to contain output data whose length equals the length of the modulus
- **Public exponent:** the public exponent used for the RSA verification. The analysis does not need the public exponent to find the primes. However, without the public exponent this analysis method cannot compute the private exponent

Results

If the module succeeds, P and Q primes will be printed. In case the public exponent is known, also the private exponent will be printed.

Background

The module implements an attack known as the "Bellcore attack". It targets the exponentiation phase of the CRT to obtain a faulty output. The RSA CRT execution consists of the following phases:

- $M_p = C^{d_p} \bmod p$
- $M_q = C^{d_q} \bmod q$
- $M = ((M_q - M_p) * K) \bmod q * p + M_p$

If a fault is injected in step 1 or 2, a faulty M_p or M_q will be returned. Suppose M_q fails, let's call M'_q the faulty result.

The computation output is $M' = ((M'_q - M_p) * K) \bmod q * p + M_p$.

Subtract M' from M : $M - M' = (((M_q - M_p) * K) \bmod q * p - ((M'_q - M_p) * K) \bmod q * p) = (x_1 - x_2) * p$

Compute $\text{Gcd}(M - M', n) = \text{Gcd}((x_1 - x_2) * p, p * q) = p$. This means one of the two factors has been recovered.

Since the public modulus is known, computing the other factor is trivial: $q = n / p$

As long as the fault is injected in the modular exponentiation phase, this attack needs only one fault to recover the whole private exponent.

Related tutorials

- **DFA attack on a RSA CRT implementation** [\[../tutorials/sectionsRSACRTDFATutorial.html\]](#) [\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsRSACRTDFATutorial\]](#)

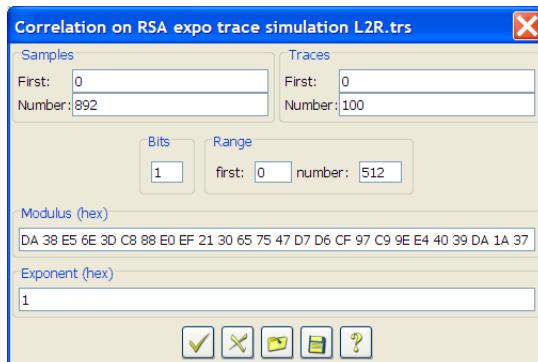
E.6.15 RsaCorrelation

Version: SCA

Purpose

To explore the exponentiation scheme used for a specific implementation of the RSA algorithm.

Dialog



Inputs

The *Samples* and *Traces* panels are used to set the scope of the analysis. The sample scope should include at least one modular operation of the algorithm.

The *Bits* edit field selects the unit size, e.g. the number of bits that are processed simultaneously in the hardware.

The *Range* panel is used to select the units to correlate, and consequently the number of correlation traces to be computed. For example, with a data size of 512 bits, and a unit choice of 8 bit in the *Bits* edit field a maximum of 64 units can be correlated.

The *Modulus* edit field contains the modulus used for the RSA computation. This value is needed to compute intermediate values.

The *Exponent* edit field contains the trial key bits for which correlation is to be computed.

Result

The analysis method can serve to demonstrate side channel leakage of an RSA implementation. Furthermore it can be used to verify that specific modular operations are executed. The latter is helpful in reconstructing the RSA exponentiation scheme. For example, the following schemes can be detected:

- binary exponentiation (left-to-right or right-to-left)
- k-ary exponentiation (left-to-right or right-to-left, various group sizes)

The resulting traces show the correlation computed between the hamming weight of the bits for each unit with the samples.

Background

Detecting a specific exponentiation scheme requires knowledge on how different schemes work. See help files RsaBinExpoAnalysis and Rsa High Order Analysis for more information on specific implementations.

For example, consider a situation where the first six consecutive modular operations show correlation with the following powers of c : c , c^2 , c^3 , c^6 , c^{12} , c^{13} . This could correspond with a left-to-right binary exponentiation with respective operations: multiply, square, multiply, square, square. These match with key fragment: '1101'.

The same key fragment would for right-to-left binary exponentiation show correlation with the following values: $c, c^2, c^4, c^5, c^8, c^{13}$.

Finally, with left-to-right k-ary exponentiation and group size two, this key fragment would result in correlation to the following: c^3, c^6, c^{12}, c^{13} .

Therefore, when correlation is tested for a number of consecutive powers (e.g. c, c^2, c^3, c^4, c^5) it is possible to asses the specific scheme implemented.

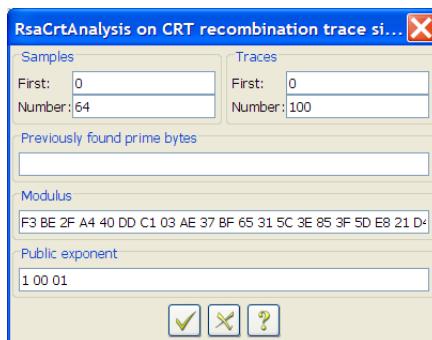
E.6.16 RsaCrtAnalysis

Version: SCA

Purpose

To recover a prime value used in a CRT implementation of the RSA algorithm.

Dialog



Inputs

The *Samples* and *Traces* panels are used to set the scope of the analysis. The sample scope should include the recombination step in the CRT algorithm.

The *Previously found prime bytes* edit field can be left empty at first. During the course of the analysis this field will be filled with prime bytes found so far.

The *Modulus* edit field should contain the modulus used for the RSA computation. The analysis strictly does not need the modulus to find a large part of one prime. However, without the modulus this analysis method cannot fully retrieve both primes.

The *Public exponent* edit field may contain the public exponent used for the RSA verification. The analysis does not need the public exponent to find the primes. However, without the public exponent this analysis method cannot compute the private exponent.

Result

The analysis method must be invoked repeatedly to retrieve the prime byte by byte. At each invocation the method will present the next prime byte, which is a result of a correlation computation. The actual value of a prime byte may in reality deviate slightly from the presented value, but this will be corrected in subsequent rounds.

After two rounds the result may look like:

```
Best correlation:  
0, sub key: 300 (0x012C), value: 0.9807, at position: 1  
1, sub key: 301 (0x012D), value: 0.8297, at position: 1  
2, sub key: 299 (0x012B), value: 0.7398, at position: 1  
3, sub key: 302 (0x012E), value: 0.7193, at position: 1  
Partial prime estimate: F3 D4
```

After finding P-2 prime bytes (where P is the length of the prime) the analysis method will automatically do a brute force to find the remaining two bytes, and compute the matching prime (if the modulus is present). Also the private key exponent will be computed in case the corresponding public key exponent was provided in the user dialog. The result may look like this:

```
Best correlation:  
0, sub key: 130 (0x82), value: 0.9969, at position: 61  
1, sub key: 129 (0x81), value: 0.7729, at position: 61  
2, sub key: 131 (0x83), value: 0.6906, at position: 61  
3, sub key: 136 (0x88), value: 0.6420, at position: 61  
Partial prime estimate: F3 D3 C8 AB 3A FE 18 09 8C D2 80 A8 0D E3  
CD B6...  
Prime search complete, factors found  
p: F3 D3 C8 AB 3A FE 18 09 8C D2 80 A8 0D E3 CD B6 80 99 F9 9C C5  
63 B5...  
q: FF E9 52 F1 B5 39 2E E8 A0 24 26 DA 2C F5 F1 D3 28 D9 B4 0D DE  
77 26...  
Private key exponent found:  
d: 63 16 7D 82 AE 36 18 39 D9 50 E3 77 8D 65 FD 2A A9 34 91 62 73  
53 EC...
```

If the analysis failed another result will be presented:

```
Best correlation:  
0, sub key: 11 (0x0B), value: -0.4338, at position: 14  
1, sub key: 495 (0x01EF), value: -0.4185, at position: 60  
2, sub key: 51 (0x33), value: 0.4158, at position: 27  
3, sub key: 1094 (0x0446), value: 0.4017, at position: 31  
Partial prime estimate: F3 D3 C8 AB 3A FE 18 09 8C D2 80 A8 0D E3  
CD...  
Prime search complete, factors not found
```

The analysis failure can also be seen from the relatively low correlation of the best prime value compared to what was found in the successful conclusion above.

In case the analysis is successful but the modulus is not available the following result is presented:

```
Best correlation:  
0, sub key: 130 (0x82), value: 0.9969, at position: 61  
1, sub key: 129 (0x81), value: 0.7729, at position: 61  
2, sub key: 131 (0x83), value: 0.6906, at position: 61  
3, sub key: 136 (0x88), value: 0.6420, at position: 61  
Partial prime estimate: F3 D3 C8 AB 3A FE 18 09 8C D2 80 A8 0D E3  
CD...  
Prime search complete, last few bytes cannot be found with this  
method
```

Background

For background of the algorithm and attack, please refer to Section K.8.1, “Analysis steps”.

E.6.17 RSA High Order Analysis

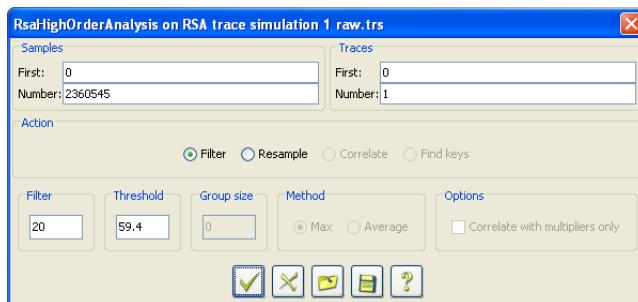
Version: SCA

Purpose

To investigate an RSA implementation's susceptibility to side channel analysis. This module uses so called High-order differential analysis, where modular operations are compared to each other using a mathematical correlation function.

Other modules that may be applied for RSA analysis are: RsaCorrelation, RsaExpoAnalysis, RsaCrtAnalysis.

Dialog



Input

Successful execution of the *RsaHighOrderAnalysis* module comprises the following steps:

1. **Acquisition** Collection of raw traces that cover the entire RSA exponentiation.
2. **Filtering** Removing noise to allow the modular operations to be characterised.
3. **Resampling** Replacing all samples taken during a single modular operation with a single one.
4. **Correlation** Computing a correlation matrix for the samples.
5. **Key retrieval** Deriving key values from the correlation matrix.

The first step is performed during the data acquisition phase and falls outside the scope of the *RsaHighOrderAnalysis* module. The other steps can be performed by the module. As the volume of data included in the raw trace can be extremely high, an analyst may want to already implement steps 2 and 3 during the acquisition phase and then proceed with the correlation step in this module using the reduced data set.

The purpose of the filtering is to remove noise and to obtain traces that can be resampled. The filter method included in *RsaHighOrderAnalysis* is a simple low-pass filter in which the sample array is processed as follows:

```
// make every sample dependent on the previous sample
for (int i=1; i<sample.length; i++){
    sample[i] = ( weight*sample[i-1] + sample[i] ) / ( weight+1); // weight sets the
    effect of the previous sample on the current sample
} // from right to left to avoid a sample shift
for (int i = sample.length-2; i>=0; i-- ){
    sample[i] = (weight*sample[i+1]) + sample[i] ) / ( weight + 1);
}
```

The weight parameter (default value: 20) can be set in the Filter panel. If weight is set to 0, no filtering is performed. A large value results in a low cut-off frequency, and may result in information loss. The value should be experimentally tuned to obtain result traces that have modular operations that can be distinguished by using a threshold value.

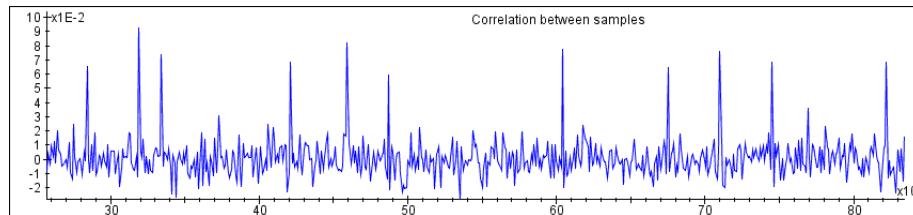
Resampling generates one sample for each modular operation. This is achieved by adding up all sequential samples above a threshold. In this way the new samples represent the amount of energy consumed during the modular operation. The function is invoked by choosing the Resample option in the modules dialogue.

It is possible to combine the Filter and Resample steps. If the signal is not yet filtered, it will be done on the fly, applying the filter parameter.

An essential parameter for resampling is the threshold value. It should be set to a mean value that allows modular operations to be distinguished from the normal processor activity. For the filtered signal in the above picture this would be a value of around 55. It is extremely important that the threshold value be appropriate for all modular operations in the entire trace set. There is no point in continuing the analysis if the input traces cannot be filtered to find a threshold that unambiguously allows all modular operations to be distinguished. Upon successful completion of this function, each trace in the resulting trace set will contain a number of samples exactly equal to the number of modular operations performed during one RSA decryption action, which should be between 25% and 55% more than the key length in bits.

The correlate function considers a trace set as a matrix in which each column represents multiple executions of the same modular operation within the exponentiation sequence. The correlate function creates a matrix of correlation coefficients in which each column is correlated with each column. The resulting matrix will be square, all values will be in the range from -1 to 1, and the diagonal will have all values set to 1, as it represents the correlation of columns with themselves. The *RsaHighOrderAnalysis* module assumes a key size of up to 4096 bits. Based on this assumption it tests whether the number of samples (modular operations) is below 6500, and if so, it enables the Correlate function in the module dialogue.

A successful completion of the correlate function will immediately indicate whether a key can be found. In that case several traces may resemble the trace in the figure below.



The peaks indicate multiplications that share the same operand. If none of the correlation traces contains significant peaks, it could mean that the modular operations do not leak information. The presence of these peaks indicates that it is possible to derive the private key.

The final step uses the correlation peaks to derive a key value. In a sense this is a superfluous step, because the presence of correlation peaks already demonstrates the vulnerability. On the other hand this step allows the analyst to prove the success of the attack by revealing the private key.

In this step it is possible to choose to only use cross-correlation between multipliers and other samples, or between all of the samples. Depending on how the exponentiation is implemented and on the leakage model only one of the two options may yield correct results.

The *RsaHighOrderAnalysis* module selects the Find Key function when the module is invoked on a correlation matrix (number of samples equals number of traces). Upon invocation the user should indicate a threshold value and may indicate a group size or method. The threshold value is used to distinguish genuine correlation from accidental correlation. The appropriate threshold value would be a little lower than the typical value of the peaks in the trace set with correlation values.

Result

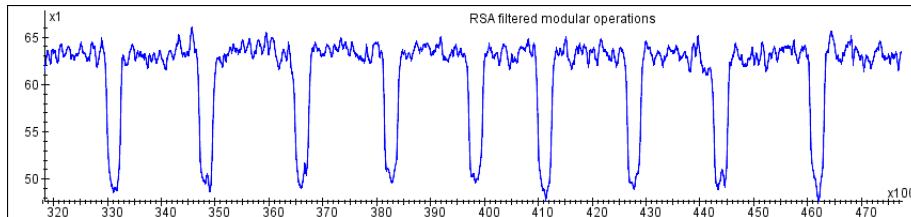
Since there are a lot of different RSA implementations, this final step may not always be successful even though the correlation peaks are present. The analyst may want to adapt the module code to suit a specific implementation. This is what the module currently attempts to do:

- Compute a reference trace from maximum or average values over the trace set. This reference trace will use the indicated threshold value to identify operations with recurring operands and will enable the analyst to distinguish multiplication from squaring operations. Using the maximum value trace often provides better results than the average value trace, but may be distorted by ‘ghost’ peaks. An analyst can select the best method experimentally. The function will return this reference trace, which allows the analyst to tune the appropriate threshold value.
- Find the key group size (default 1). If the key group size is not provided by the analyst in the dialogue, a frequency analysis is performed to detect this parameter.
- Find the multiplier generation (only if the key group is greater than 1). Multiplier generation can be recognized as it involves a sequence of multiplication operations without interleaved square operations.
- Identify multiplication factors. Multiplication positions are identified in the reference trace by comparison with the threshold value. For each multiplication operation, the correlation coefficients are compared with the multiplier generation operations. This should reveal the corresponding multiplier.
- Construct key. The number of interleaving square operations and the multiplication factors are used to construct the entire key.

Example

To enable you to easily get started, the module comes with two files, called RSA trace simulation 1 raw.trs and Rsa trace simulation 500 Energy.trs. The first file contains a single raw trace that can be used to practice low-pass filtering and resampling. The second file contains a set of 500 traces that are already resampled. Each sample represents the energy consumed during one modular operation. The second trace set can be used to demonstrate key retrieval.

After filtering, the raw trace might look similar to the figure below.



The output of the module might look like similar to this:

```
Chosen key group size: 3 // computed by frequency analysis
Multipliers start at: 0 // here starts the sequence of multiplier generation
skipping correlation at: 7 // ghost correlation by square operation is ignored
Multiplication positions: 6 9 13 17 21 25 32 36 40 44 48 52 56 60 64 68 72 76 80 84
88 92 96 100 107 111 ...
key parts: 11 111 001 001 110 111 000001 100 101 101 111 010 101 111 111 101 001
001 100 101 100 001 ...
key: 3E4E E0CB 7ABF D265 84F0 E636 FB7D 422E CAB3 9028 7676 6E35 393E E9BA 99B4
4B40 17FD 9AF6 6D85 ..
```

Background

RSA is one of the most popular cryptographic public key algorithms. Although much slower than symmetric algorithms, it is becoming more popular even in smart cards because of the ease of key management.

The RSA algorithm uses 3 numbers: n (modulus), e (public exponent) and d (private exponent). The numbers e and n are considered to be the public key, while d and n are considered to be the private key.

The public key is used for encryption, and for signature verification. The private key is used for decryption and signature generation. A message M is encrypted by computing $C = M^e \bmod n$ and decrypted by computing $M = C^d \bmod n$, i.e. using modular exponentiation.

Exponentiation is implemented by repeated (for each key bit) modular squaring and multiplying. The simplest form is called binary exponentiation, and uses the following procedure:

```
C = 1; // key k is an array of key bits
for (int bit = k.length-1; bit >= 0; bit-- ) { // for each key bit
    C = C * C; // always square
    if (k[bit] == 1) C = C * M; // only multiply for '1' bit
}
```

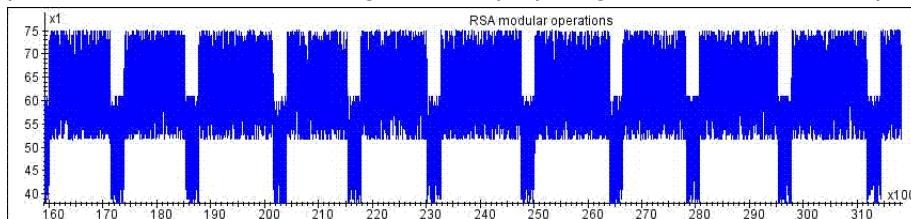
A more advanced implementation uses bit groups in which a set of multipliers M_i is precomputed:

```
int multipliers = 1 << groupSize; // precomputed multipliers = 2groupSize
M0 = 1;
M1 = M;
for (i=2; i < multipliers; i++){ // compute multipliers
    Mi = M * Mi-1;
}
C = 1; // key k is an array of bit groups, where 0 <= k[group] < multipliers
for ( int group=k.length-1; group >= 0; group-- ) { // for each group
    for (i=0; i < groupSize; i++)
```

```
C = C * C;
C = C * Mkey[group];
}
```

Grouped exponentiation is more efficient, as fewer multiplications are needed, but there is a little added overhead for preparing the precomputed multipliers. Furthermore, it is possible to skip the unnecessary multiplication in case key[group] = 0, because then $M_0 = 1$.

Modular operations will generally be executed by a dedicated co-processor that performs a rapid execution of big number modular multiplications. As the operation processes many bits simultaneously, it will very likely result in a temporary increase in power consumption. When the operation is completed, the power consumption drops back to a lower level as the normal processor takes care of moving data and preparing for the next modular operation.



The figure above shows a power consumption profile representing a sequence of modular operations during an RSA exponentiation. The modular operations can be recognized as the modules during which power consumption is much higher.

Modular square operations are similar to multiplications since they obviously are multiplications themselves. If the square operations could be distinguished from the multiply operations it might be possible to retrieve the private key during a decryption process. However, such an approach would only work for binary exponentiation. Other implementations may use bit groups to process multiple bits at a time where for instance a series of square operations is always alternated with a multiplication.

When applying square and multiply operations in a sequence, an important difference becomes apparent:

- The square operations have a variable operand, which is likely to be different for each execution.
- The multiply operations have one static operand (e.g. the message M for binary exponentiation).

Since separate multiplications may share a single operand, it may be possible to observe a correlation between these multiplications. This correlation may enable the analyst to distinguish the square operations from the multiplications, and may even allow multiplications that use different multipliers (as for grouped exponentiation) to be distinguished. This phenomenon lies at the basis of the analysis performed by the RsaHighOrderAnalysis module. This approach is fundamentally different from common DPA attacks, since we are not correlating crypto data with power traces, but correlating samples from the power traces with each other. This is a high-order DPA attack.

Related tutorials

- **RSA High Order Analysis Tutorial** [\[../tutorials/sectionsRSAHighOrderAnalysisTutorial.html\]](#) [\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsRSAHighOrderAnalysisTutorial\]](#)

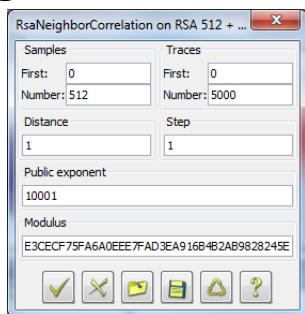
E.6.18 RsaNeighborCorrelation

Version: SCA

Purpose

To recover the exponent used in a binary exponentiation implementation of the RSA algorithm by doing cross correlation. This could be used to verify if a vulnerable multiply-always implementation is used. In this case adjacent multiply and square operations would share an argument if the multiply result was discarded. Shared operands may result in detectable correlation.

Dialog



Inputs

The *Samples* and *Traces* panels are used to set the scope of the analysis. The sample scope should include at least as many samples as bits to be recovered.

The *Distance* panel selects the distance between potentially correlating operations in RSA. By default this is one, but it could be several. For instance, if consecutive modular operation are resampled into one sample, we would start our sample scope with the first multiply (M) operation and set distance to 1.

The *Step* panel selects the step size while going through the samples. By default this is one, but it could be several. If each modular operation is compressed into one sample we would set step size to 2, but if each pair of modular operations is compressed to one sample we would set step size to 1.

The *Public exponent* input sets the exponent of the associated RSA public key. This value is by default hex 10001, and is used to verify the correctness of the recovered private exponent.

The *Modulus* input sets the modulus of the RSA key. A verification will not be done if the modulus is not set. The modulus is used to verify the correctness of the recovered private exponent.

Result

The analysis method can extract the entire private exponent at once.

At completion the extracted private key is tested against the provided public key (exponent and modulus). Single bit errors are corrected by a small brute-force search.

Example

The module can be tested with data file *RSA 512 bits + Pattern resampled*. This test set needs the default public exponent, and the following modulus:

```
E3CE CF75 FA6A 0EEE 7FAD 3EA9 16B4 B2AB 9828 245E 75CE D9F3 D67C 8DF9 6D58 4F5A 7C4F  
E6A5 A7D3 16AE FB3D 470A 1311 3E76 A26D 5A75 E41E EFBC 3C76 7055 39B8 F759
```

The result of running the module should be:

```
Found and corrected key at distance 1, threshold: 0.1767945776832206  
found binary key:  
101110101100010100011100111110000101010000100010111100000101010  
01111011100011010010101100010111000000111110011011111011001  
0100110110000111011000111100000111111010001011100101110000101100  
1111100100100101111001110011010010011111001111000110111011011001  
0100000110000110000001011111011100010111000011100110000001000111  
1000011100000111111100010011101011010011011111001100111100110111  
1110100011101001011000100001110011100011010011101101101101100000001  
0010010110011101101010111100001101001100111110011000000111  
(hex: 2eb1473f 0a845e0a 9ee34ace 5c0fcdf6 5361d8f0 7e8b970b 3e4979cd 27cf1bb6  
5061817d  
c5c39811 e1c1fc4e b4df33cd fa3a5887 38d3b6c0 49676af0 d33e7307)  
corrected binary key:  
101110101100010100011100111110000101010000100010111100000101010  
0111101110001101001010110011100101110000011111001101111011001  
  
0100110110000111011000111100000111111010001011100101110000101100  
1111100100100101111001110011010010011111001111000110111011011001  
0100000110000110000001011111011100010111000011100110000001000111  
1000011100000111111100010011101011010011011111001100111100110111  
111010001110100101100010000111001110001101001110110110110110000001  
0010010110011101101010111110000110100110011111001110000001111  
(hex: 5d628e7e 1508bc15 3dc6959c b81f9bec a6c3b1e0 fd172e16 7c92f39a 4f9e376c  
a0c302fb  
8b873023 c383f89d 69be679b f474b10e 71a76d80 92ced5e1 a67ce60f)
```

With this example it appears that the final modular operation is missing, as we got only 511 samples. The module tries several variants of the key and is able to correct for the missing bit.

E.6.19 SEED Analysis

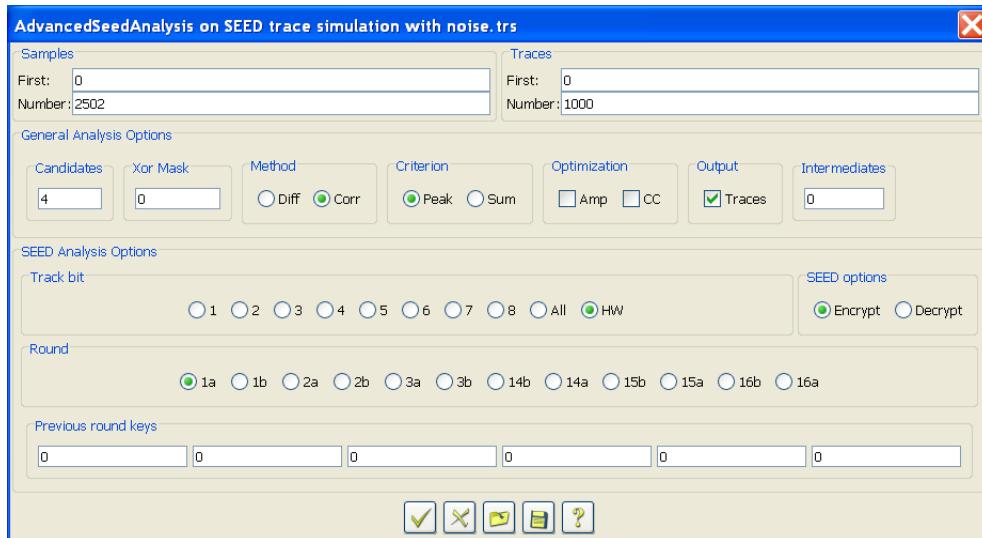
Version: SCA

Purpose

To retrieve a SEED key using sophisticated statistical analysis methods.

The *SeedAnalysis* module provides a basis for sophisticated differential side channel analysis experiments on SEED, a symmetric encryption algorithm used in Korea. The module extends *AdvancedDifferentialAnalysis* which implements the analysis features that can also be used by other algorithm specific modules.

Dialog

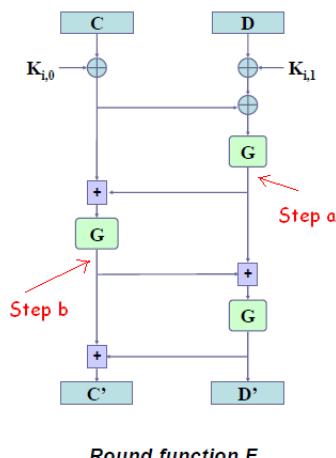


Input

This help file explains the settings for a differential analysis of the SEED algorithm. For a understanding the dialog options of differential analysis read the DesAdvancedAnalysis help file.

Analysis steps

The attack takes up to 6 steps (3 rounds, 2 steps each). Each step reveals 4 bytes of round key information. After the 4th step, enough information is collected to make a first attempt to find the key. This results in a list of at most 256 key candidates. This list is further refined in the 5th and 6th step, until only one candidate remains. An attack starts with step 1a or 16a, depending on whether the beginning or ending of the algorithm is targeted. The successive steps are 1a, 1b, 2a, 2b, 3a, 3b, or 16a, 16b, 15a, 15b, 14a, 14b. The 'b' steps may be run twice for optimal results due to a dependency between key bytes. Repeating a 'b' step will improve the correlation values for the first 3 key bytes found for that step. The figure below identifies the intermediate data used for the selection functions of the steps.



Related tutorials

- **DES Advanced Analysis tutorial** [\[../tutorials/sectionsAdvancedDES.html\]\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsAdvancedDES\]](#)
- **Analysis of a DES implementation** [\[../tutorials/sectionsDESCardTutorial.html\]\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsDESCardTutorial\]](#)
- **Analysis of a DES implementation with random delays** [\[../tutorials/sectionsDESCardRndDelayTutorial.html\]\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsDESCardRndDelayTutorial\]](#)
- **Analysis of a DES implementation with S-box randomization** [\[../tutorials/sectionsDESCardRndSboxTutorial.html\]\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsDESCardRndSboxTutorial\]](#)

E.7 Crypto2

E.7.1 Calculator

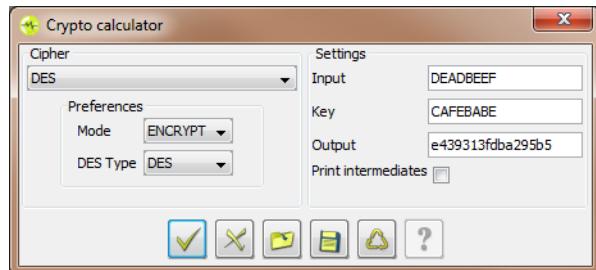
For explanations of, and attacks on, specific ciphers, please refer to Appendix K, *Cipher Suites*.

For explanations of functionality shared between multiple modules, please refer to Chapter 3, *Cryptography*.

Purpose

Calculate crypto output and intermediates given input and key

Dialog



Inputs

- Input / Output: specify parameters
- Print intermediates: if enabled, prints all intermediate states of the algorithm to the output window

E.7.2 First Order Analysis

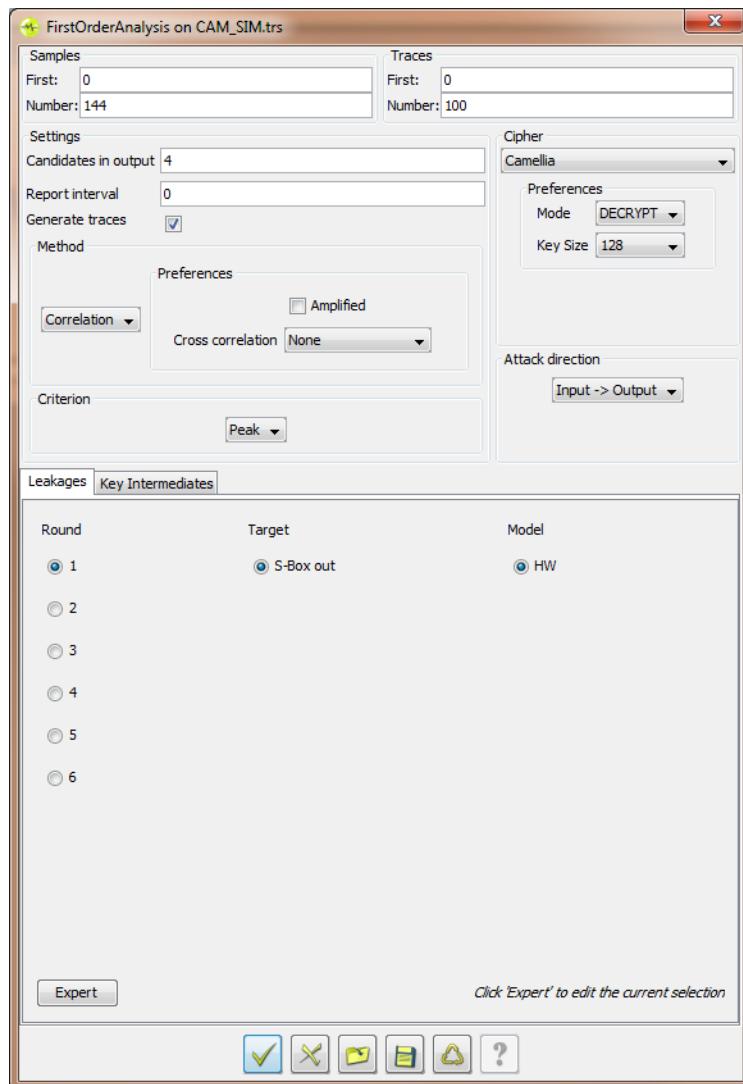
For explanations of, and attacks on, specific ciphers, please refer to Appendix K, *Cipher Suites*.

For explanations of functionality shared between multiple modules, please refer to Chapter 3, *Cryptography*.

Purpose

To retrieve the key of a supported crypto algorithm using sophisticated statistical analysis methods. The FirstOrderAnalysis module uses a framework that support several crypto algorithms.

Dialog



Inputs

None, other than the ones already detailed in Chapter 3, *Cryptography*.



Note

It is possible to perform Correlation Power Analysis in the *frequency domain*, rather than the time domain. This is accomplished by first running the Spectral

module and feeding the resulting trace set (i.e. the FFT traces) to the First-Order module. See Section E.9.14, “Spectral”.

E.7.3 AES Chosen Input First Order Analysis

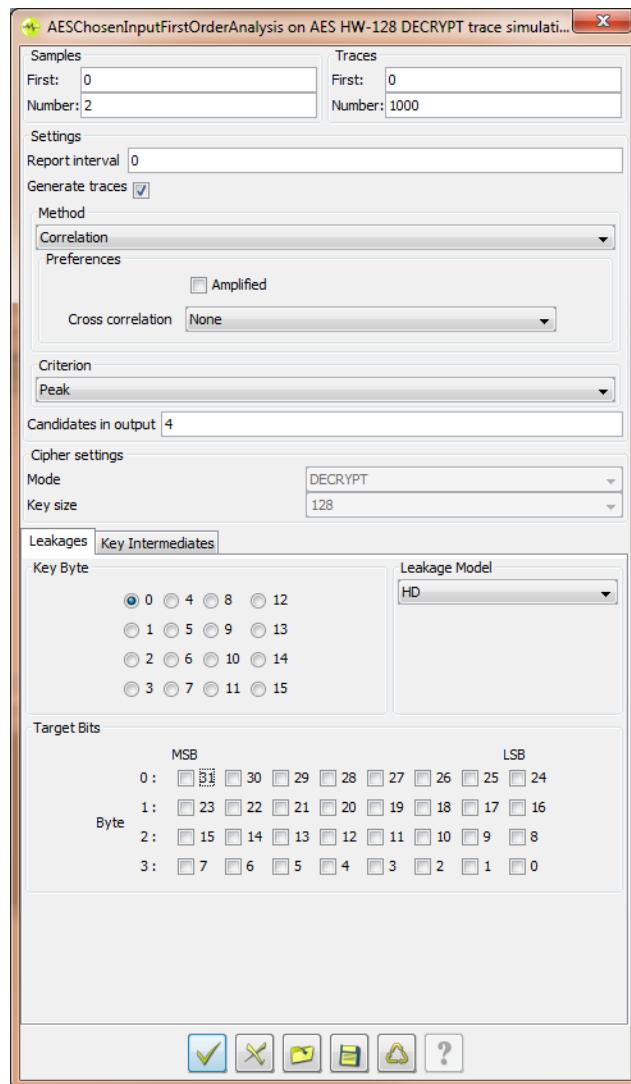
For explanations of, and attacks on, specific ciphers, please refer to Appendix K, *Cipher Suites*.

For explanations of functionality shared between multiple modules, please refer to Chapter 3, *Cryptography*.

Purpose

To retrieve the key of a hardware implementation of AES using input data. For additional information, see the ‘Background’ sub-section at the end of this section.

Dialog



Inputs

Since this module is designed specifically for 128-bit AES using input in decrypt mode, the cipher and direction panels found in Section E.7.2, "First Order Analysis" are not present in this module. To reflect the current (unchangeable) settings, an additional panel was added to the settings dialog.

Furthermore, this module has no Expert view, and a custom version of the Normal view. In this view the user has to make 3 selections: Key Byte, Leakage Model, and Target Bits.

- **Key Byte:** This panel allows the user to select the key byte that he wants to attack. As such, recovering a full 128-bit key requires 16 consecutive runs of this module.
- **Leakage Model:** This panel allows the user to select either Hamming Weight (HW) or Hamming Distance (HD). This module has implicit intermediates associated with these models, being: HW: HW(Round 1: Inverse Shift Rows In) and HD: HD(Round 0: Round Input, Round 1: Inverse Shift Rows In). The technical details of these leakages are explained in the 'Background' sub-section at the end of this section.
- **Target Bits:** This panel allows the user to select the bits that display the best leakage. As shown, the bits have been separated into a single column, where the last bit in the column relates to the 8 least significant bits in the vector.

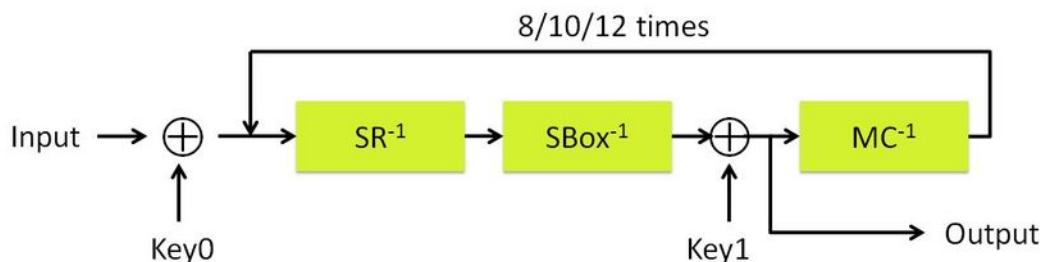
Results

See Section E.7.2, "First Order Analysis"

Background

This module targets a particular type of *chosen-input* side channel attack on AES, where the output of the *MixColumns* (or the *inverse MixColumns*) of an encryption (respectively decryption) is exploited to recover the *first* round key.

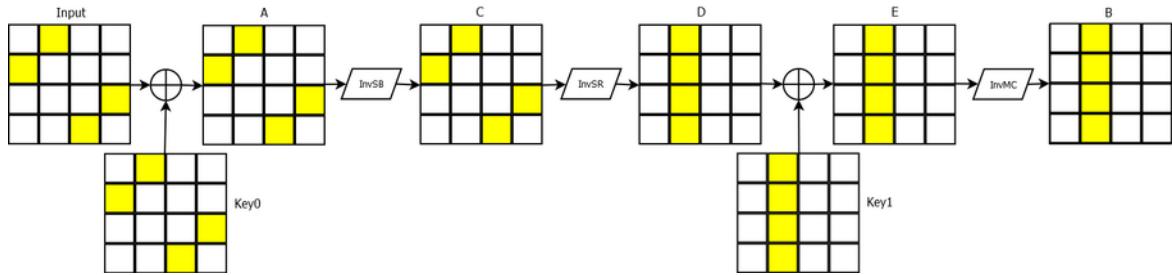
This attack is in particular inspired by a hardware design of AES implementation as illustrated in the diagram below. (The diagram shows only the case for decryption, while the design for encryption is analogue to decryption.)



In this design, the output of the inverse MixColumns are written into a register, which previously contains either the result of the first AddRoundKey (only for the first round) or the output of the inverse MixColumns from the previous round. This write may cause the value of the inverse MixColumns, and/or the XOR of the first AddRoundKey output and the first inverse MixColumns output, to leak through side channels. Since these intermediate values depend on the first round key, their leakages may be used in Side Channel Analysis to recover the first round key.

However, due to the structure of the AES algorithm, exploiting the (inverse) MixColumns output is not straightforward. Figure E.3, “AES decryption with all-random input data” illustrates the computation of one of the columns of the inverse MixColumns output in an AES decryption. One can see that every bit of the output is dependent on 4 bytes of the first round key and 4 bytes of the second round key (highlighted yellow in the figure). This size of the unknown keys is too large to be recovered in practice using the straightforward side channel attack (where completely random input data is used).

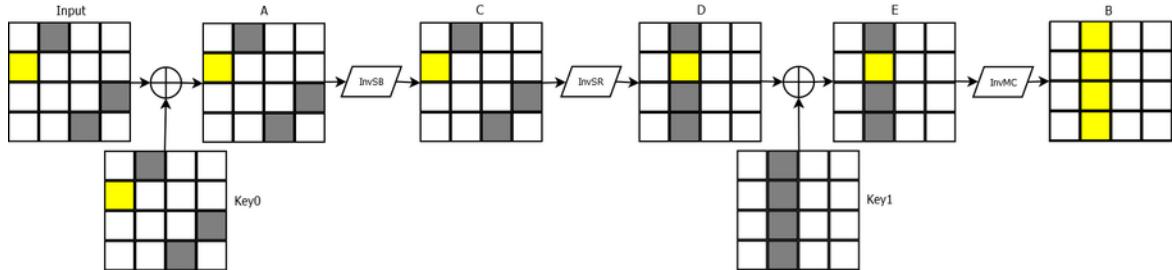
Figure E.3. AES decryption with all-random input data



To circumvent this issue, one can fix part of the input data constant, which will lower the number of key bytes to attack and therefore make the attack feasible in practice. How many and which of the input bytes to fix constant are dependent on the index of the key byte to recover.

Figure E.4, “AES decryption with partly-constant input data” shows the computation of the same column of the inverse MixColumns output, as illustrated earlier in Figure E.3, “AES decryption with all-random input data”, but with 3 (out of 4) of the dependent input bytes fixed constant (highlighted gray in the figure).

Figure E.4. AES decryption with partly-constant input data



As a result, the corresponding bytes (in gray) of the intermediate bytes are constant, leaving key byte Key0(0,1) (in yellow) the only unknown variable that influences the targeted (highlighted) column of the inverse MixColumns. This allows us to test all the 256 candidates of byte Key0(0,1) by correlating any bits in the highlighted column of the inverse MixColumns to the measured side channel signals, and therefore recover the value of Key0(0,1).

This attack is possible only if a single bit in the targeted column is correlated to the measurement at a time. Nonetheless, one can perform the attack on several (if not all) bits in the same column and integrate their correlation results. Combining the targeted bits will generally strengthen the attack.

Please note that for different key byte, a different subset of input bytes should be fixed constant for this attack to work properly. It is up to the user to make sure that the correct input settings are given to the measurement prior to the attack.

References

[1] Amir Moradi, Markus Kasper, and Christof Paar, "Black-Box Side-Channel Attacks Highlight the Importance of Countermeasures – An Analysis of the Xilinx Virtex-4 and Virtex-5 Bitstream Encryption Mechanism", CT-RSA 2012, LNCS 7178, pp 1-18, 2012. Springer-Verlag Berlin Heidelberg 2012

E.7.4 Known Key Correlation

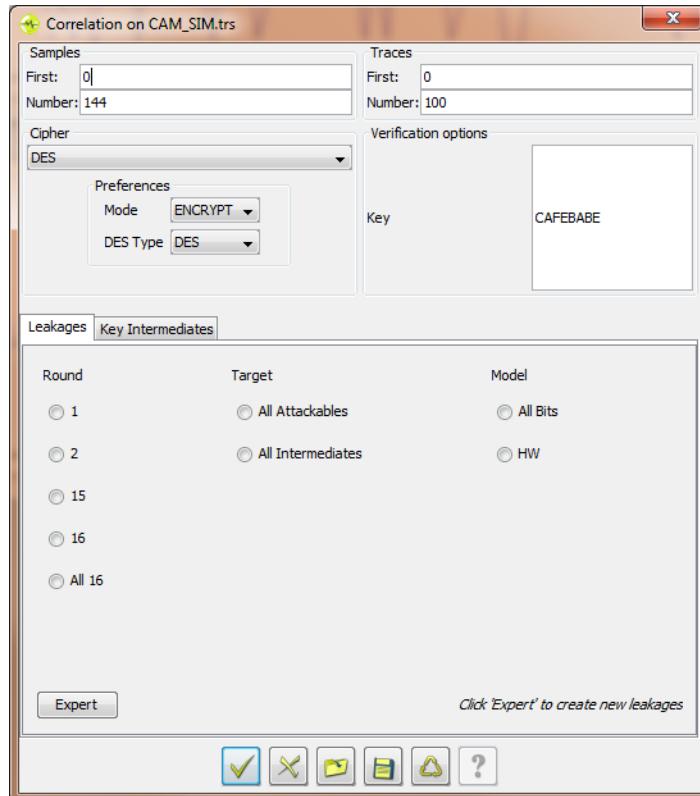
For explanations of, and attacks on, specific ciphers, please refer to Appendix K, *Cipher Suites*.

For explanations of functionality shared between multiple modules, please refer to Chapter 3, *Cryptography*.

Purpose

Find correlation with intermediate values for different leakage models.

Dialog



Inputs

None, other than the ones already detailed in Chapter 3, *Cryptography*.

Results

A trace set is returned that for each intermediate value and each leakage model gives a correlation trace.

E.7.5 Simulator

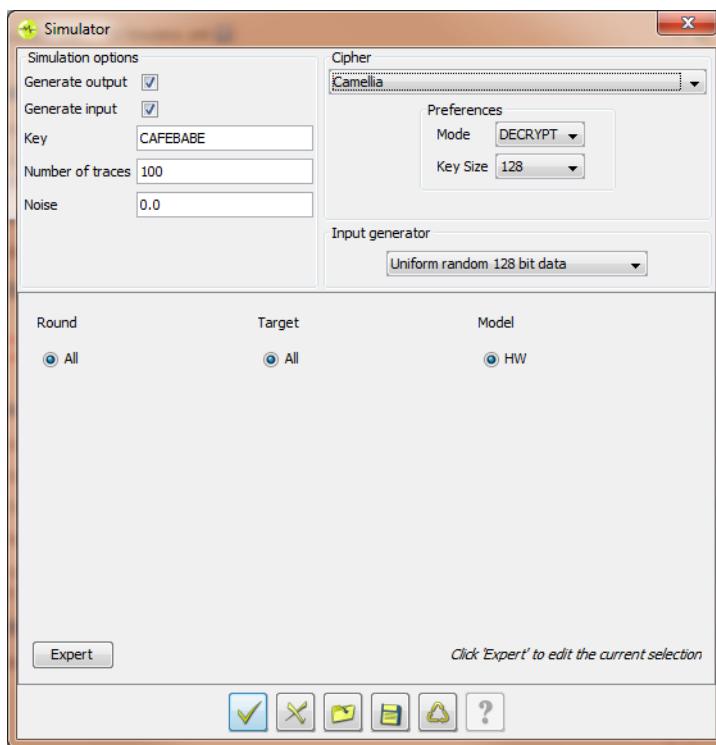
For explanations of, and attacks on, specific ciphers, please refer to Appendix K, *Cipher Suites*.

For explanations of functionality shared between multiple modules, please refer to Chapter 3, *Cryptography*.

Purpose

Simulate simulation traces of a supported crypto algorithm by leaking all its selected leakage models.

Dialog



Inputs

The *Generate output* selection field specifies if output data should be stored in the result trace set.

The *Generate input* selection field specifies if input data should be stored in the result trace set.

The *Number of traces* edit field specifies the number of traces that will be generated by the simulator.

The *Noise* edit field specifies the standard deviation of the noise that will be added to the result traces.

The *Generator* panel can be used to select the input data generator that will be used by the simulator. Optionally, input generator specific options can be configured.

Results

A trace set is returned that has simulated values for every selected leakage.

E.7.6 Template Analysis

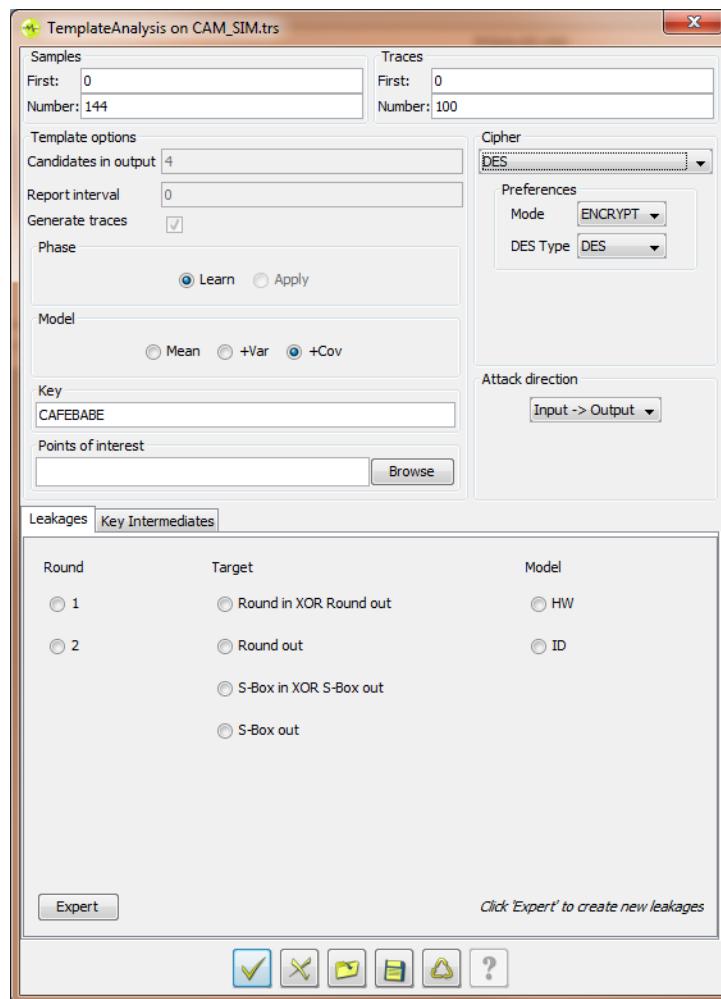
For explanations of, and attacks on, specific ciphers, please refer to Appendix K, *Cipher Suites*.

For explanations of functionality shared between multiple modules, please refer to Chapter 3, *Cryptography*.

Purpose

Perform a differential first order template attack as described in [1] to the trace set.

Dialog



Inputs

- *Phase: learn.* This phase learns the patterns in the traces, requiring a correct key to be entered. The patterns are stored in the memory of the module, such that they are used the next time the 'apply' phase is used.
- *Phase: apply.* This phase applies the previously learned patterns to the selected trace set without known key. In this phase the attack model should be selected. The applied trace set should be aligned to the learned trace set, and also have the same number of samples.
- *Model.* Characterize multivariate Gaussian by only mean, mean+variance, or mean +covariance.
- *Points Of Interest.* The locations where activity happens that we interested in analysing.

Results

In the learning phase, the result of the module is a set of mean traces for each leakage model. This allows selection of a model that seems to leak (e.g. based on standard deviation). After the apply phase, the module outputs sub key guessed based on the mean log-likelihood of the templates.

Background

In the learning phase, the profiles of the measurements are learned for each intermediate. It therefore requires a measurement set with a known key. This is done by characterizing each template as a multivariate Gaussian distribution. Theoretically, a multivariate Gaussian model is one of the strongest attacker models. This model is selected by selecting 'Cov', which calculates the full covariance matrix. However, this is often infeasible in terms of time/memory. To speed up the process, 'Var' can be selected, which only calculates individual samples' variance and mean. For further speedup, 'mean' can be selected, which only calculates the mean of the distribution. The latter two options gain computational efficiency at the cost of attack strength.

After the learning phase, the module outputs the means of the learned Gaussian distributions (not the variance or covariance). This is done for each leakage model, such that the user at this point can select what leakage model is most appropriate (e.g. HW sbox out vs HD sbox in/out).

Also, a user can at this point decide to reduce the number of samples to include in the template analysis to only points in time with high variance in the templates. This should be done by generating a trace set with only these samples, and reapplying the template leaning phase.

In the application phase, the learned models can be applied to a similar trace set with an unknown key. The trace set to be analysed should have exactly the same number of samples, and should be acquired and processed using the same steps. Important is also to align this trace set to the one on which the templates are learned.

After the application the module outputs the results as the mean log-likelihood. This value is derived from the probability of the Gaussian model, with the highest value being the most likely template, and thus the most likely key.

References

[1]: Elisabeth Oswald and Stefan Mangard, "Template Attacks on Masking - Resistance Is Futile", CT-RSA 2007, LNCS 4377, pp. 243–256, 2007. c Springer-Verlag Berlin Heidelberg 2007

E.7.7 Verify

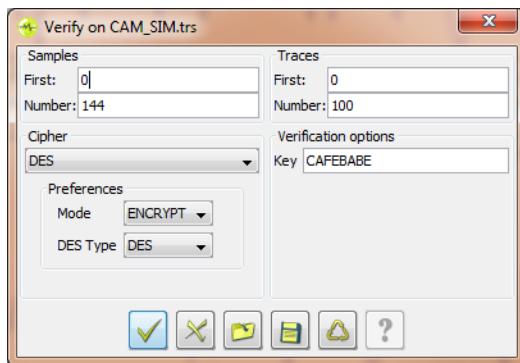
For explanations of, and attacks on, specific ciphers, please refer to Appendix K, *Cipher Suites*.

For explanations of functionality shared between multiple modules, please refer to Chapter 3, *Cryptography*.

Purpose

Verify input/output data of traces are correct given a key and algorithm

Dialog



Inputs

None, other than the ones already detailed in Chapter 3, *Cryptography*.

Results

The module prints how many input/output pairs verify correctly.

E.7.8 Known Key Analysis

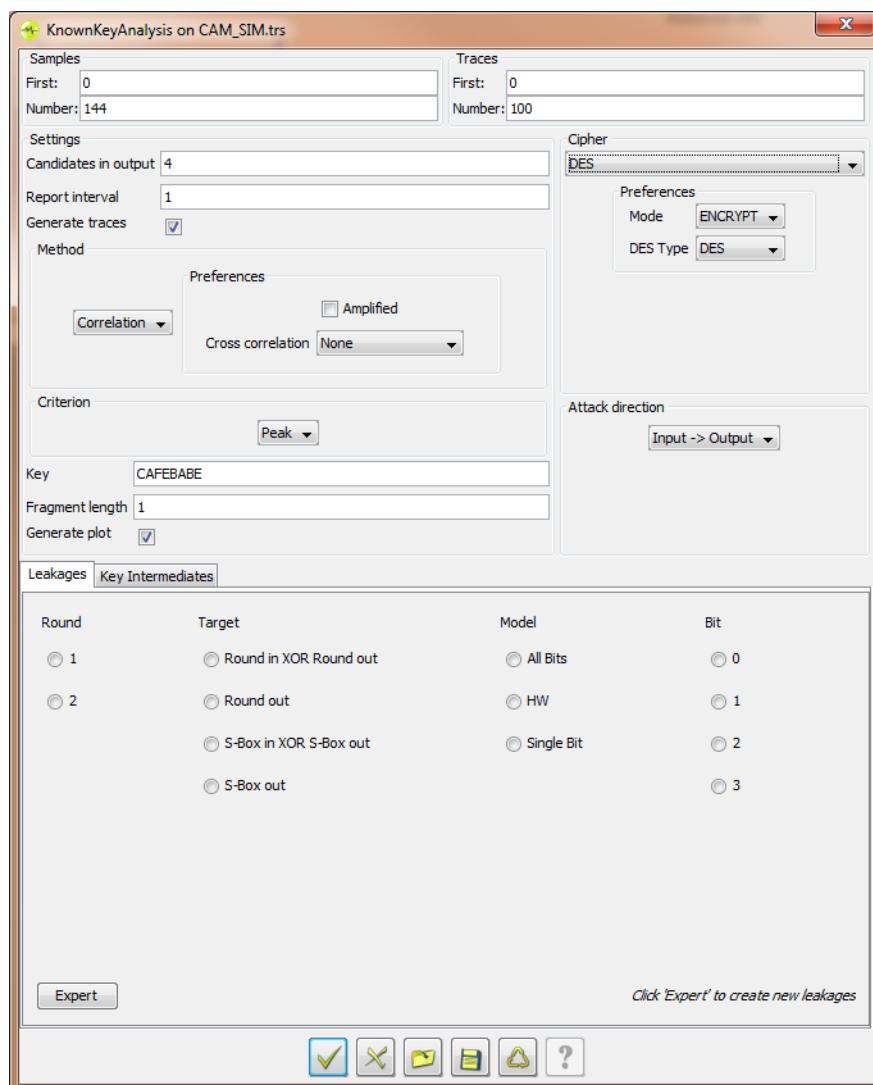
For explanations of, and attacks on, specific ciphers, please refer to Appendix K, *Cipher Suites*.

For explanations of functionality shared between multiple modules, please refer to Chapter 3, *Cryptography*.

Purpose

Find the strongest locations of leakage.

Dialog



Inputs

- *Fragment length*. This is the number of samples that are combined into a single fragment.
- *Generate plot*. Whether to generate a known key plot.

Results

The strength of the leakage related to every key byte at each datapoint of the input trace and/or a plot of this data.

E.7.9 AES Chosen Input Known Key Analysis

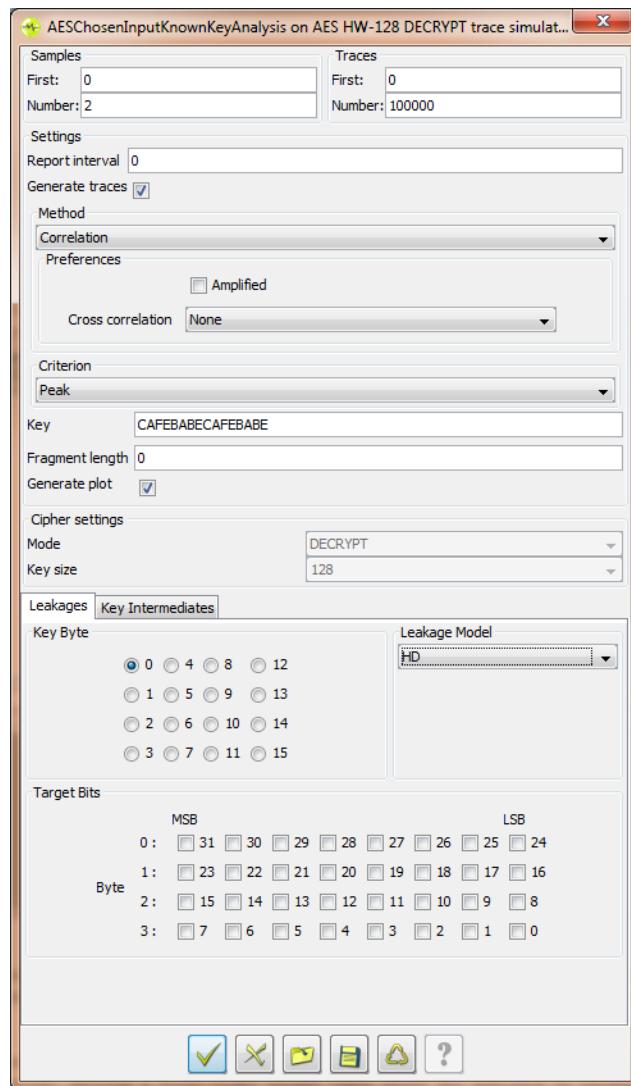
For explanations of, and attacks on, specific ciphers, please refer to Appendix K, *Cipher Suites*.

For explanations of functionality shared between multiple modules, please refer to Chapter 3, *Cryptography*.

Purpose

See Section E.7.8, “Known Key Analysis” and Section E.7.3, “AES Chosen Input First Order Analysis”

Dialog



Inputs

See Section E.7.8, “Known Key Analysis” and Section E.7.3, “AES Chosen Input First Order Analysis”

Results

See Section E.7.8, “Known Key Analysis”

E.7.10 Points Of Interest Selection

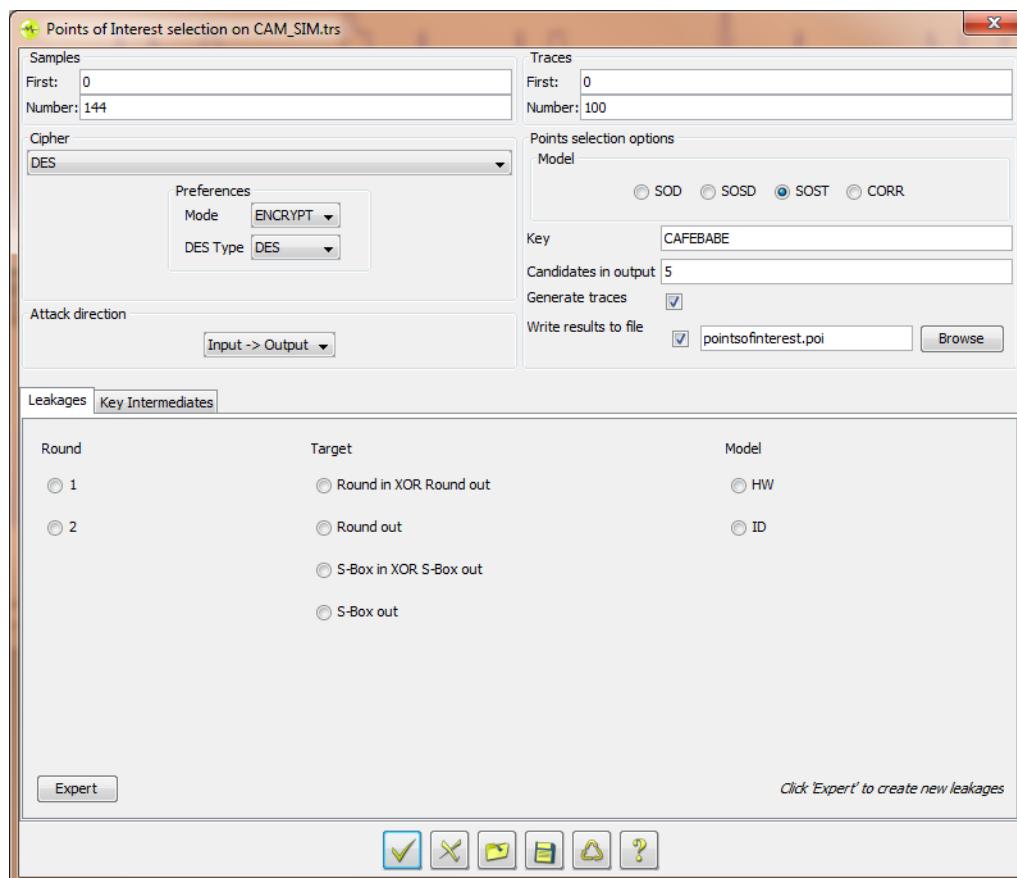
For explanations of, and attacks on, specific ciphers, please refer to Appendix K, *Cipher Suites*.

For explanations of functionality shared between multiple modules, please refer to Chapter 3, *Cryptography*.

Purpose

This module is used to select samples in traces that are likely to give the best result for side-channel analysis and in particular Template attacks. These samples are typically referred to as *points of interest* (POI). The module selects POI by analyzing the traces based on statistical tests over the leakage models input by the user, and choosing samples in traces that give the best statistical test results. POI selection requires knowledge of the key used for the input traces.

Dialog



Inputs

- *Model*. Statistical tests used to analyze the traces.
- *SOD*. Sum of pair-wise difference of averaged signals. This test first calculates the value of the targeted leakage for every trace and partitions the traces into several groups such that all traces that correspond to the same leakage value are grouped together. This

partitioning is done for every given leakage model. The test then computes the average of the traces in each partition, computes the difference of every pair of the averaged traces, and sums the results. See [TemplateAttacks] for details.

- **SOSD**. Sum of squared pair-wise difference of averaged signals. This test is identical to SOD, except that in addition the differences of the averaged traces are squared before they are summed. See [TemplateVSStochastic] for details.
- **SOST**. Sum of squared pair-wise t-difference. This test is similar to SOSD. The difference is that this test also takes into account the variances of the traces in each partitions. In detail, the difference of the averages from a pair of partitions is further divided by the square root of the sum of the averaged variances from these partitions. See [TemplateVSStochastic] for details.
- **CORR**. Correlation coefficient. The test returns the correlation coefficients between the traces and the leakage values.
- *Candidates in output*. Number of POIs to select.
- *Write result*. True if to save the selected POI result to a file, which can be imported by other modules, e.g. by the TemplateAnalysis module.

Results

This module returns both traces and textual outputs. The traces plot the result of the statistical test. For each leakage model, one of such traces is generated. The textual output prints the POIs selected for each leakage model and their statistical test results. The POIs are sorted in the order of the input leakage models and in the descending order of their test results for each leakage model. The selected POIs (excluding the statistical test results) are also saved to a .poi file in the CVS style if the 'Write result' option is enabled.

References

[TemplateAttacks] Suresh Chari. Josyula R. Rao. Pankaj Rohatgi. *Template Attacks*. 13-28. Copyright © 2003 Springer-Verlag Berlin Heidelberg. 0302-9743. Springer Berlin Heidelberg. CHES. August 13-15, 2002. Redwood Shores, CA, USA. .

[TemplateVSStochastic] Benedikt Gierlichs. Kerstin Lemke-Rust. Christof Paar. *Templates vs. Stochastic Methods*. 15-29. Copyright © 2006 Springer-Verlag Berlin Heidelberg. 3-540-46559-6. Springer Berlin Heidelberg. CHES. October 10-13. Yokohama, Japan. .

E.8 Edit

This section contains the descriptions for all the modules concerning editing trace sets.

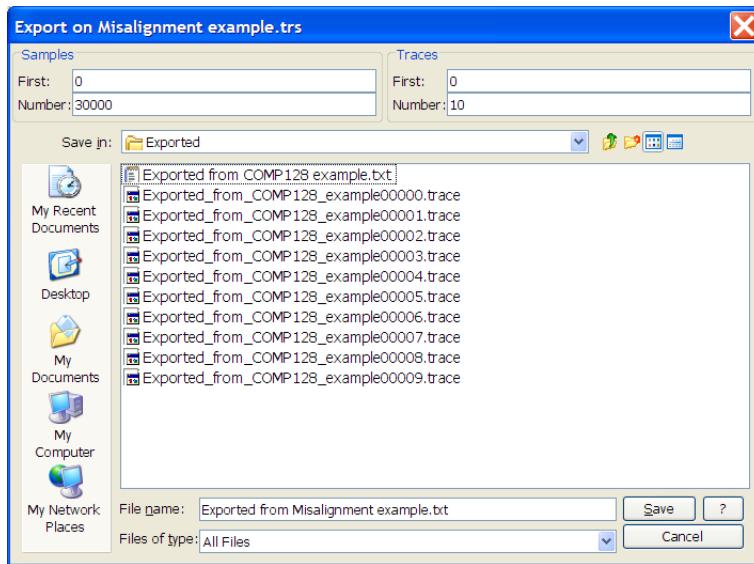
E.8.1 CriExport

Version: SCA + FI

Purpose

To export a trace set to a list file and traces according to the format used by the CRI tool, DPAWS.

Dialog



Result

The module produces a list file in text format and separate binary files for the traces.

The list file contains for each trace a text line with two tab-separated items:

- The file name of the trace
- The crypto data in hex format enclosed in brackets []

The trace file names are automatically derived from the trace set name and appended with five digit numbers.

The trace files contain a 24 byte header followed by the samples. The header is coded little endian with the following fields:

- Version, short (2 bytes), value: 0x2F01
- Sample frequency, float (4 bytes)
- Acquisition input voltage range, int (4 bytes), value: 1000
- Acquisition offset, float (4 bytes), value: 0
- Sample coding size, byte, value: either 8 (for 8 bit samples) or 16 (for 16 bit samples)
- Time offset, int (4 bytes), value: 0
- Padding, 5 bytes, value: 0

The samples themselves are coded as byte (8 bit) or short (16 bit, little endian).

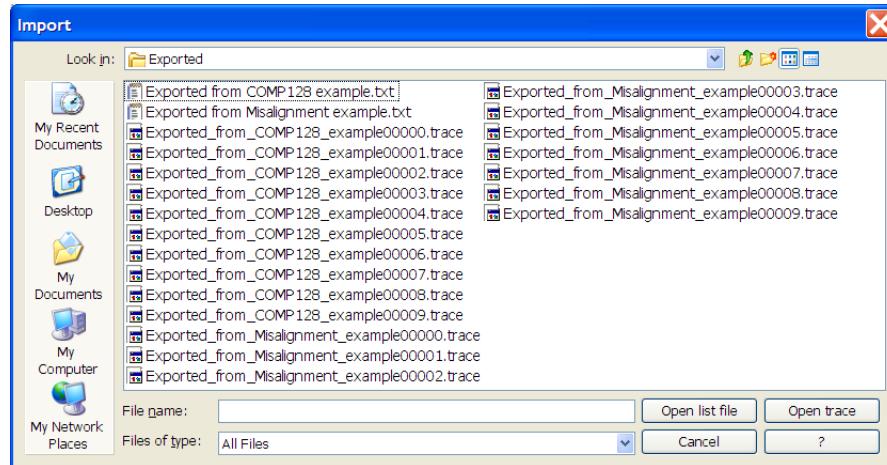
E.8.2 Crilimport

Version: SCA + FI

Purpose

To import a trace or trace set files produced by the CRI tool, DPAWS.

Dialog



Input

The analyst can select either a binary file (produced by DPAWS) and click the 'Open trace' button, or select a list file and click the 'Open list file' button to open a trace set. The module produces the imported traces after checking the format of the traces and list files.

The list file is expected to contain for each trace a text line with tab-separated items, including at least the corresponding trace file name. The line may further include crypto data in hex format enclosed in brackets [].

The trace files are expected to contain a 24 byte header followed by the samples. The header is coded little endian with the following fields:

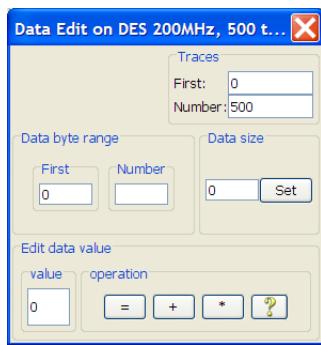
- Version, short (2 bytes), value: 0x2F01
- Sample frequency, float (4 bytes)
- Acquisition input voltage range, int (4 bytes)
- Acquisition offset, float (4 bytes)
- Sample coding size, byte, value: either 8 (for 8 bit samples) or 16 (for 16 bit samples)
- Time offset, int (4 bytes)
- Padding, 5 bytes, value: 0

The samples themselves are coded as byte (8 bit) or short (16 bit, little endian).

E.8.3 DataEdit

Version: SCA + FI**Purpose**

To manipulate the data stored along with a trace set.

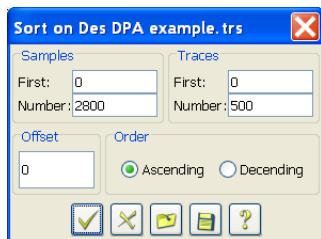
Dialog**Input**

The *Data byte range* panel is used to set the range of bytes participating in the edit operation. The *Data size* panel is used to extend or shrink the size of the data array.

The *Edit data value* panel allows values of individual data bytes within the selected scope to be set. Data bytes can be set to the value set in the *Value* edit field by clicking the '=' button. In addition the selected value can be added to, or multiplied by the selected samples by clicking the '+' or '*' buttons. The latter functions can for instance be used to change the offset or scale of an entire trace set.

E.8.4 DataSort**Version: SCA + FI****Purpose**

To sort traces with respect to the included data.

Dialog

Input

The *Offset* panel is used to set the offset in the array of data bytes for the sorting operation. The *Order* panel is used select an ascending or descending sort order.

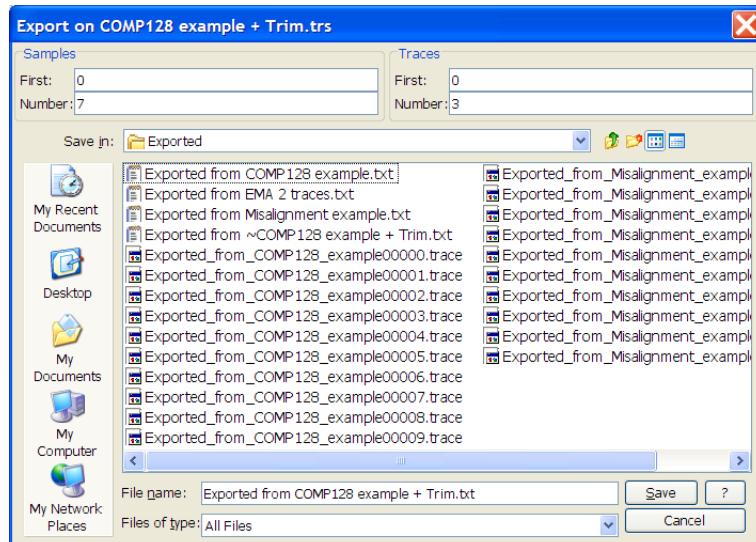
E.8.5 Export

Version: SCA +FI

Purpose

To export a trace set to a text file in comma-separated values format.

Dialog

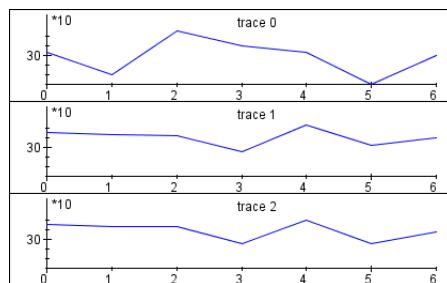


Background

This is a format that can be used by e.g. MatLab. The module produces a single text file where one line is generated for each sample. Each line contains a comma-separated list of floats. The first value represents the x value, while all other values represent the y values (sample values) of the respective traces. **Important note:** As all traces are transposed in memory this module is rather limited in trace and trace set length.

Example

Consider the three small traces in the figure below:



After export the resulting text files contains the following lines:

```
0.0,      304.0,   316.0,   316.0
2.8E-7,   280.0,   314.0,   314.0
5.6E-7,   326.0,   312.0,   314.0
8.4E-7,   310.0,   296.0,   296.0
1.12E-6,  304.0,   324.0,   320.0
1.4E-6,   270.0,   302.0,   296.0
1.68E-6,  300.0,   310.0,   308.0
```

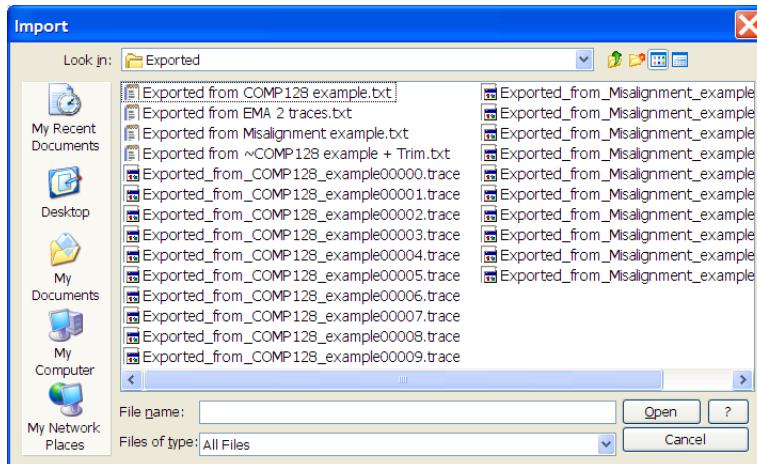
E.8.6 Import

Version: SCA + FI

Purpose

To import a trace set from a text file in comma-separated values format.

Dialog



Background

This is a format that can be used by e.g. MatLab. The module reads from text files where each line represents one sample for all traces. Each input line should contain a comma-separated list of floats. The first value represents the x value, while all other values represent the y values (sample values) of the respective traces. As all traces are transposed in memory this module is rather limited in trace and trace set length.

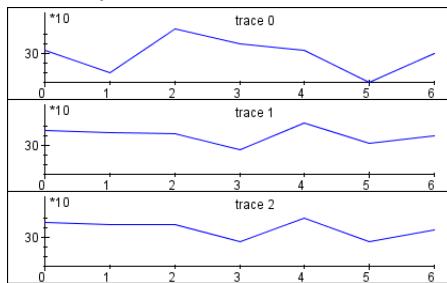
Example

Consider a text file with the following values:

```
0.0,      304.0,   316.0,   316.0
```

2.8E-7, 280.0, 314.0, 314.0
 5.6E-7, 326.0, 312.0, 314.0
 8.4E-7, 310.0, 296.0, 296.0
 1.12E-6, 304.0, 324.0, 320.0
 1.4E-6, 270.0, 302.0, 296.0
 1.68E-6, 300.0, 310.0, 308.0

After import these lines result in the following three traces:



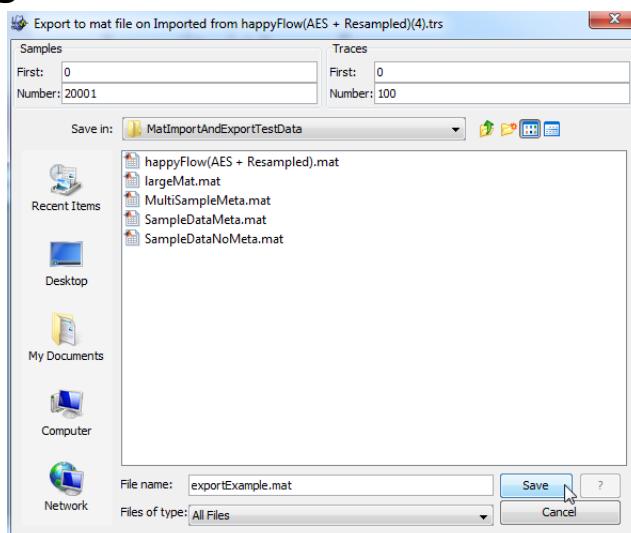
E.8.7 MatExport

Version: SCA +FI

Purpose

To export a trace set to a .mat file in Version 5 MAT-file format.

Dialog



Description and Design

The exported file format is complaint with Level 5 MAT-files. Level 5 MAT-files are compatible with MATLAB Versions 5 and up.

Level 5 MAT-files are made up of a 128-byte header followed by one or more data elements. The data elements are storing named matrices of different primitive types, which are called variables.

For performance reasons, we store multiple traces per variable. Each group of traces is called a block. Let "d" be the number of samples per trace and b the number of traces per block. Inspector export data elements as follows:

| Variable name | Description |
|-----------------|---|
| <metadata name> | Each element of metadata that is stored per TraceSet is stored under "metadata_" followed with its Java field name in the TraceSet class. Element type in file is set as close as possible to Java types. |
| sampleBlock%d | Block of trace samples start with index 0. sampleBlock 0 contains sample [0...b-1], sampleBlock 1 contains sample [b...2b-1], etc. The size of this variable is therefore: b (rows) x d (columns). Each sample is an array of single precision float. |
| dataBlock%d | Block of trace data, indexing and layout same as for sampleBlocks. Difference is that each element is an 8-bit signed integer. Not stored if data length is 0. |
| titleBlock%d | Block of trace titles, indexing and layout same as for sampleBlocks. Difference is that data type is char. Not stored if title length is 0. |

The ordering of these data elements in the file is {all metadata}, {sample|data|title}Block0, {sample|data|title}Block1, {sample|data|title}Block2, etc.

The block size is decided by the sample size. The predefined block size is $32 * 1024 * 1024$ bytes. Given the sample(float[]) size, there are at most $1024 * 1024$ elements in each block except when one sample is larger than the block size. In this case, we store one sample per block.

Below is an example of loading a .mat file exported by Inspector. In this example, the traceset being exported contains only sample and data:

| Name | Value |
|----------------------|------------------|
| metadata_GlobalTitle | " |
| metadata_Description | " |
| metadata_xLabel | 'sec' |
| metadata_yLabel | 'Volt' |
| metadata_xScale | 2.4999e-07 |
| metadata_yScale | 0.0394 |
| metadata_Offset | 0 |
| sampleBlock0 | <52x2001 single> |
| dataBlock0 | <52x32 int8> |
| sampleBlock1 | <48x2001 single> |
| dataBlock1 | <48x32 int8> |

Limitations

By design, the Inspector trace set is exported with the predefined data structure and the specific data type (e.g. single for samples, int8 for data, etc.).

For data analysis, user may need to manipulate data in various ways and save them. To achieve a two way interaction with Inspector, the changes made to the exported file must be saved in a format complaint with Level 5 MAT-files and also to the predefined data structure as well as sequences.

As all traces are transposed in memory, this module is limited in number of samples per trace and number of traces per trace set.

Current traceset metadata and type are restricted as: metadata_GlobalTitle(char, nullable), metadata_Description(char, nullable), metadata_XLabel(char, NOT Null), metadata_YLabel(char, NOT Null), metadata_XScale(single, NOT Null), metadata_YScale(single, NOT Null), metadata_XOffset(single, NOT Null)

Examples

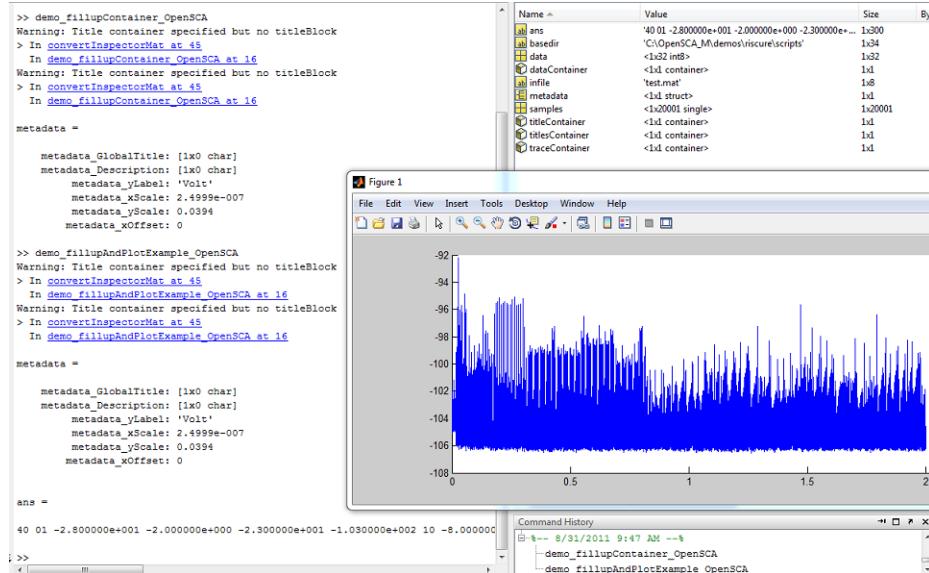
The exported file from Inspector is no different from any other Level 5 MAT-files compliant file. In this section, we first demonstrate how the exported file can be used in Matlab with OpenSCA, a Matlab tool box. Secondly, we chose Octave as a representative for software who support Level 5 MAT-files compliant files.

Using OpenSCA toolbox in Matlab

For OpenSCA tests, the procedure we applied:

1. Manually export a traceset from Inspector and save it as test.mat
2. Execute test script demo_fillupContainer_OpenSCA.m to load data from test.mat and store traceinfo in containers(OpenSCA data folder); then display metadata info.
3. Execute test script demo_fillupAndPlotExample_OpenSCA.m to load data from test.mat and store traceinfo in containers(OpenSCA data folder); plot the first sample from the container.

Matlab can load test.mat sucessfully and plot the first sample from SCA containers. We obtain the result from scripts as the following screenshot:



The resources discussed in this section can be found in </inspector installation folder>\doc\manual\samples\openSCA\

NOTE :The test scripts are constructed for rough demonstration of the feasibility, so there are issues that data in containers are accumulating instead of overwriting when read the

same .mat file. And when executing demo_fillupAndPlotExample_OpenSCA.m 1st time, one may encounter errors that “No elements have been added to the container object yet!”. When executing demo_fillupAndPlotExample_OpenSCA.m 2nd or more times, “plot” will work.

Level 5 MAT-files compliant software example

Inspector MatExport Module will export trace set to a Version 5 MAT-file format, which means that any software whose complaint with Version 5 MAT-file format will be able to interact with Inspector indirectly through the exported .mat file. For instance, GNU Octave can interact with Version 5 MAT-files. Our experience shows that Octave can interact with Inspector exported .mat files as expected.

Below is an example of a sequence of commands in Octave to interact with an Inspector export, test.mat: clear workspace, load test.mat exported by Inspector, display data being loaded from the test.mat, and display first 10 elements in the frist row of sampleBlock0. And in this example, the trace set being exported contains only sample and data. Besides the command "whos" will display block names in alphabetical order:

```
octave-3.2.4.exe:30> clear
octave-3.2.4.exe:31> load test.mat
octave-3.2.4.exe:32> whos
Variables in the current scope:
  Attr Name           Size            Bytes  Class
  ==== ====== ====== ====== =====
  dataBlock0          52x32          1664  int8
  dataBlock1          48x32          1536  int8
  metadata_Description 1x0             0  char
  metadata_GlobalTitle 1x0             0  char
  metadata_xlabel     1x3             3  char
  metadata_ylabel     1x1             8  double
  metadata_xScale    1x1             8  double
  metadata_yLabel    1x4             4  char
  metadata_yScale    1x1             8  double
  sampleBlock0        52x20001       8320416 double
  sampleBlock1        48x20001       7680384 double

Total is 2003310 elements using 16004031 bytes
octave-3.2.4.exe:33> sampleBlock0(1,:)
ans =
Columns 1 through 8:
 -102.70 -102.82 -105.72 -105.82 -102.92 -102.54 -102.50 -105.70
Columns 9 and 10:
 -105.74 -102.36
octave-3.2.4.exe:34>
```

When use the same Matlab scripts to use OpenSCA in Octave, we observed that Octave has many differences in API and programming approaches comparing with Matlab. For this reason, OpenSCA needs to be adapted when used with Octave.

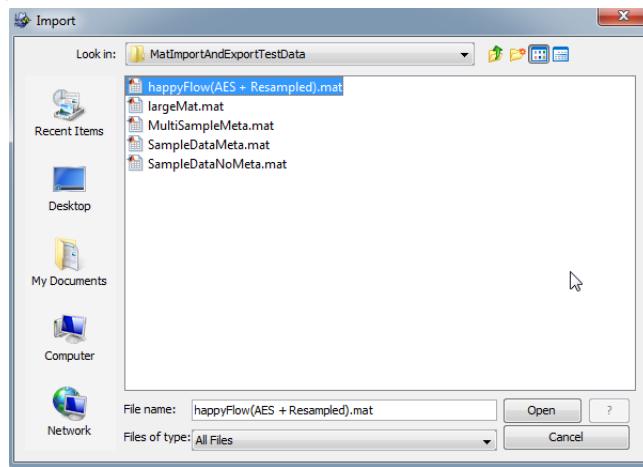
E.8.8 MatImport

Version: SCA + FI

Purpose

To import trace set from a Version 5 Mat-file complaint .mat file.

Dialog



Description

In general, the .mat file to be imported must be a Version 5 Mat-file complaint file. The trace set must be stored following a predefined structure.(Please refer to MatExport section for more info)

The file can be obtained from the Matlab or the Inspector Export module. Thus, in this .mat file, after level 5 Mat-file header, the rest data is structured as follows:

1. trace set metadata blocks: the trace set metadata, e.g. xlabel, ylabel, etc. have been stored right after header
2. trace set sample|data|title blocks: sample, data(if any) and title(if any) are grouped separately however equally and stored into blocks, starting with SampleBlock0 (dataBlock0 and titleBlock0, if any). Equally means that the blocks in the same level, e.g. SampleBlock0, dataBlock0 and titleBlock0, contains equal amount of sample|data|title.

The block size has been defined from MatExport Module as 32*1024*1024 bytes. For performance reasons, we strongly suggest the user to divide samples, as well as data and titles (if any), into blocks with no more than 1024*1024 elements. The data type needs to be set strictly as predefined. (Please refer to Section E.8.7, "MatExport" for more info)



Memory limitations

The maximum size of an input file is limited by the amount of memory available to Inspector. On 32 bit architectures, available memory will be hundreds of MBytes, while for 64bit versions it could be in excess of the maximum file size of 2Gbytes. Consider using Inspector 64bit when conversion of large .mat files is needed.

E.8.9 Recover

Version: SCA + FI

Purpose

To repair a corrupted trace set.

The Recover module can be used when a trace set got corrupted. This may occur when Inspector is unexpectedly terminated (e.g. by a system reboot) during acquisition, or by an accidental overwrite of part of the file.

The module analyses a trace set file and verifies its integrity. This is done by comparing the actual length with the parameters found in the header. If the file is ok, no changes will be made. If the NT (Number of Traces) tag is missing the file is not recognized as a trace set and no changes will be made.

Otherwise the module will first try to correct NT to approach the length of the file. Next it will correct the actual length to align this with the length computed from the header parameters.

E.8.10 Reverse

Version: SCA + FI

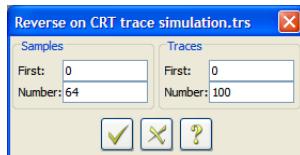
Purpose

To reverse all samples over the time axis.

Inputs

Any trace to reverse.

Dialog



Related tutorials

- **ECC SPA Tutorial** [[..../tutorials/sectionsECCSPATutorial.html](#)][PDF] [[..../tutorials/Tutorials.pdf#sectionsECCSPATutorial](#)]

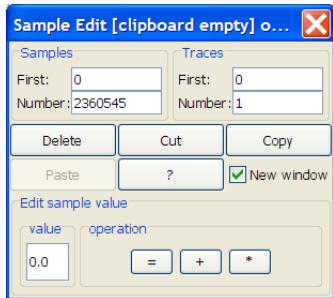
E.8.11 SampleEdit

Version: SCA + FI

Purpose

To manipulate the size and contents of a trace set. This module is useful if several trace sets need to be merged, or if some samples of the traces must be set to specific values.

Dialog



Input

The Samples and Traces panels are used to set the scope of the next edit operation.

The following functions are available in the button panel in the centre of the window:

- **Delete** Deletes the selected samples or traces. As every trace in a trace set must have the same length this operation is possible only if all samples in some traces or some samples in all traces are selected.
- **Cut** Cuts the selected samples and traces from the trace set and puts them in the system clipboard. As every trace in a trace set must have the same length this operation is possible only if all samples in some traces or some samples in all traces are selected.
- **Copy** Copies the selected samples and traces from the trace set and puts them in the clipboard. No result trace set is produced as there is no change to the current trace set. Also there is no limitation to the scope, e.g. any combination of samples and traces is allowed. The next time the Edit module is invoked on a trace set, the clipboard dimensions are indicated in the title bar of the form, and the clipboard data can be pasted.
- **Paste-** Adds the clipboard data to the trace set before the first selected sample / trace. As every trace in a trace set must have the same length, this operation is only possible if the number of samples or the number of traces in the clipboard equals the number of samples or the number of traces in the current trace set. In the former case the clipboard data is inserted as new traces before the first selected trace. In the latter case the clipboard data is inserted as new samples in each trace, before the first selected sample.
- **Paste+** Adds the clipboard data to the trace set after the last selected sample / trace. As every trace in a trace set must have the same length this operation is possible only if the number of samples or the number of traces in the clipboard equals the number of samples or the number of traces in the current trace set. In the former case the clipboard data is appended as new traces after the last selected trace. In the latter case the clipboard data is appended as new samples in each trace, after the last selected sample.

The *Edit samplevalue* panel allows for setting values of individual samples within the selected scope. Samples can be set to the value set in the *Value* edit field by clicking the '=' button. Further the chosen value can be added, or multiplied with the selected samples by clicking the '+' or '*' buttons. The latter functions can for instance be used to change the offset or scale of an entire trace set.

The *New window* check box determines whether results of this edit operation are to be shown in a new window, or that the existing window is replaced with the results.

Related tutorials

- **ECC SPA Tutorial** [\[../tutorials/sectionsECCSPATutorial.html\]](#) [\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsECCSPATutorial\]](#)

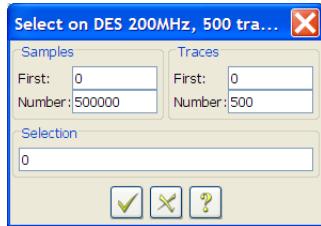
E.8.12 Select

Version: SCA + FI

Purpose

To extract an arbitrary subset of traces from a trace set.

Dialog



Inputs

The select edit field defines the selection as a comma-separated list of traces or ranges. A range is indicated by two numbers separated by a minus sign. For example, a selection 3,5-7,9 will produce traces 3, 5, 6, 7 and 9.

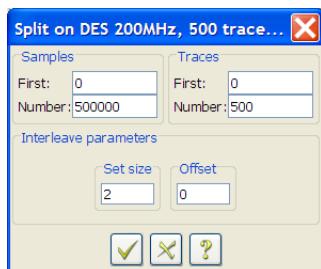
E.8.13 Split

Version: SCA + FI

Purpose

To split a trace set of interleaved traces. The input could for instance be a trace set containing alternating traces from different sources (e.g. different oscilloscope channels), or a trace set with the results of different signal operations.

Dialog



Input

The *Samples* and *Traces* panels are used to set the scope of the next edit operation.

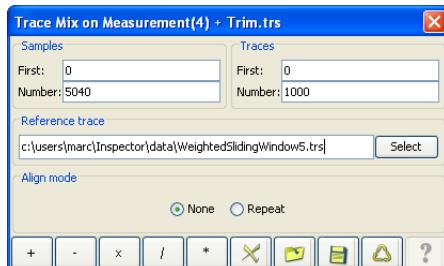
The *Interleave parameters* panel is used to select the set size; e.g. the number of interleaved sources, and the offset; e.g. the index within the set of sources. Suppose the trace set contains traces from three groups a, b, and c, then the input traces are a0, b0, c0, a1, b1, c1, a2, b2, c2, etc. When set size is set to 3, and offset is set to 2, the module produces c0, c1, c2, etc.

E.8.14 TraceMix

Purpose

To mix a selection of traces with a single reference trace. The mix function can be either 'Add', 'Subtract', 'Multiply', 'Divide', 'Convolute', or 'Correlate'.

Dialog



Input

The TraceMix dialog requires selection of a file with a reference trace. When the reference trace is longer than the input traces, the reference excess samples are simply ignored. However, when the reference trace is shorter the effect depends on the align mode. With the 'Align mode' panel the user can choose to either apply no operation to the input samples at offsets exceeding the reference trace length, or to wrap the reference trace and apply the same operation with repeated reference trace samples again to input samples at offsets exceeding the reference trace length. The user can start the module by pressing the button corresponding to the desired operation.

Background

The TraceMix module can serve different purposes. The subtract operation can be useful when differences between individual traces are to be magnified. In that case the user would first make an average trace, and then use TraceMix to subtract the average from all traces in the trace set.

Multiplication can for instance be useful to benefit from the effect of a specific leakage model, where a chip has a non-constant leakage curve over a clock cycle. By applying this leakage curve to the (correlation) trace it is possible to increase the signal to noise ratio.

Convolution is a mechanism to implement advanced filters. For instance, a moving average with window N is simply implemented by using a trace of N samples all set to 1. For a weighted

moving average, the ref trace values should represent the weight function. More information on convolution can be found in <http://en.wikipedia.org/wiki/Convolution>.

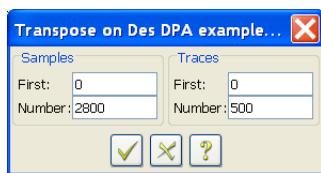
E.8.15 Transpose

Version: SCA + FI

Purpose

To swap samples and traces. E.g. the first result trace will be the concatenation of the first sample of all traces, etc.

Dialog



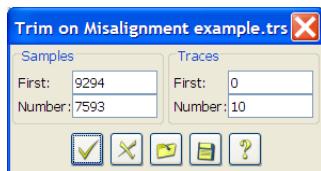
E.8.16 Trim

Version: SCA + FI

Purpose

To trim traces that are longer than needed. The analyst can select the part of interest, and all remaining samples will be left out in the resulting traces.

Dialog



Inputs

The module uses just the regular sample and trace scope values.

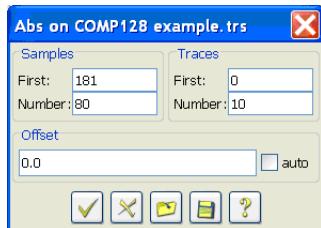
E.9 Filter

This section contains the descriptions for all the filter modules.

E.9.1 Abs

Version: SCA + FI**Purpose**

to rectify an AC signal relative to an offset value.

Dialog**Inputs**

The Samples and Traces panels are used to set the scope of the Abs operation. The auto option which instructs the module to compute the average value and use it as a reference offset value.

Result

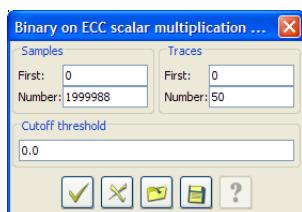
For trace set S : $\text{Abs}_{\text{offset}}(S) = \{ \text{Abs}_{\text{offset}}(s) \mid s \in S \}$. For trace s : $\text{Abs}_{\text{offset}}(s_i) = |s_i - \text{offset}|$

E.9.2 Binary**Version: SCA****Purpose**

To convert traces into a binary representation. The module thresholds every sample in the trace, setting each output sample to 1 if the sample is higher than the threshold and to 0 otherwise.

Inputs

Sample threshold value and trace to convert.

Dialog

Related tutorials

- **ECC SPA Tutorial** [..../tutorials/sectionsECCSPATutorial.html]

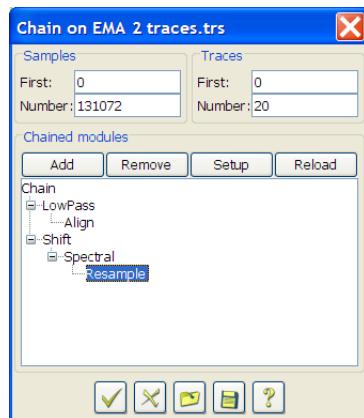
E.9.3 Chain

Version: SCA + FI

Purpose

To perform several filter operations in one step.

Dialog



Inputs

The Samples and Traces panels are used to set the scope of the operation. The Chained modules panel shows the modules that are added to the filter list. The panel contains the following buttons:

- *Add*: Adds a new module to the list.
- *Remove*: Removes a selected (click on module name) module from the list.
- *Setup*: Opens the normal module dialogue to configure the module.
- *Reload*: Reloads the most recent chain.

Result

The modules in the module list will be applied to the input traces in sequential order. Note that the File Open and Save buttons at the bottom of the dialogue can be used to load an earlier chain or to save the current chain. A limitation is that modules in the list have no advance knowledge about the trace set, for example the scale values. This means that attributes normally preset in the invocation dialogue may have to be set manually. Acquisition2 modules (such as ScopeAcquisition) can be inserted in a chain as the first element. Whenever a chain

contains an Acquisition2 module, the input traceset will be ignored, and a new traceset is generated.

E.9.4 FilterGuidance

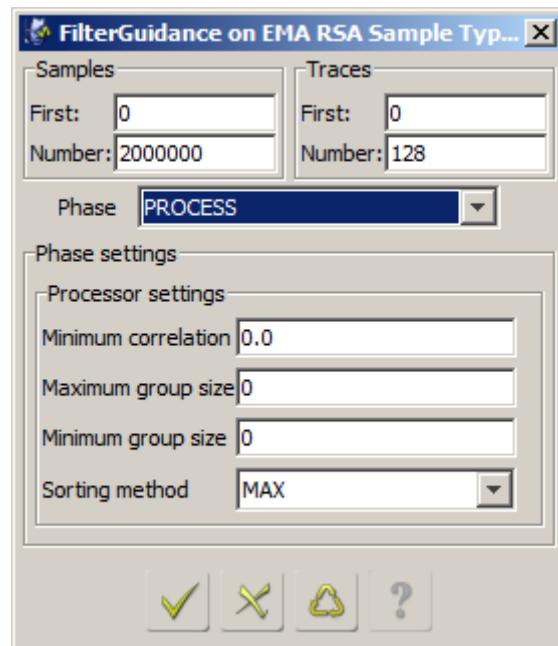
Version: SCA

Purpose

This module gives suggestions on the choice of appropriate frequency filters for side-channel signals. It automatically constructs multi-bandpass filters based on cross-correlation between narrow-band side-channel traces. It should be applied to the output trace of FrequencyBandDecomposition or Spectrogram.

Dialog

Figure E.5. FilterGuidance dialog with PROCESS phase settings



Input

The module takes a set of narrow-band side-channel traces as an input. The *Samples* and *Traces* panels are used to set the scope of the module application.

The module comprises three sequential phases:

- PROCESS: first phase. Analyze the input set of narrow-band side-channel traces and construct groups of bandpass filters.
- SELECT: second phase. Out of the bandpass filter groups constructed by the PROCESS stage, perform band combinations and produce output traces.

- CREATE_FILTER: third phase. Save a bandpass filter group to a file in the XtalClear format.

The phase is selected by the *Phase* drop-down list. Each phase has its own settings that can be configured by controls in the *Phase settings* panel. For the description and explanation of the settings see Background.

Result

The module produces a set of multi-bandpass filters that can be saved in to a file in XtalClear format. Note that the output of the PROCESS phase is silent, and the output of the SELECT phase is a set of traces resulting from grouping the input narrow-band traces using the constructed multi-bandpass filters.

Background

The idea behind the module is to create multi-bandpass filters by grouping frequency bands where the corresponding narrow-band signals exhibit similar behaviour. This can be often seen in the spectrogram. Here the choice of similar bands is automated by cross-correlating the narrow-band traces. The narrow-band side channel traces can be produced by FrequencyBandDecomposition or Spectrogram modules.

PROCESS phase

In the PROCESS phase, a cross-correlation based processor aims at detecting implementations that produce long patterns in time domain, namely, public key algorithm implementations and software symmetric key algorithm implementations.

The processor builds a matrix of correlation coefficients for all pairs of narrow-band filtered traces. Then it sorts the rows of the correlation matrix by the maximum correlation value within a row (diagonal excluded). Finally, it goes through the sorted set of rows and creates a group of passbands from each row, including the bands that lead to a correlation value larger than a certain threshold. If a band is included in a group for some row, it cannot be re-used in subsequent rows. The process is configured by a number of parameters:

- *Minimum correlation*: threshold on the value of the correlation coefficient between the narrow-band traces.
- *Maximum group size*: maximum number of bands in a filter group.
- *Minimum group size*: minimum number of bands in a filter group.
- *Sorting method*: the way groups of filters are sorted

The minimum correlation threshold can be overridden by the value automatically computed by the module if it is larger than the user-provided value. Setting the maximum group size to zero will make the module use the maximum possible size. The sorting method can be MAX or AVG. In the case of MAX, the groups are produced as described above. In the case of AVG, the groups are additionally re-sorted by the average correlation value within a group.

SELECT phase

In this phase, the module will take the groups produced in the previous phase and create one trace for each of these groups. This allows the analyst to inspect the output traces and select a filter configuration which results in interesting patterns.

CREATE FILTER phase

The user selects a group of multiple narrow-band filtered traces (as outputted from the previous phase) to generate a filter file for the XTal Clear filter.

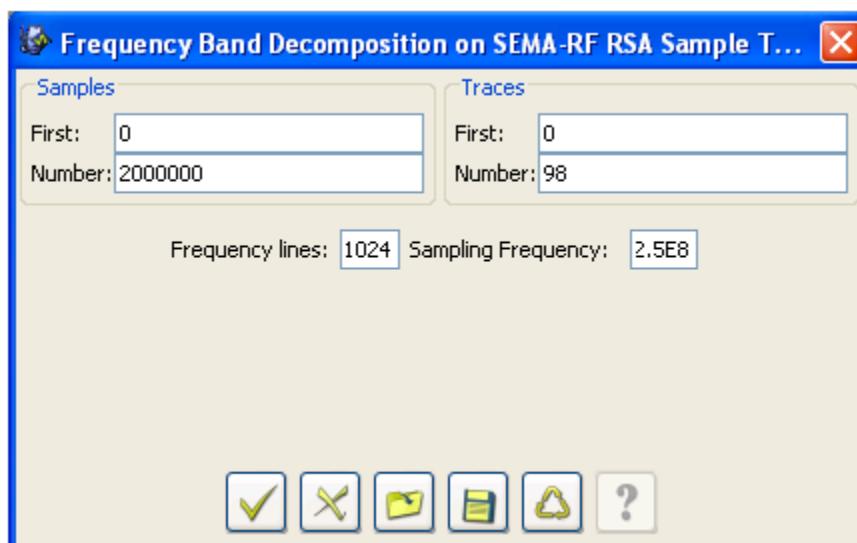
E.9.5 Frequency Band Decomposition

Version: SCA + FI

Purpose

The module first applies a set of band-pass filters on original traces, secondly takes the absolute values, and finally applies a low-pass filter on these traces. The bandwidth of band-pass filter is B and that of low-pass filter is B/2. The slopes of these applied filters are infinitely steep. In the end, the processed trace set will be produced. The new traces present the specified number of frequency components.

Dialog



Input

The Frequency Band Decomposition module takes the following input:

- The *frequency lines* text box specifies the number of frequency components. This value is determined as

$$\text{Frequency Lines} = (f_{\text{Sampling}} / 2) / B \quad (\text{E.1})$$

- The *Sampling Frequency* text box specifies the sampling frequency of the original traces.

Results

The Frequency Band Decomposition module produces a set of new traces. Each new trace contains the same pattern as that in each frequency band of the original traces. In the case of

several traces selected, an average of the output trace is generated so the possible patterns are easier to be detected.

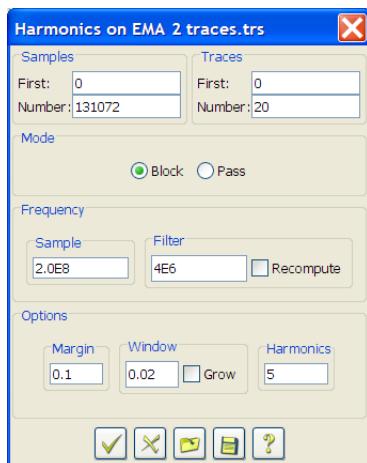
E.9.6 Harmonics

Version: SCA

Purpose

To perform a special form of spectral filtering of a trace. A filter can be configured to facilitate efficient filtering of specific frequencies and their harmonics from a trace set. An analyst can estimate the filter frequency, and the module will centre the filter around the strongest frequency in the neighbourhood of the estimated filter frequency.

Dialog



Input

The *Samples* and *Traces* panels are used to set the scope of the frequency analysis or filtering.

The *Filter Mode* panel defines two options:

- Pass: A frequency pass filter is applied.
- Block: A frequency block filter is applied.

The *Frequency* panel includes two edit fields and a check box:

- Sample: The frequency of the trace set.
- Filter: The (estimated) frequency to be filtered.
- Recompute: Once more compute the actual filter frequency for each input trace.

The *Options* panel defines a number of options:

- Margin: The relative range that the module scans for the strongest frequency to select as filter frequency. A value of 0.1 means that the chosen filter frequency will be between 10%

below and 10% above the estimated frequency. Setting the margin to 0 implies that the actual filter frequency will be exactly the estimated filter frequency.

- Window: the relative window around the filter frequency to filter. A value of 0.02 means that the filtered signal's frequency range is 2% of the filter frequency.
- Grow: If this option is selected, the window grows for the subsequent harmonics (relative window), otherwise the absolute frequency window is applied for the harmonics too.
- Harmonics: the number of harmonics (multiples) to include. A value of 1 means that only the base frequency is filtered.

Background

Please visit the Spectral filter module for more information regarding spectral filtering.

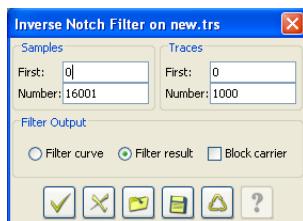
E.9.7 InvNotchFilter

Version: SCA

Purpose

This module inverts the EMA-RF signal after passing the multi-notch filter of the CleanWave. The CleanWave attenuates the RF carrier and 2nd, 3rd and 4th harmonics and amplifies the signal-frequency range in between these harmonics. The frequency characteristic of this filter is not flat. The InvNotchFilter module has an inverted frequency characteristic. The combination of the CleanWave and the InvNotchFilter module gives a flat frequency characteristic while preserving the higher resolution in the analog-to digital conversion process. Additionally the InvNotchFilter has the option to block the RF carrier and its higher harmonics, similar to the Harmonics filter tuned to 13.56 MHz and a window of 2%.

Dialog



Input

The user may select part of a trace by choosing the corresponding samples or may select part of the trace set by selecting the corresponding traces. No other parameter settings are required.

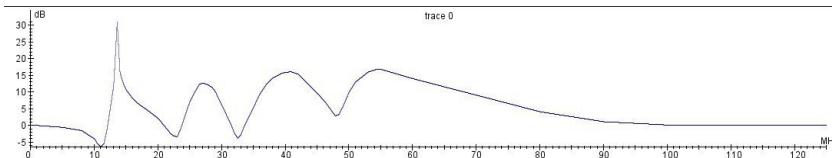
The *Filter Output* panel defines whether only the filter curve is generated, or that it is also applied to the input traces. The former allows the analyst to study the applicability of the filter curve before using it.

By checking the *Block carrier* the RF carrier and its harmonics are blocked, similar to using the module Harmonics with Margin = 0, Window = 2%, no Grow and all harmonics.

Background

The *InvNotchFilter* should be used in combination with the CleanWave for EM analysis on contactless cards (EMA-RF). Please refer to the CleanWave section for EMA-RF for details about the filtering process.

The InvNotchFilter has a fixed frequency characteristic that matches the CleanWave, see figure below.



E.9.8 LowPass

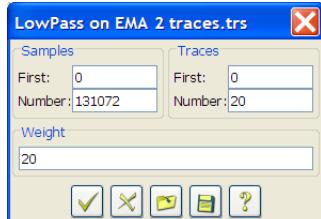
Version: SCA + FI

Purpose

To perform a simple and fast low-pass filter operation. The module applies a fast bidirectional filter to each trace, making each sample a weighted average of the previous sample and the current sample:

$$\text{sample}_i = (\text{weight} * \text{sample}_{i-1} + \text{sample}_i) / (\text{weight}+1)$$

Dialog



Input

The *Samples* and *Traces* panels are used to set the scope of the operation.

The *Weight* panel defines a simple parameter that sets the low-pass characteristic. The best value for weight should be determined experimentally.

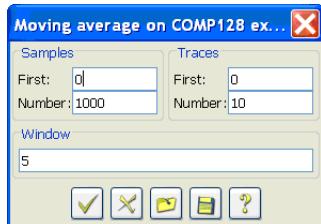
E.9.9 MovingAverage

Version: SCA

Purpose

To average samples within a trace and to reduce noise.

Dialog



Input

The module has a variable window which defines the number of samples to use for averaging. Each output sample is the average of the number of adjacent input samples indicated by the 'Window' value.

Result

For trace set S:

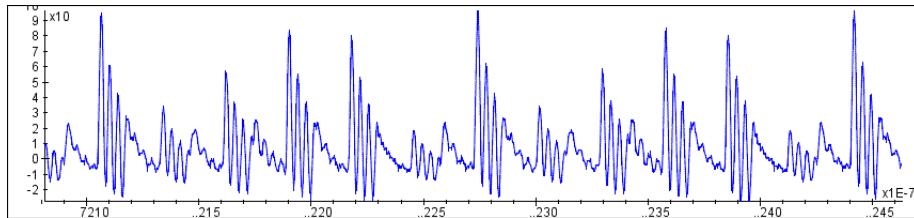
$$\text{MovingAverage}_{\text{window}}(S) = \{ \text{MovingAverage}_{\text{window}}(s) \mid s \in S \}$$

For trace s:

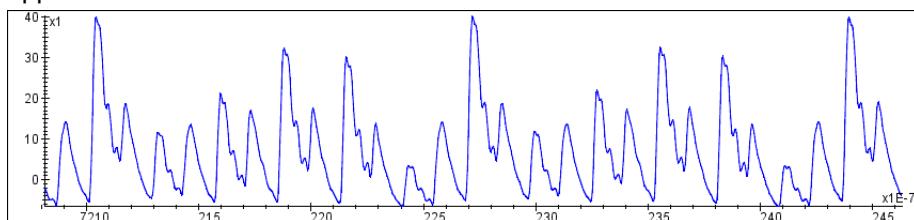
$$\text{MovingAverage}_{\text{window}}(s_i) = (s_{i-\text{window}+1} + \dots + s_i)/\text{window}$$

Example

The figure below shows a trace disturbed by a severe oscillation.



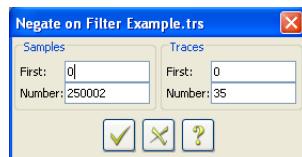
After application of the module with an appropriate value for the window variable, the signal appears in a more subdued form.



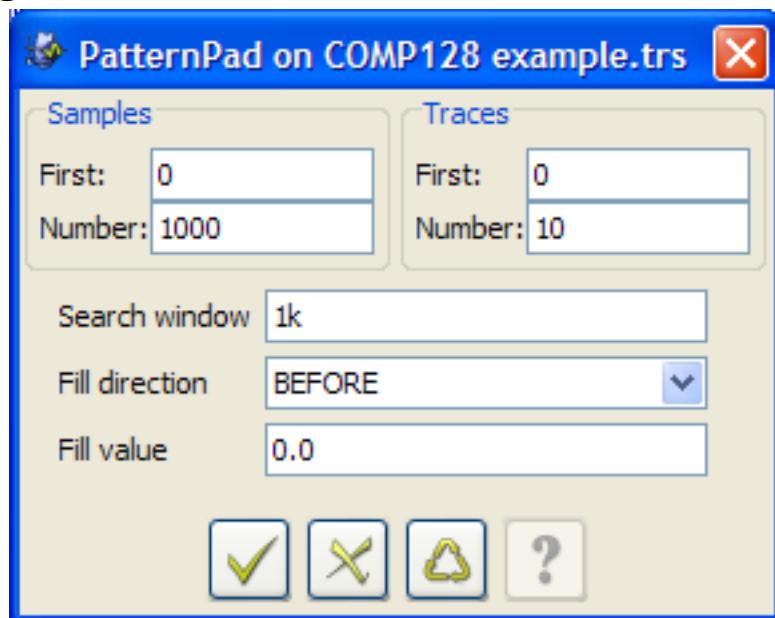
E.9.10 Negate

Version: SCA**Purpose**

To mirror traces over the x-axis

Dialog**E.9.11 PatternPad****Version: SCA****Purpose**

PatternPad is used to pad a trace with a constant value either before or after a pattern match. This is useful as preprocessing step for Elastic Alignment, as the first patterns and/or last patterns in a trace are alignable.

Dialog**Inputs**

The PatternPad module requires the following input values:

- A *reference pattern* is defined by selecting it in the input trace set before loading the module, or by entering the pattern first sample and number of samples in the *Samples* scope panel.
- A *search window* defines how far (in samples) from the original reference pattern the matched patterns are to be found. This allows limiting the scope of the pattern matching, avoiding matches similar to the reference in other parts of the trace.
- A *fill value*, which will be used to pad the trace.
- The *direction* in which the padding will be applied. There are two options for this variable: padding before the pattern match or padding after it.

Usage

- Determine a fill value, e.g. the minimum sample value.
- Select a common pattern before the area to align, and determine the number of samples timing variation there is for this pattern between the traces.
- Run the module, selecting the 'Before' option.
- Go to the first step and perform the same with a pattern at the end of the trace ('After' option)
- Afterward, run Elastic Alignment

Result

The result after the above steps is a trace set starting with a constant value, then a common pattern, the rest of the trace, another common pattern, and ending with a constant value. The constant values are stretched and compressed to the same length by Elastic Alignment, making it more easy to align the other patterns.

Background

Elastic Alignment presumes the first and last sample of a trace are aligned. In practice, this is not always the case. In order to make sure this condition is satisfied, it is possible to pad traces with a constant value, up to a point where they have a similar pattern. As preprocessing step, it is advised to pad both the beginning and the ending of the trace set. The fill value can be any value, though it is advisable to take a value below the minimum value of the rest of the samples (or above the maximum), so that it is easily distinguishable from the actual measurements.

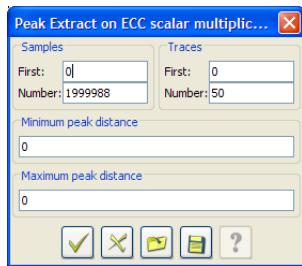
E.9.12 PeakExtract

Version: SCA

Purpose

To extract positive peaks repeating in a trace set. The peaks need not be exactly evenly spaced; instead, they are found within a specified interval.

Dialog



Inputs

The "Minimum peak distance" and "Maximum peak distance" specify the interval at which positive peaks are found. A positive peak is the maximum sample value within each interval.

Result

If a peak is detected at interval i , the next peak is detected within the interval $[i+\minDist, i + \maxDist]$. The first peak is defined to be within $[0, \maxDist - \minDist]$. Obviously, these indices are relative to the start of the selection in the trace set. The indices of all detected peaks are printed in the "Out" panel.

Each of the positive peaks that is found, is copied into the resulting trace set. The length of the resulting trace set is defined to be the number of peaks in the first processed trace.

Related tutorials

- **ECC SPA Tutorial** [[..../tutorials/sectionsECCSPATutorial.html](#)][PDF] [[..../tutorials/Tutorials.pdf#sectionsECCSPATutorial](#)]

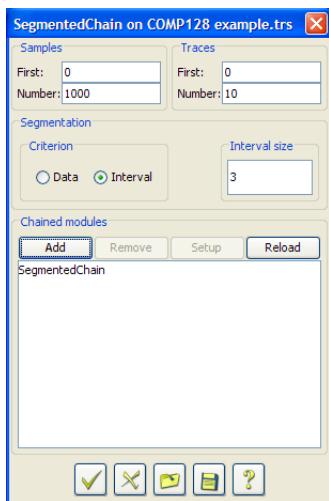
E.9.13 SegmentedChain

Version: SCA

Purpose

Input groups of traces to the chained modules as if they are separate trace sets, and output the concatenation of the module's results.

Dialog



Inputs

This module's inputs are the same as the Chain module, with the addition of a segmentation criterion:

- Data: use changing trace data to determine segment boundaries.
- Interval: use a fixed interval for determining segment boundaries.
- Interval size: the size of the fixed interval for the 'Interval' criterion.

Background

This module splits the input traces into separate trace sets, and runs the specified chain on each of these trace sets. See the Chain module for details on specifying a chain of modules.

The resulting traces produced by the chain are concatenated. This is convenient when we have e.g. a measurement set where each 100 traces are acquired with the same input data, and we want to produce an average trace for each of the 100 traces. This can be achieved by selecting the 'Interval' criterion with an 'Interval size' of 100. Another possibility is selecting the 'Data' criterion, which automatically detects the segment boundaries by detecting changes in the input data. The output is an average for each set of 100 traces.

E.9.14 Spectral

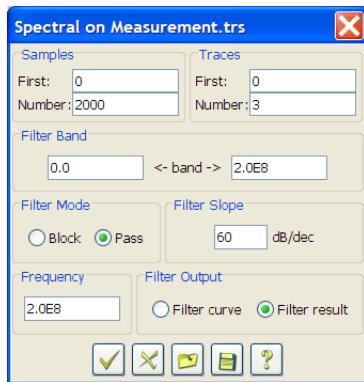
Version: SCA + FI

Purpose

To perform spectral filtering of a trace. A filter can be configured that facilitates efficient filtering of specific frequency ranges from a trace set.

It is also possible to perform Correlation Power Analysis in the *frequency domain*, rather than the time domain. This is accomplished by first running the Spectral module and feeding the resulting trace set (i.e. the FFT traces) to the First-Order crypto module.

Dialog



Input

The Samples and Traces panels are used to set the scope of the frequency analysis or filtering.

The *Filter Band* panel defines the minimum and maximum frequencies for the filter.

The Filter Mode panel defines two options:

- Pass: a frequency-pass filter is applied.
- Block: a frequency-block filter is applied.

The *Filter Slope* edit field defines the slope of the filtering curve. The default slope is 60 dB per decade.

The *Frequency* edit field is used to define the sampling speed of the input trace. By default this is the inverse of the x-scale of the input trace. If, for instance, the x-scale is 2 ns, the frequency is 500 MHz.

Finally, the *Filter Output* panel defines whether only the filter curve is generated, or that it is also applied to the input traces. The former allows the analyst to study the applicability of the filter curve before using it.

Background

Spectral filters use the Fast Fourier Transform (FFT) to block or pass certain frequency components. The FFT involves a trade-off between the number of samples in the time-domain signal, and the number of different frequencies it can represent. For the precision of the frequency filter, it is therefore important to have enough samples in the input signal before filtering. If you are trimming a signal, it is better to do this after frequency filtering.

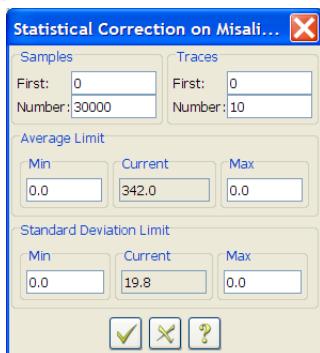
E.9.15 Statistical Correction

Version: SCA

Purpose

To improve the statistical quality of a trace set by rejecting outlying traces and correcting small divergences of the average values.

Dialog



Input

The *Average Limit* panel shows in the middle the average of the selected trace. With the *Min* and *Max* input text fields the analyst can select the range of average values to be accepted by the filter.

The *Standard Deviation Limit* panel shows in the middle the standard deviation of the selected trace. With the *Min* and *Max* input text fields the analyst can select the range of standard deviation values to be accepted by the filter.

Note that without setting the *min* and *max* values the module will not produce any output traces.

Result

The module:

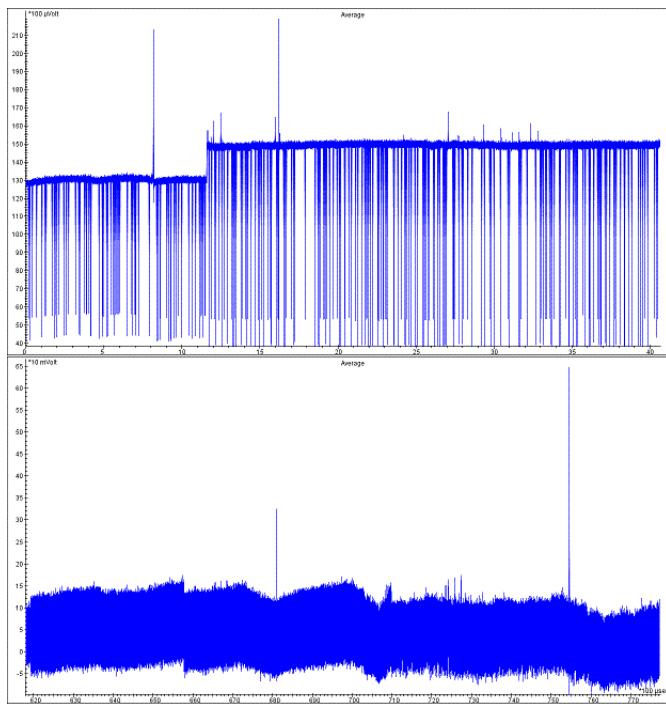
- Rejects traces with an average or standard deviation outside the preset limit values
- Compensates slow variation in average and standard deviation by adjusting the average value and the standard deviation to the middle of the preset limit values.

Background

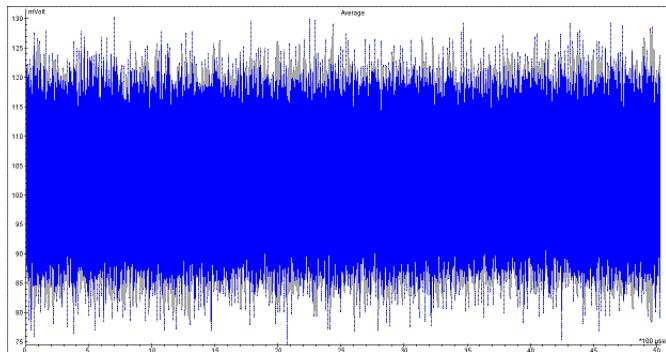
The quality of large trace sets can not be checked manually. One way to perform this verification automatically is to calculate the average (or standard deviation) of each trace in a trace set and plot these average values. Inspector provides this functionality in the average module (select option ‘average per trace’).

The figure below shows an example result of averaging per trace (bottom: overview over complete set, top: zoomed detail). As can be seen in the figure, the quality of a trace set can be reduced by

- Drop outs, single traces with a significantly lower or higher average (or standard deviation)
- Slow variation of average (and standard deviation)



The statistical correction module improves the quality of a trace set, see figure below.



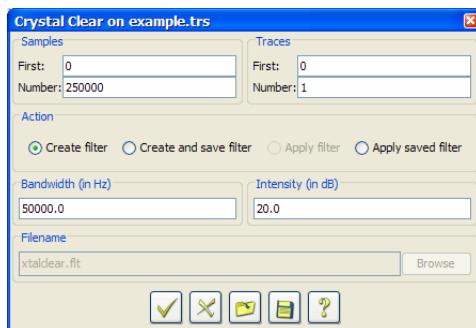
E.9.16 XtalClear

Version: SCA

Purpose

The XTalClear filter is designed to filter all the external clock sub- and higher harmonics that appear in the power or EM frequency spectrum without the need to define them. The XTalClear filter operates in two runs. In the first run the filter searches for all the peaks in the spectrum with a maximum frequency band width and a minimum peak level relative to the background level. These two values (bandwidth and peak level) are the only two parameters for the XTalClear filter. The XTalClear filter needs an averaged frequency spectrum of the RFA or EMA-RF signal as an input. In the second run, the XTalClear is applied to a set of time signal traces.

Dialog



Input

The *Samples* and *Traces* panels are used to set the scope of the frequency analysis or filtering. The *Action* panel defines four options:

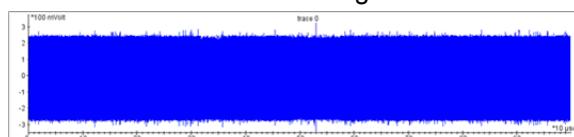
- Create filter: Create the frequency characteristic of the XTalClear filter based on a single frequency spectrum trace as input. The result filter is stored internally.
- Create and save filter: Create the frequency characteristic of the XTalClear filter based on a single frequency spectrum trace as input. The result filter is saved in the specified filter file.
- Apply filter: Just after creating the XTalClear filter, the filter characteristic is still stored in memory and can be used directly. This option is only highlighted when a XTalClear filter is present in memory.
- Apply saved filter: Apply the XTalClear filter as saved in the specified filter file to a set of time signal traces.

The *Bandwidth* panel and the *Intensity* panels give the possibility to tune the XTalClear filter. The Bandwidth refers to the maximum bandwidth of the peaks in the frequency spectrum which are to be removed. The Intensity refers to the minimal difference between the maximum signal level of a peak and the average signal levels at both sides of the bottom of the peak.

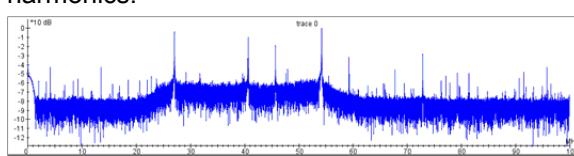
The file used to save or load the filter characteristic can be specified in the *Filename* panel.

Results

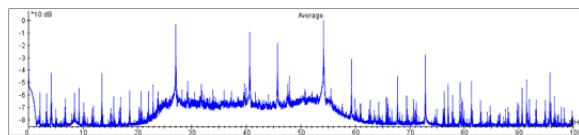
To illustrate the use of the XTalClear filter a trace set is selected with EM measurements on a contactless smart card. The figure below shows the time signal.



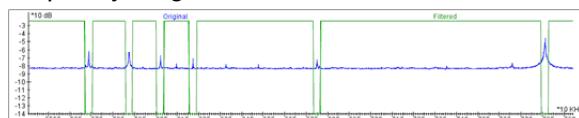
The frequency spectrum of a single trace (below) is noisy with several peaks corresponding the 13.56 MHz RF carrier wave and its sub harmonics, multiples of sub harmonics and higher harmonics.



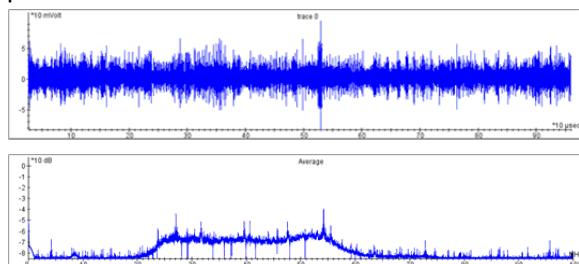
The average of several (>10) spectra (below) is less noisy and usable to create the XTalClear filter



The figure below illustrates the detection of sharp peaks and the generation of blocked frequency ranges in the characteristic of the XTalClear filter.



The figures below show the time signal and frequency spectrum after application of the XTalClear filter. The time signal shows a clear pattern. The frequency pattern only shows wide peaks with a related to unstable – not x-tal driven – internal clocks of the smart card.



Background

The frequency spectrum of the Power or EM signal may contain many sharp peaks. The peak frequencies are at sub harmonics, multiples of sub harmonics and higher harmonics of the external clock frequency or the RF carrier frequency of 13.56 MHz for contactless smart cards. The sharpness of the peaks is due to the stability of the external clock or the clock that generates the RF carrier frequency. The stability of these clocks in turn depends on the stability of the quartz crystal (x-tal) that drives the clock. All sharp peaks in the frequency spectrum are related to an x-tal driven clock.

The leakage of the crypto data is not contained in the pure periodic clock signal and the clock signal can be removed without losing relevant information for side channel analysis. In the frequency spectrum the corresponding sharp peaks can be removed without reducing the crypto data information that is present in the signal.

If the chip runs on an internal clock – not x-tal driven – then the external clock (or RF carrier) sub- and higher harmonics disturb this chip signal because the external clock and the internal chip clock are not synchronized. In that case the external clock sub- and higher harmonics need to be removed.

E.10 icWaves

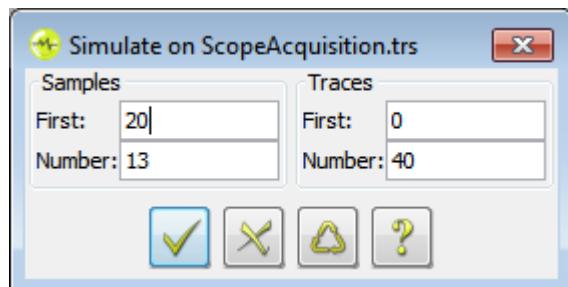
This section contains the descriptions for the icWaves Simulate module.

E.10.1 Simulate

Purpose

To simulate the SAD calculations performed by icWaves in order to determine a proper threshold setting.

Dialog



Input

The Simulate module requires the selection of an input pattern. The selected pattern will be compared against every point in the input trace set using the SAD.

As a result, a trace set containing SAD values is returned to the user. This trace set allows the user to determine whether the selected pattern can be detected by icWaves, and allows to select an initial value for the detection threshold.

See the tutorials provided by Riscure together with icWaves for more information on this module, including hands on exercises.

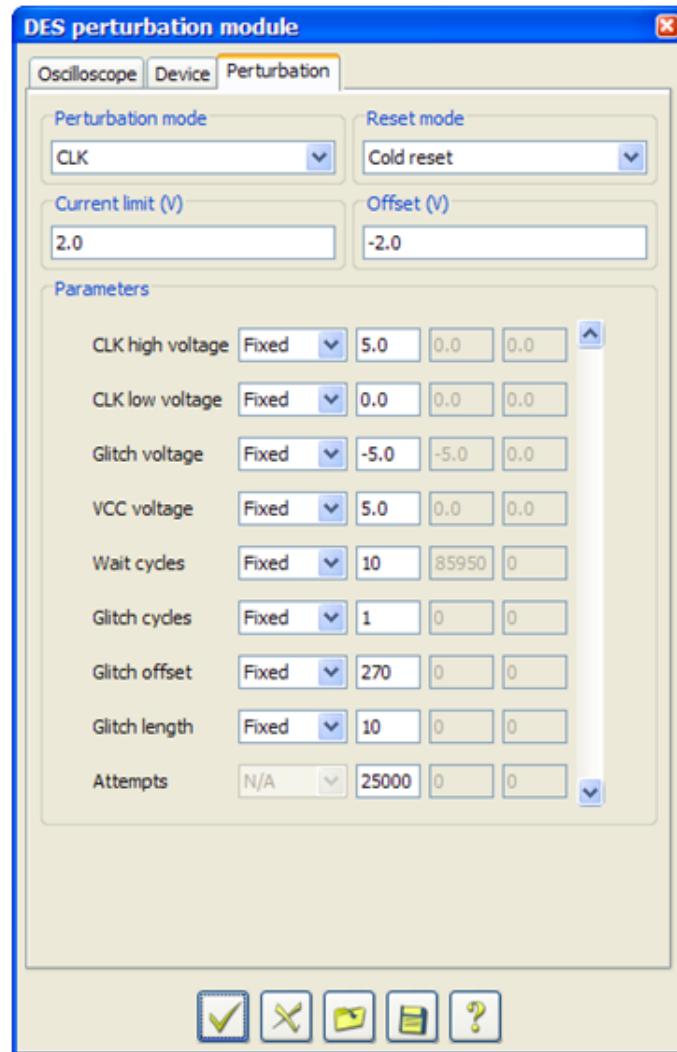
E.11 Perturbation

This section contains the descriptions for all Classic perturbation modules. These modules extend the Perturbation module.

E.11.1 Perturbation Module

To get started with the VC Glitcher quickly, on different implementations, Inspector FI comes with example perturbation modules for implementations like ATR, DES, RSA and PIN glitching.

All modules are pre-configured for glitching the supplied test cards. Refer to the tutorials document for more information. This section describes how to use these modules, based on the DES Perturbation module. First of all open the perturbation module by clicking Run -> perturbation -> DESPerturbation. A window containing three tabs will be opened, as shown in Figure E.6, “DES perturbation module user interface”. For information about the Oscilloscope and Device tabs, refer to the SCA acquisition section. In the Device tab, select VCGlitcherDevice. In the Perturbation tab, the perturbation settings can be entered.

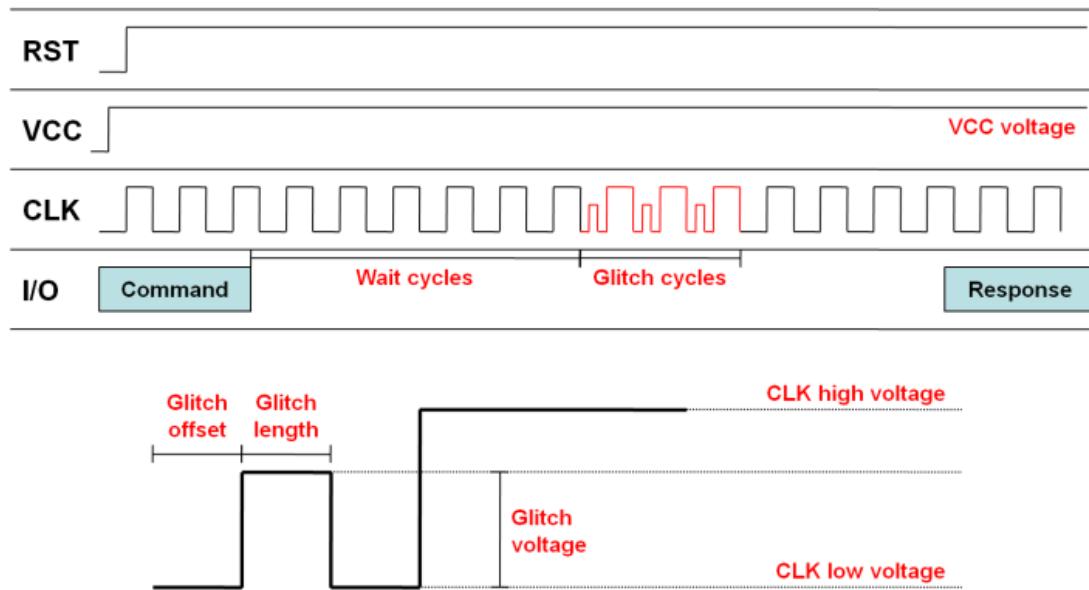
Figure E.6. DES perturbation module user interface

The complete description of all parameters in the DES perturbation module dialog can be found in Smartcard Perturbation in the module chapter. For optical perturbation we refer to Smartcard Optical Perturbation. Perturbation modules can also be used just like acquisition modules. This means that these modules can also perform power measurements using an oscilloscope.

Parameters

A perturbation program can be configured using parameters. The clock, glitch and VCC voltage parameters are automatically available in all perturbation modules. The rest of the parameters are added by the example perturbation modules. An analyst can extend this list by adding more parameters in the perturbation module source code.

See Figure E.7, “Standard perturbation parameters” for an visual overview of the parameters settings.

Figure E.7. Standard perturbation parameters

Each parameter can be set to Fixed, Random or Range. A fixed parameter has a fixed value of x. A random parameter will result in a value between x and y. A range parameter will get all values from x to y with a step size of z.

Suppose the following parameters are defined:

- A Range 100 - 110 with step size 10 = 2 combinations
- B Random in range 100 - 200 = 1 combination
- C Fixed to 7 = 1 combination
- D Range 100 - 105 with step size 1 = 6 combinations

Inspector FI will automatically compute all different combinations of these parameters. In this case the total number of iterations would be $2 \times 1 \times 1 \times 6 = 12$. The following combinations that will be tried are shown in Table E.1, "Parameter combinations".

Table E.1. Parameter combinations

| Test | A | B | C | D |
|------|-----|-----|---|-----|
| 1 | 100 | 159 | 7 | 100 |
| 2 | 100 | 106 | 7 | 101 |
| 3 | 100 | 149 | 7 | 102 |
| 4 | 100 | 163 | 7 | 103 |
| 5 | 100 | 184 | 7 | 104 |
| 6 | 100 | 111 | 7 | 105 |
| 7 | 110 | 138 | 7 | 100 |
| 8 | 110 | 182 | 7 | 101 |
| 9 | 110 | 163 | 7 | 102 |

| Test | A | B | C | D |
|------|-----|-----|---|-----|
| 10 | 110 | 187 | 7 | 103 |
| 11 | 110 | 155 | 7 | 104 |
| 12 | 110 | 183 | 7 | 105 |

VCC and clock glitch modes

The VC Glitcher supports glitching on either the clock signal or the power supply (VCC) of a smart card. The settings are similar for all modes. The regular voltage levels of the VCC and the clock signal, as well as the glitch voltage are specified using Inspector FI. The glitch voltage is superimposed on the target signal, regardless of its original signal level.

The glitch pattern is the pattern that is generated during a clock cycle if glitching is enabled. As described in Section 1.2, the perturbation module is responsible for setting the new glitch pattern and corresponding voltages before each test in a test run, while the perturbation program is responsible for enabling or disabling glitching during the test.

The length of the glitch pattern depends on the chosen smart card frequency, as described in **Smart Card Clock Speed** in the Writing Modules section. One sample in a glitch patterns always represents a period of exactly 2 nanoseconds.

Figure E.8, “The effects of glitch voltages on a clock cycle” shows the effects of different glitch voltages superimposed on a clock cycle.

Figure E.8. The effects of glitch voltages on a clock cycle

| | | | |
|----------------------|----------|----------|----------|
| Clock cycle | | | |
| Glitch pattern | | | |
| CLK high/low voltage | +5V / 0V | +5V / 0V | +5V / 0V |
| Glitch voltage | +5V | +3V | -5V |
| Result (CLK) | | | |

Note that the VC Glitcher cannot drive the clock signal below -0.8V or above +7.5V.

Figure E.9, “The effects of glitch voltages on the VCC” shows the effects of different glitch voltages superimposed on the power supply line (VCC).

Figure E.9. The effects of glitch voltages on the VCC

| | | | |
|----------------|-----|-----|-----|
| Glitch pattern | | | |
| VCC voltage | +5V | +5V | +5V |
| Glitch voltage | -5V | -2V | +2V |
| Result (VCC) | | | |

Note that the VC Glitcher cannot drive VCC below -0.8V or above +7.5V.

Optical

The VC Glitcher can be used to drive a laser station using the Analog glitch and Digital glitch SMB connectors. In this mode, the VCC or CLK signals remain unaffected. Note that in this mode the VCC and CLK voltage cannot be set individually.

VC Glitcher report

In addition to the trace sets, which are used by the analysis modules, Inspector FI produces a report for each test run. An example is shown in Figure E.10, "The VC Glitcher report window". In this report the parameter values for all test runs are stored. In addition, a raw log of the communication is stored.

Figure E.10. The VC Glitcher report window

| VC Glitcher report - DES perturbation module (started 2009-05-05 15:22:38) | | | | | | | | | | | | |
|--|---------------|--------------|----------------|-------------|---------------|---------------|---------------|---------------|---------------|---------------|-----------|--|
| ID | CLK high v... | CLK low v... | Glitch volt... | VCC voltage | Skip cycles 1 | Glitch cyc... | Skip cycles 2 | Glitch cyc... | Glitch offset | Glitch length | Timed out | Data |
| 0 | 5.0 | 0.0 | -5.0 | 5.0 | 725 | 2 | 0 | 0 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 1 | 5.0 | 0.0 | -5.0 | 5.0 | 735 | 3 | 106 | 0 | 258 | 12 | 1 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 2 | 5.0 | 0.0 | -5.0 | 5.0 | 735 | 3 | 106 | 0 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 3 | 5.0 | 0.0 | -5.0 | 5.0 | 735 | 3 | 106 | 0 | 258 | 12 | 1 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 4 | 5.0 | 0.0 | -5.0 | 5.0 | 735 | 3 | 106 | 0 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 5 | 5.0 | 0.0 | -5.0 | 5.0 | 747 | 3 | 106 | 1 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 6 | 5.0 | 0.0 | -5.0 | 5.0 | 747 | 3 | 106 | 1 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 7 | 5.0 | 0.0 | -5.0 | 5.0 | 747 | 3 | 106 | 1 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 8 | 5.0 | 0.0 | -5.0 | 5.0 | 747 | 3 | 106 | 1 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 9 | 5.0 | 0.0 | -5.0 | 5.0 | 747 | 3 | 106 | 1 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 10 | 5.0 | 0.0 | -5.0 | 5.0 | 745 | 3 | 106 | 2 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 11 | 5.0 | 0.0 | -5.0 | 5.0 | 745 | 3 | 106 | 2 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 12 | 5.0 | 0.0 | -5.0 | 5.0 | 745 | 3 | 106 | 2 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 13 | 5.0 | 0.0 | -5.0 | 5.0 | 745 | 3 | 106 | 2 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 14 | 5.0 | 0.0 | -5.0 | 5.0 | 745 | 3 | 106 | 2 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 15 | 5.0 | 0.0 | -5.0 | 5.0 | 734 | 3 | 106 | 3 | 258 | 12 | 1 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 16 | 5.0 | 0.0 | -5.0 | 5.0 | 734 | 3 | 106 | 3 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 17 | 5.0 | 0.0 | -5.0 | 5.0 | 734 | 3 | 106 | 3 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 18 | 5.0 | 0.0 | -5.0 | 5.0 | 734 | 3 | 106 | 3 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 19 | 5.0 | 0.0 | -5.0 | 5.0 | 734 | 3 | 106 | 3 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 20 | 5.0 | 0.0 | -5.0 | 5.0 | 741 | 3 | 106 | 4 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 21 | 5.0 | 0.0 | -5.0 | 5.0 | 741 | 3 | 106 | 4 | 258 | 12 | 0 | 8E 00 FF 81 31 FE 45 52 69 73 63 72 65 SA 00 00 DA A0 18 |
| 22 | < n | < n | < n | < n | 741 | 1 | not | 4 | 748 | 19 | n | |

All reports are stored in a local PostgreSQL database server. Old reports can be opened using the Tools -> Open log file... menu in Inspector FI.

The columns can be sorted on by clicking the column header. Clicking a second time will change the sorting order. At the bottom of the logging window, a filter expression can be entered. As the logging data is stored in an SQL database, the filtering capabilities are quite powerful. For example: "Glitch cycles" = 3 AND "Skip cycles" = 8 OR "Timed out" = 1

Common operators, such as <, <=, >, >= or != are also available.

All standard modules use the same color scheme:

Perturbation mode is used to select one of the following modes:

- Green means that the glitch did not have an effect.
- Yellow means that the smart card reset or muted during the test.
- Red means that the glitch caused some kind of fault.

It is also possible to filter on the row colors. Although "Color" is not a column in the report, it is possible to use it in the SQL expression

- "Color"=3 to show only the red rows
- "Color"=4 to show only the green rows
- "Color"=5 to show only the yellow rows

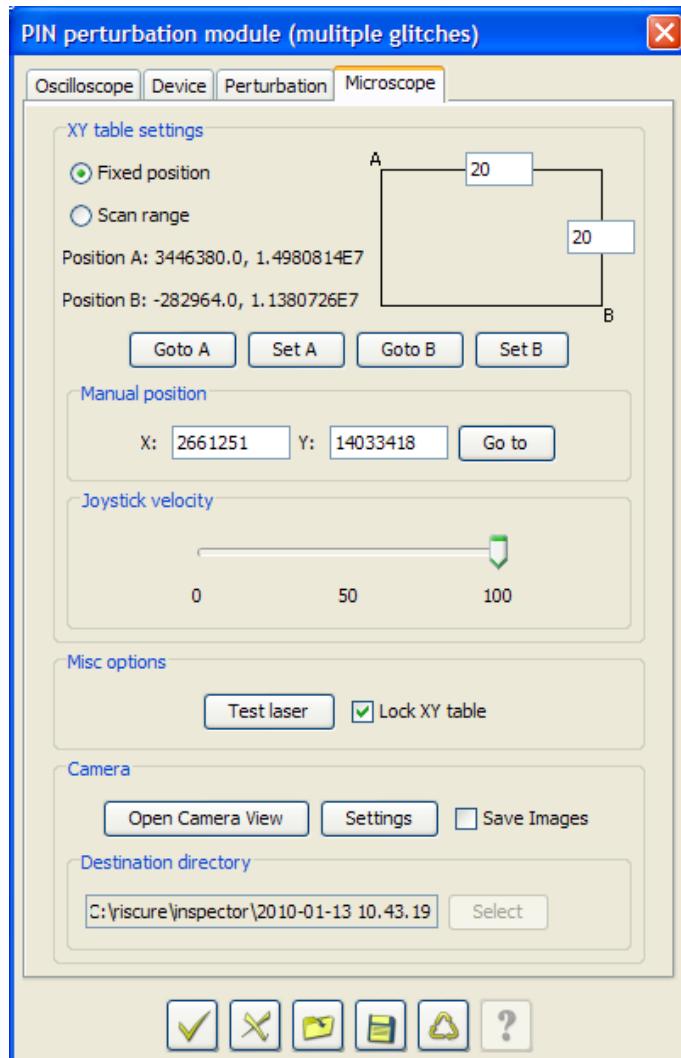
E.11.2 SmartCardOpticalPerturbation

Version: F1

Purpose

This module is intended to perform an optical perturbation attack on a device. It is module is application-independent, and cannot be executed directly. It must be extended for a specific application. This module is used in all the optical perturbation modules like *AESOpticalPerturbation* and *DESOpticalPerturbation*. For more information on writing perturbation applications refer to the writing perturbation modules section.

Dialog



Input

This helpful describes the content of the "Microscope" tab. For more information about the other tab, refer to the Oscilloscope, Side Channel Acquisition and Smart card perturbation help files.

- The *XY table settings* panel is used to configure the XY table. It can be set to either stay on a fixed position or to scan a range. In order to scan a range, the top-left (A) and bottom-right (B) corners need to be set. Then, on the right side, specify the number of columns and rows.
- The *Manual position* panel is used to manually move the XY table to the specified position. Just insert the X and Y coordinates and press *Go to*.
- The *Joystick velocity* panel allows to set the speed of the joystick.
- The *Misc options* panel contains two controls:

- The *Test laser* button will fire the laser by using the current perturbation settings (i.e. Laser power, Glitch length, offset and cycles). This can be used to visually verify whether the laser is working and on which area it is firing. Note that, depending on the power filter and the perturbation settings, the pulse might be too weak/short to be captured by the camera. Therefore if it is not visible we suggest to increase the laser power and/or the glitch cycles.
- The *Lock XY Table* check box specifies whether the joystick is locked during an acquisition. It is usually recommended to leave it locked so that it is not possible to accidentally move it during a perturbation. However, it is sometime required to be able to manually move it during a perturbation. If this is the case, then the check box should be unchecked.
- The *Camera* panel contains several controls:
 - The *Open camera view* button opens a new window which shows the camera view
 - The *Settings* button opens a new window containing some camera-related settings
 - The *Save Images* check box specifies whether, during an XY scan, a picture of every location should be saved to disk
 - The *Destination directory* field specifies the target directory for the camera pictures

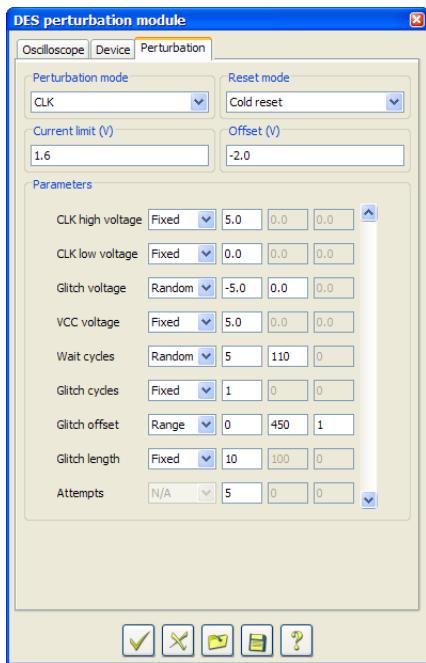
E.11.3 SmartCardPerturbation

Version: F1

Purpose

This module is made to perform a perturbation attack on a device. It is application-independent, and cannot be executed directly. It must be extended for a specific application. This module is used in all the non-optical perturbation modules like *AESPerturbation* and *DESPerturbation*. For more information on writing perturbation applications refer to the writing perturbation modules section.

Dialog



Input

The *Oscilloscope* tab is defined in the help file for the Oscilloscope module, while the *Device* tab is explained in the help file for the Side Channel Acquisition module. The following additional controls are provided:

- The *Perturbation Mode* drop-down box allows the selection of the kind of perturbation. Three attacks are possible:
 - VCC: performs a glitch on the VCC line of a smart card
 - Clock: performs a glitch on the clock line of a smart card
 - Optical: drives an external device for laser fault injection
- The *Reset Mode* is used to select one of the following reset modes:
 - Cold reset: The smart card is reset by disabling the power supply and pulling down the reset line.
 - Warm reset: The smart card is reset only by using the reset line.
 - No reset: No reset performed between measurements. As the smart card is not reset, de-synchronization can occur. Therefore, this mode is not recommended.
- In the *Current limit* field, the current limit for the smart card can be specified. Although current is specified in Amperes, the current limit should be entered in Volts, in order to match the vertical unit used by Inspector. Depending on the smart card, this feature can be used to prevent a smart card from writing to EEPROM memory. In case the threshold is reached, the smart card will be cold reset. Note that the current limiter must be enabled/disabled in the perturbation program using the `enableCurrentLimiter` and `disableCurrentLimiter` instructions.

- The *Offset* field allows specifying an offset (in Volts) to be applied to the power consumption output. It must be set accordingly to the smart card power profile, so that the signal doesn't clip during acquisition. The accepted range for this field is -2.77 - 0.0 Volt
 - The *Parameters* panel holds a series of parameters that define the perturbation. A module can extend the list (please refer to the VC Glitcher manual for instructions on how to add parameters). A parameter can be of three different types:
 - *Fixed*: The parameter always holds the same value
 - *Random*: The parameter holds a random value within the specified range
 - *Range*: The parameter holds all the possible values starting from the minimum value to the maximum value; the parameter is incremented by the specified setting.
- Depending on the type of perturbation, several parameters are present:
- CLK
 - *CLK high voltage* parameter represents the voltage of the clock signal when the clock is high.
 - *CLK low voltage* parameter represents the voltage of the clock signal when the clock is low.
 - *Glitch voltage* defines value to add to the clock line voltage when glitching. This field can hold a negative value.
 - *VCC voltage* defines the voltage of the VCC line.
 - *Attempts* defines the number of times every combination of the parameters will be tested.
 - VCC
 - *VCC voltage* defines the voltage of the VCC line.
 - *Glitch voltage* defines value to add to the VCC line voltage when glitching. This field can hold a negative value.
 - *CLK voltage* defines the voltage of the high part of a clock cycle. The low part is always set to 0.
 - *Attempts* defines the number of times every combination of the parameters will be tested.
 - Optical
 - *Glitch voltage* defines value output to the digital/analog glitch lines of the VC Glitcher.
 - *VCC/CLK voltage* defines the voltage of the VCC line and the high part of a clock cycle. The low part is always set to 0.
 - *Attempts* defines the number of times every combination of the parameters will be tested.

Furthermore, several other parameters are defined in the provided perturbation modules:

- *Wait Cycles* defines the number of clock cycles to wait before performing the glitch. Note that in case the clock speed provided to the smart card is changed, this number might have to be recomputed (i.e. when the smart card uses an internal clock).
- *Glitch Cycles* defines the number consecutive clock cycles to glitch.
- *Glitch Offset* defines the start offset of a glitch. Note that in case the clock speed provided to the smart card is changed, this number must be recomputed.
- *Glitch Length* defines duration of a glitch in 2 ns samples. Note that in case the clock speed provided to the smart card is changed, this number must be recomputed.
- The Number of attempts parameter is a different from the others. Using this parameter, one can repeat each test a number of times. This is especially useful for optical glitching.

E.11.4 TriggeredOpticalPerturbation

Version: F1

Purpose

To perform a perturbation attack on a device, using the VC Glitcher in triggered mode. This module is application-independent, and cannot be executed directly. It must be extended for a specific application. This module is used in the triggered optical perturbation modules like *RSATriggeredOpticalPerturbation* and *DESTriggeredOpticalPerturbation*. For more information on writing perturbation applications refer to the writing perturbation modules section.

Input

Please read the section on SmartCardOpticalPerturbation for descriptions of the input fields.



Note

The sample modules are developed for these settings:

The *reset mode* on the *Perturbation* tab should be set to 'cold reset'.

The *Error Handling* on the *Device* should be set to 'abort'.

See also

The user interface for triggered optical perturbation modules is identical to the SmartCardOpticalPerturbation modules. Please read that section on information of the GUI elements. More information on triggered perturbation setups can be found in the section on Triggered Perturbation

E.11.5 TriggeredPerturbation

Version: FI

Purpose

To perform a perturbation attack on a device, using the VC Glitcher in triggered mode. This module is application-independent, and cannot be executed directly. It must be extended for a specific application. The sample module *DESTriggeredPerturbation* extends *TriggeredPerturbation*

Input

Please read the section on SmartCardPerturbation for descriptions of the input fields.



Note

The sample modules are developed for these settings:

The *reset mode* on the *Perturbation* tab should be set to 'cold reset'.

The *Error Handling* on the *Device* should be set to 'abort'.

See also

The user interface for triggered perturbation modules is identical to the SmartCardPerturbation modules. More information on triggered perturbation setups can be found in the section on Triggered Perturbation

E.12 Perturbation2

This section describes the Perturbation2 modules. Inspector ships with a large number of FI modules. These will not all be discussed in this section, because many modules overlap considerably with regard to their functionality. This chapter covers six FI modules in detail and the reader is encouraged to browse through the different sections if the particular module that he is using is not mentioned explicitly.

Please do not confuse the Perturbation module with the other (old) Perturbation module in perturbation1. Since perturbation1 is considered deprecated the reader should take note that we always refer to perturbation2, except when explicitly mentioned otherwise.

E.12.1 SC Perturbation

Purpose

This is the most basic Perturbation2 module. It allows for use of the VC glitcher while communicating with the target. This module can be used for clock glitching and voltage glitching. Inspector FI comes with a number of protocols for standard implementations (like AES, DES, RSA, etc.) which can be used with the training cards that are shipped with Inspector. This allows for a quick start.

The Perturbation module has six tabs, which are also present in the Optical Perturbation modules. These six basic tabs are necessary to communicate with a target, set up a glitching device, and specify the actual perturbation behavior. The tab labelled "icWaves Setup" is not covered here, since it is already discussed in Section 6.5.1, "icWaves". Note that some

additional options are available when the user checks the *Show advanced settings* option at the bottom of the module frame.

General tab

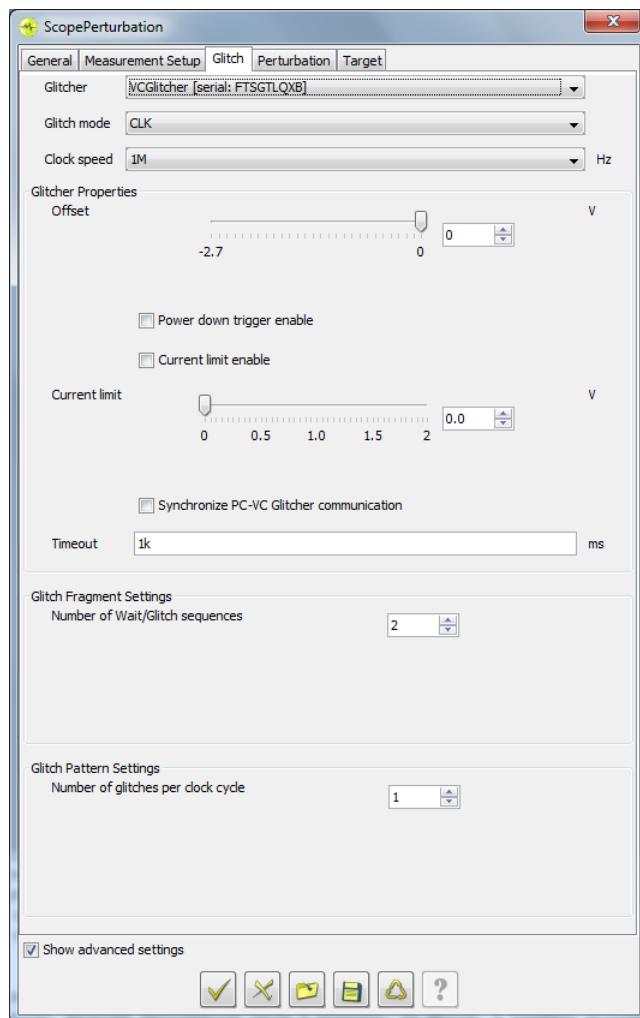
- *Accept errors*: accept measurements that had errors during acquisition or running the target.
- *Limit errors*: Limit the number of consecutive errors allowed during acquisition or running the target.

Measurement Setup

- *Oscilloscope*: Select the oscilloscope device. The sine generator should be listed there, as well as other connected devices.
- *Oscilloscope Settings*: Oscilloscope-specific settings
- *Channel properties*: Channel-specific properties

Glitch

Figure E.11. Glitch tab of Perturbation module

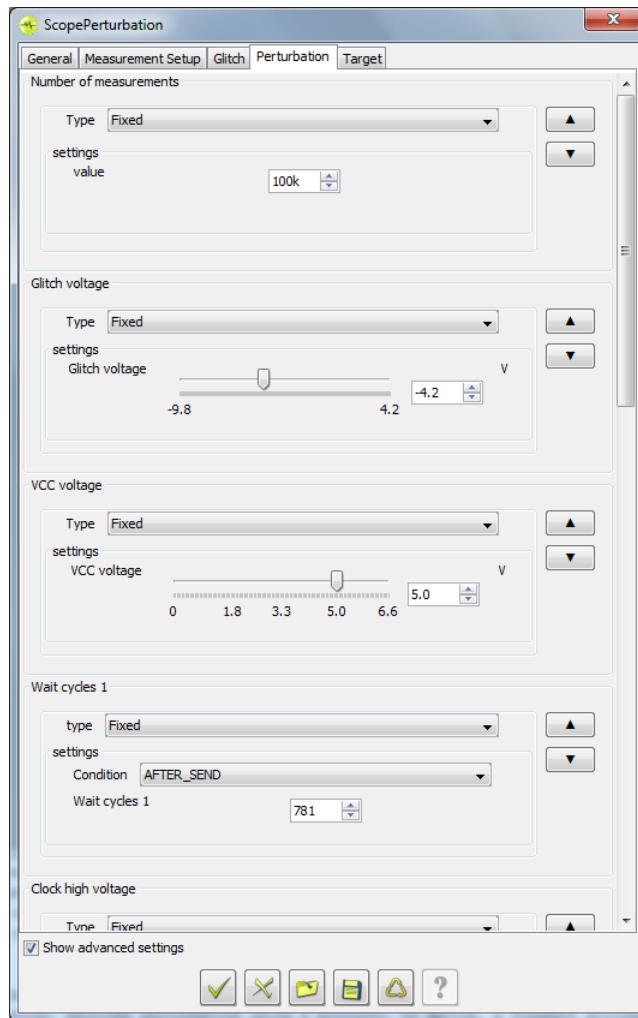


The Glitch tab of the Perturbation module is shown in Figure E.11, "Glitch tab of Perturbation module". There are a number of options here, where it should be noted that the Glitcher Properties panel is only visible when a physical VC Glitcher is selected.

- *Glitcher*: Select the device. There should at least be a "dummy glitcher" device in the dropdown list. Any glitcher device that is connected to the computer should be visible here. If not, see if it is visible in the Hardware Manager or Inspector or in the Device Manager of Windows. Possibly the device driver was not properly installed.
- *Glitch mode*: The VC glitcher supports both voltage glitching (VCC) and clock glitching (CLK).
- *Clock speed*: Select the clock speed that the VC glitcher generates for the smartcard interface.
- *Glitcher Properties*: These are device-specific settings.
 - *Offset*: Select an offset that will be applied to the Power Monitor output.
 - *Power down trigger enable*: If enabled, the VCGlitcher will monitor the "Power down trigger" input channel. When a high signal is detected, the smartcard will be immediately powered off.
 - *Current limit enable*: If enabled, the VCGlitcher power off the smartcard if the card current exceeds the specified threshold.
 - *Current limit*: The threshold value for the current limiter.
 - *Synchronize PC-VC Glitcher communication*: If enabled, communication between the VCGlitcher and the PC will be synchronized, to improve the success rate on smartcards running on external clock. **Important:** This option is only needed when using cards running on external clock and with no random delays; enabling it will slow down the communication speed. If the target is a smartcard running on its internal clock or if another device is used to communicate with the target this option should not be enabled.
 - *Timeout*: The maximum amount of time a perturbation attempt can take before aborting the operation.
- *Glitch Fragment Settings*: The user has to specify the number of glitches.
- *Glitch Pattern Settings*: The user has to specify the number of glitches per clock cycle.

Perturbation

Figure E.12. Perturbation tab of Perturbation module



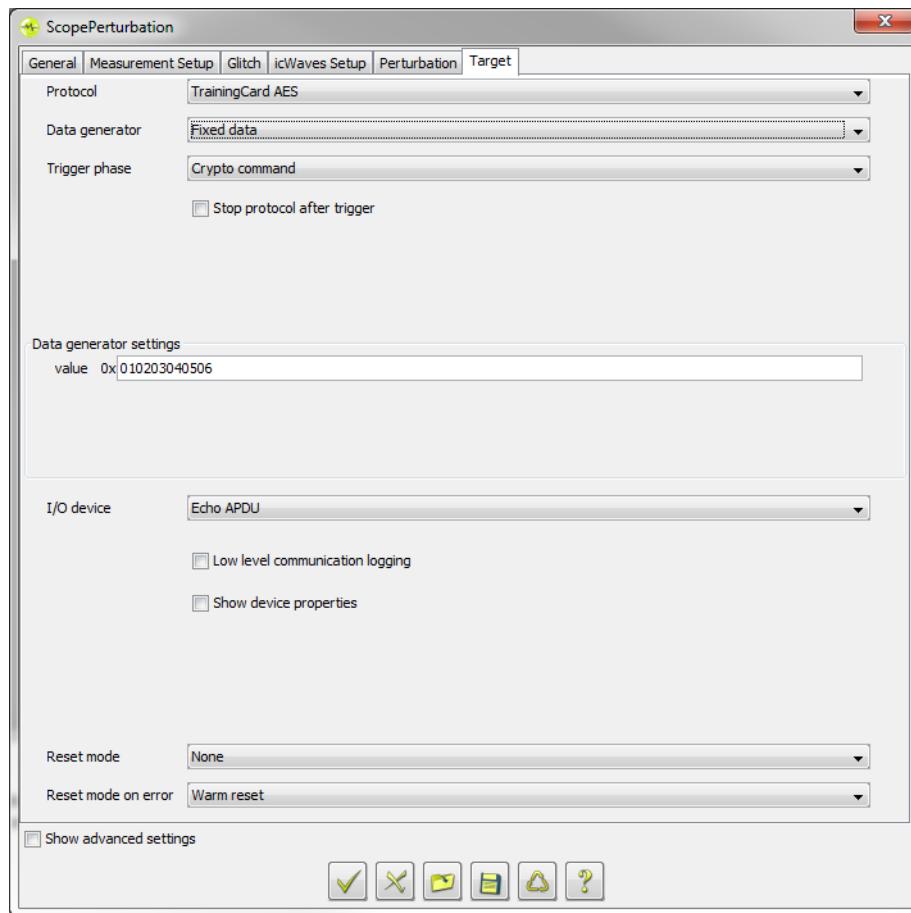
- **Number of measurements:** The number of measurements to perform.
- **Glitch voltage:** The value to add to the VCC line voltage when glitching. This field can hold a negative value.
- **VCC voltage:** The voltage of the VCC line.
- **Clock high voltage:** The voltage of the high part of a clock cycle.
- **Clock low voltage:** The voltage of the low part of a clock cycle.
- **Glitch offset:** The number of samples before a glitch (expressed in nanoseconds; since the resolution is 2ns, only even values are allowed).
- **Glitch length:** The duration of a glitch (expressed in nanoseconds; since the resolution is 2ns, only even values are allowed).
- **Wait cycles:** The amount of clock cycles to wait before performing a glitch. This parameter has an additional condition field:

- *After trigger*: The specified amount of clock cycles is relative to the trigger APDU defined in the target tab
- *On rising external trigger*: The specified amount of clock cycles is relative to a rising edge on the trigger input channel. Note that the VCGlitcher will only start listening after sending the trigger APDU defined in the target tab.
- *On falling external trigger*: The specified amount of clock cycles is relative to a falling edge on the trigger input channel. Note that the VCGlitcher will only start listening after sending the trigger APDU defined in the target tab.
- *With previous*: The specified amount of clock cycles is relative to the previously performed glitch. This option is only meaningful when multiple glitches are performed. If this condition is selected for Wait cycles 1 (which doesn't have a previous glitch) it will have the same effect as "After trigger".
- *Glitch cycles*: The number of glitches to perform after "Wait cycles".

Target

The target tab of the Perturbation module is shown in Figure E.13, "Target tab of Perturbation module". In inspector 4.6 the application protocols have been re-implemented. In Inspector 4.5 the user could specify the data validation rules (like "does it end with 9000?") in the Communications verifier settings panel. The current version allows for adapting the standard interpretation (i.e. "90 00" is normal, otherwise it is successful or inconclusive) in java code.

- *Protocol*: Select the protocol that should be used to communicate with the target.
- *Data generator*: The input can be either "fixed data", "fixed list", "random data", "random list", or "step data". See Section 4.4.2, "Data Generator" for more details. Note that not all application protocols have this field.
- *Trigger phase*: This is a protocol-specific setting. See Protocol Phases [57] for more information.
- *Stop protocol after trigger*: Whether to stop the protocol after the trigger.
- *Protocol settings*: Protocol-specific settings.
- *Data generator settings*: Data generator specific settings.
- *I/O device*: Select the I/O device and specify the settings. Note that, when using a VCGlitcher, the T=1 option "Suppress error recovery" should be selected, to avoid performing error recovery when the smartcard is in an unknown state after a perturbation attempt.
- *Reset mode*: The "cold reset" means that the smart card is reset before starting a perturbation attempt by disabling the power supply and pulling down the reset line. The "warm reset" only pulls down the reset line. "None" means that no reset is performed between measurements.
- *Reset mode on error*: Like "Reset mode", but only performed when an error is detected in the smartcard communication. This value should be as least as strong as the "Reset mode". Also see Note on reset behavior.

Figure E.13. Target tab of Perturbation module

E.12.2 SC Single XYZ Perturbation

Purpose

Apart from VCC/clock glitching, which works via the power/clock line, fault injection can also be applied via an electromagnetic or laser pulse. This type of fault injection has the advantage that it can be applied locally on a die; the disadvantage is that finding "the right spot" requires testing a wide range of locations. Currently, Inspector ships with two XYZ devices: The EM Probe Station for EM-FI (and EM Acquisition), and the Diode Laser Station for optical perturbation. The Inspector UI is transparent for the actual XYZ device, meaning that EM-FI and laser glitching can be performed with the same module.

There are a number of XYZ Perturbation modules in Inspector. These all differ slightly, depending on the exact use case: are you using a single or a dual location attack, are you evaluating a smart card or an embedded target, do you communicate with a standard protocol or with a raw protocol, do you use the standard VC Glitcher attack or an advanced (customized) glitch program? The specifics for all these combinations are described in other module descriptions. Luckily, the XYZ aspects of all these modules are the same and we will discuss them in this section. Only for the "dual XYZ" scenario are there some relevant differences. There, the typical use case is a dual-laser setup, where both laser devices can be controlled independently (notice the two "XYZ Device" tabs in the modules), facilitating a multi-area attack. Note that the term "XYZ device" is a generic denominator for any laser or EM

probe manipulator, even if it only has two degrees of freedom. In this section, we will focus on the basic SC Single XYZ Perturbation module—whenever specific dual-optical functionality is present, we will discuss it explicitly.

The SC Optical Perturbation module has eight tabs. The General tab and the Measurement Setup tab are already described in Section E.12.1, “SC Perturbation”. Also the Target tab is the same as in the Perturbation module, so we will not discuss it here.

Glitch Source

This tab is very much comparable with the Glitch tab in the SC Perturbation module (see Section E.12.1, “Glitch”). The main difference is that here, we select a “glitch source” (i.e. a laser or an EM-FI probe) instead of VCC/Clock mode.

- *Glitcher*: Select the glitching device that will be used to convey the pulses to the laser or the EM pulse generator.
- *Glitch source*: Specify the glitch device. This can be a 808nm laser, a 1064nm laser, or an EM-FI transient probe.
- *Clock speed*: The clock speed of the target.
- *Glitch Fragment Settings*: The user has to specify the number of glitches.
- *Glitch Pattern Settings*: The user has to specify the number of glitches per clock cycle.
- *Test glitch source power*: The user can specify the relative strength of the laser/EM-FI probe that is used for testing.
- *Test glitch source cycles*: The length of the test glitch pulse, expressed in number of cycles.
- *Test glitch source*: This button engages the laser or EM-FI probe. In case of optical perturbation the laser light blob should be visible in the camera view.

icWaves Setup

The user can integrate an icWaves device in his setup. We won't discuss it here, since it is already explained in Section 6.5.1, “icWaves”.

XYZ Device

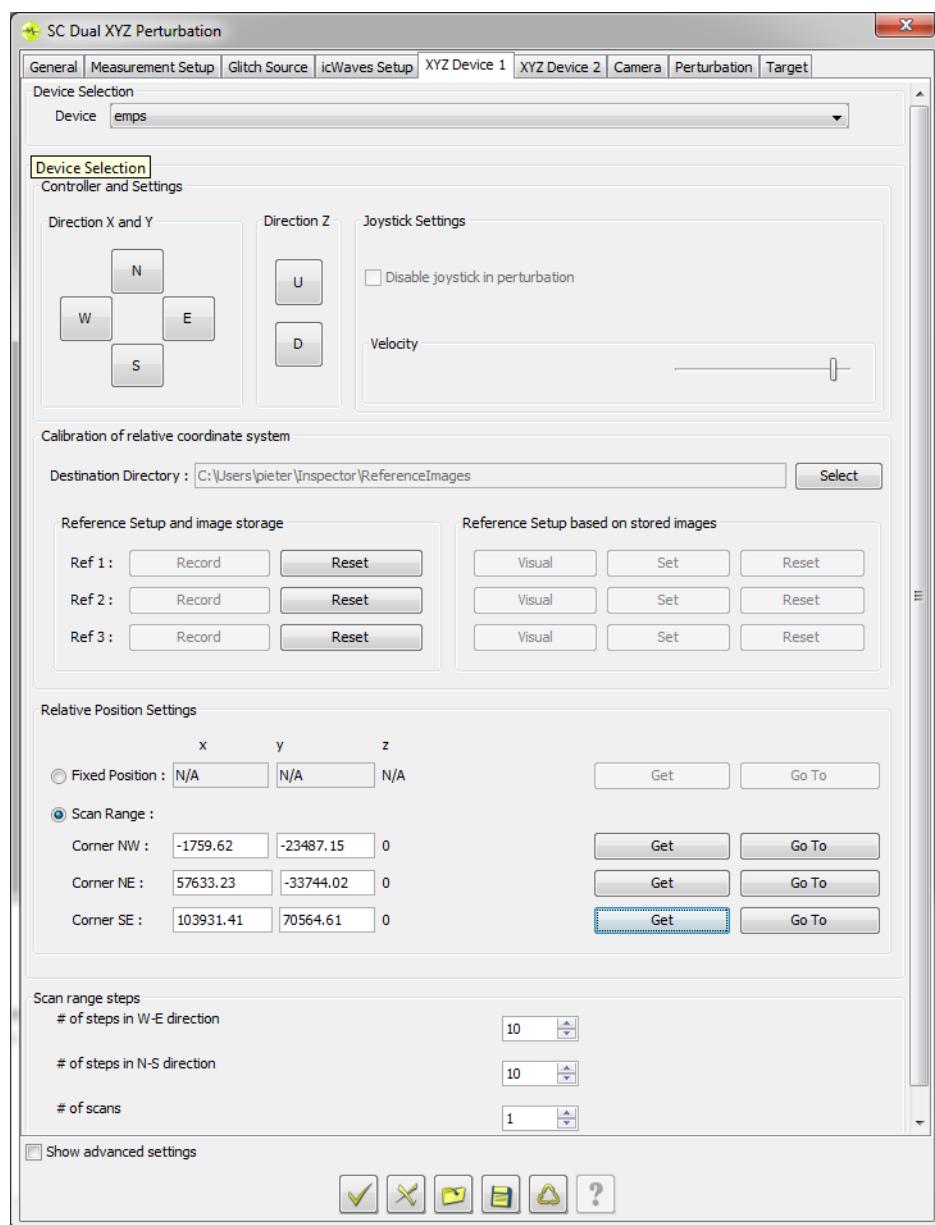
This tab represents the physical controller of the laser or EM Probe Station. The interface is the same for both use cases.

- *Device Selection*: All connected laser/EM-FI probe manipulators should be visible in the dropdown list. The “XYZ dummy device” should always be available.
- *Controller and Settings*: Though the joystick will likely be the preferred way to direct the EM-FI probe, laser table or fiber, the user can control the XYZ device with the WNES buttons (and the Up/Down buttons when applicable). The velocity of the joystick control can also be specified (between 0% and 100%). We recommend the “Disable joystick in perturbation” option, to avoid accidental movement during the session.
- *Calibration of relative coordinate system*: The user can specify the location where snapshot images will be stored. The default directory is "%USERPROFILE%\Inspector

\ReferenceImages\}. Images from previous sessions will always be saved in a subdirectory called "old_reference_points". The use of the reference points is described in Section 5.5.1, "Coordinate Systems: XYZ Device vs. Chip" of this document and in the MADLS tutorial.

- **Relative Position Settings:** The user can choose for a "fixed position", where the optical or EM glitch is fired from one specific location, or for a "scan range", where a number of positions are iterated over. Note that all coordinates are normalized, i.e. relative to the chip.
- **Scan range steps:** When the user has indicated a scan area, then he can specify the number of stops in both the West-East direction and the North-South direction. The "# of scans" indicates how many times the scan range has to be repeated. This last option should not be confused with the "number of measurements" in the Perturbation tab: that option reflects the number of optical glitches *per scan location*.

Figure E.14. XYZ tab of Dual XYZ Perturbation module

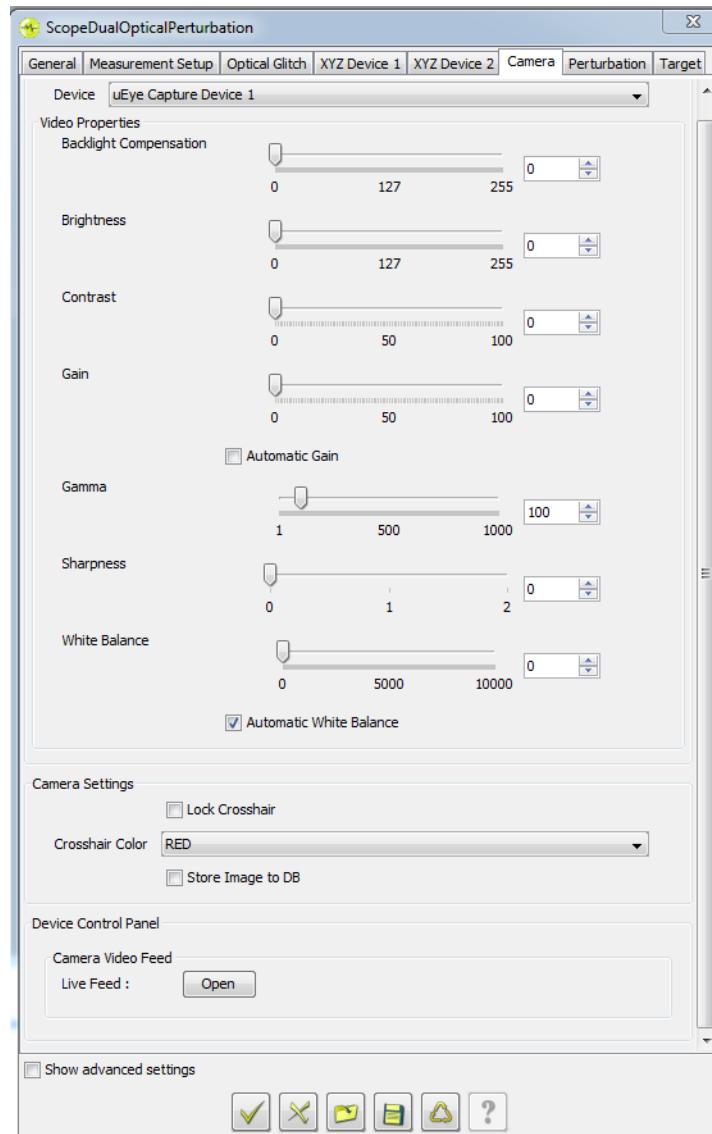


Camera

The DLS is equipped with a video camera. This camera is used to position the laser on a precise location. The camera images are also stored for each reference point. In the EM-FI scenario there typically is no camera. Nevertheless, the user has to select the Dummy Camera there.

- *Device Selection:* Each connected USB video camera will be automatically listed here. The "Dummy Video Camera" can be used to perform a test perturbation when no real camera is available.¹ For any real device a "Video Properties" panel will be displayed with all the controls that the specific camera supports.
- *Camera Settings:* There is a crosshair overlay on the camera image (press the "Open" button at the bottom of this tab to show the camera view). This crosshair must be calibrated with the laser spot. On the "Optical Glitch" tab the user can fire a test shot, which should be visible on the video. The user can then move the crosshair to that laser spot with a mouse click on the video. The option *Lock Crosshair* is useful to prevent accidental movement of the crosshair. Depending on the background color the user can choose an alternative *Crosshair Color*. When the checkbox *Store Image to DB* is selected a screenshot of each perturbation location is stored during the process.
- *Device Control Panel:* Currently, this panel only contains the *Live Feed: Open* button.

¹The dummy camera also stores dummy images of the reference points. These images can be ignored.

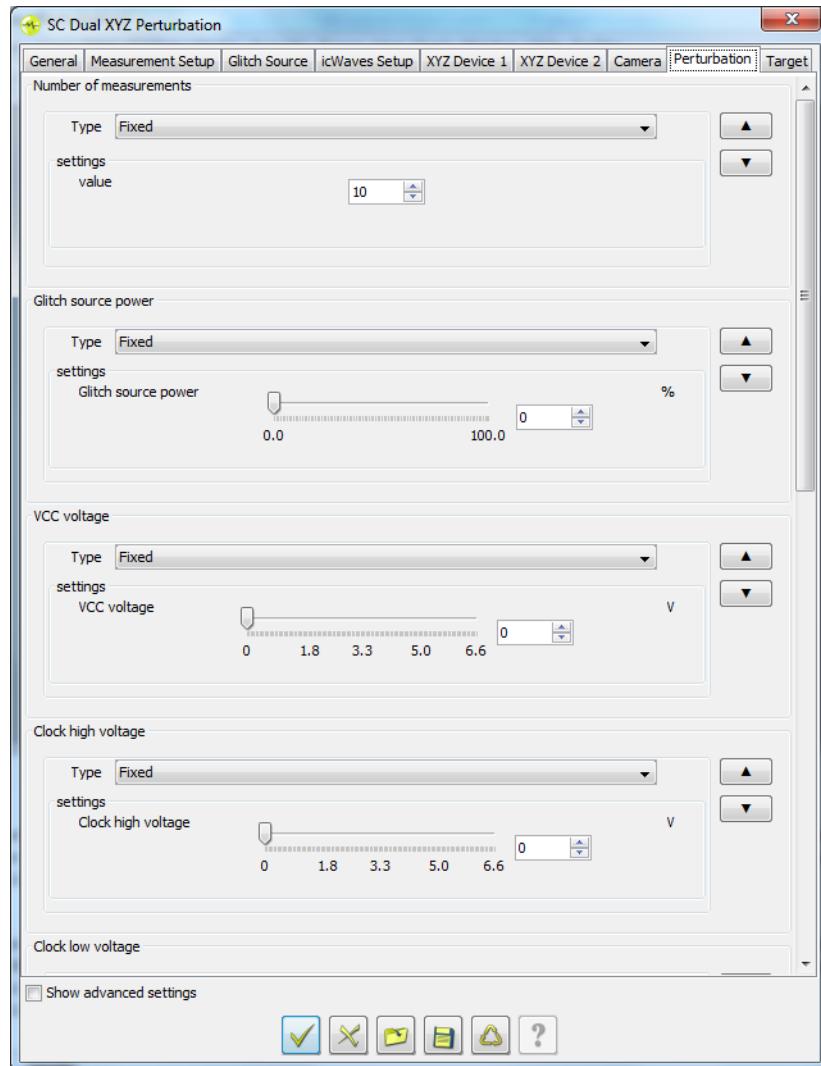
Figure E.15. Camera tab of Dual XYZ Perturbation module

Perturbation

This tab is almost identical to the Perturbation tab of the Perturbation module. See Section E.12.1, "SC Perturbation" for the details. The only panel that "XYZ" adds to this tab is the following:

- *Glitch source power*: Like the other perturbation parameters the laser/EM strength can be either "fixed", "random", or in a "range". A fixed glitch power can be specified with the slider, while a random (uniform) distribution or a range can be bounded with the range slider.

Figure E.16. Perturbation tab of Dual XYZ Perturbation module



E.12.3 SC Perturbation with Glitch Program

Purpose

Instead of using the default behavior of the VC Glitcher, the user can specify customized programs. This allows for detailed control over the device during a perturbation run. The user can write his own program, which is called a *custom glitch snippet*. Such a snippet may contain numerical parameters (which are tunable on the Perturbation tab). These parameters can then be used in for example a wait statement.

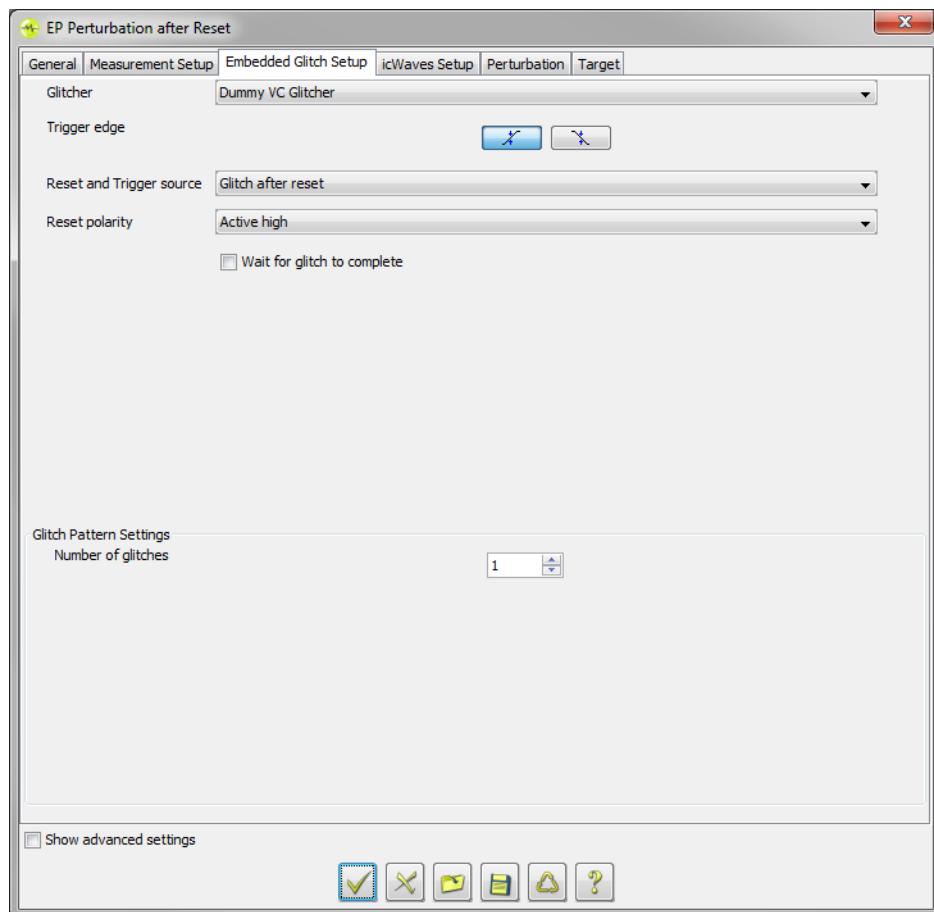
A custom snippet is basically a java class. It can be constructed by hand or by using the *wizard*. We will focus on the latter option, since it's easier to get started. Under the "File" menu there is the option "New module Wizard...", which offers several templates, including "Glitch Snippet". The next step is to specify the descriptors: "Company name" and "Snippet name". The company name will be inserted in the comments of the java class as a copyright line. The snippet name will be used for the class name. We advise users to select a name that conforms to the Java coding standards, so start with an upper case letter. Anything after a space will be

ignored. Next, the "Sniplet type" can be selected. The option "Beginner" will create a Sniplet class with some example code to get started. The "Advanced" option results in a class without any code. After clicking on "Finish" the IDE is opened with the new Sniplet code. The user can now adjust the generated code and press the Compile button to verify the validity of the class.

The SC Perturbation with Glitch Program module has five tabs, which are similar to the Perturbation module. The main difference between these two modules is the "Custom Glitch" tab, which we will discuss below. For the other tabs we refer to Section E.12.1, "SC Perturbation".

Custom Glitch

Figure E.17. Custom Glitch tab of the Sniplet Perturbation module



- **Glitcher:** Select the device. There should at least be a "dummy glitcher" device in the dropdown list. Any glitcher device that is connected to the computer should be visible here. If not, see if it is visible in the Hardware Manager of Inspector or in the Device Manager of Windows. Possibly the device driver was not properly installed.
- **Glitch mode:** The VC glitcher supports both voltage glitching (VCC) and clock glitching (CLK).
- **Clock speed:** Select the clock speed that the VC glitcher generates for the smartcard interface.

- *Custom Snippet*: Select the snippet. Any java class that extends PerturbationSniplet and is found in the user/Inspector/modules/ directory (or a subdirectory) will be listed here.
- *Custom Snippet Variable Settings*: The user can introduce variables in his custom Snippet and map these to a glitch register. Consider for instance the following line in the example code generated by the wizard:

```
glitchSnippet.LOAD(glitchRegisters.get(0), "MyPropertyName1");
```

This command will result in a GUI variable called "MyPropertyName1" and load the value of that variable to register R0 of the VC Glitcher. The list of variables must be specified in the `getVariables()` method. These variables will then automatically be displayed on the Perturbation tab, NOT on the glitch tab.

- *Glitch Pattern Settings*: The user has to specify the number of glitches per clock cycle.

E.12.4 Perturbation Advanced Program

Purpose

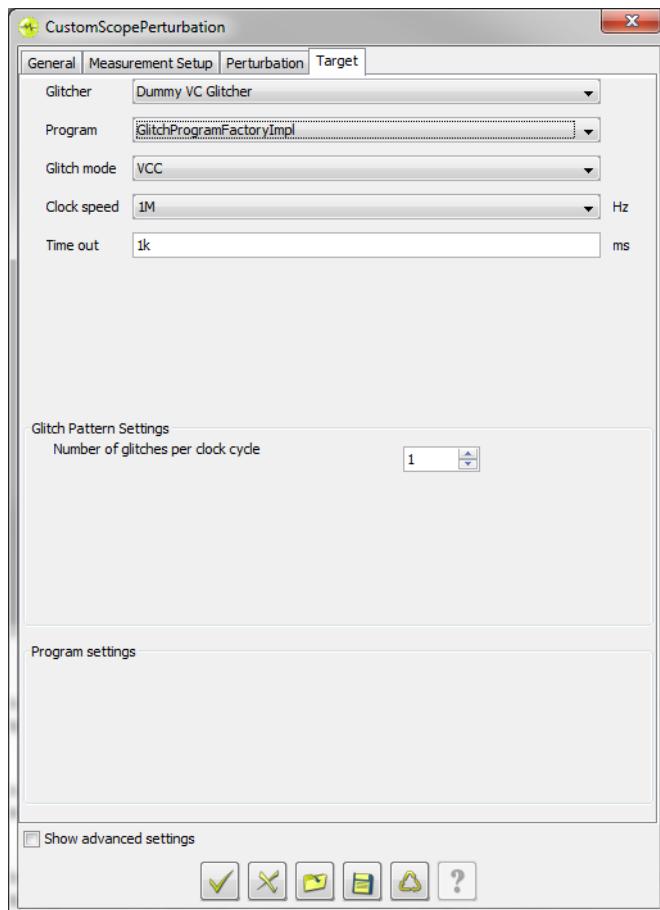
The VC Glitcher does not only offer a transparent mode. You can also program the VC Glitcher yourself. This allows for detailed control over the device during a perturbation run. The user can specify a *custom glitch program* which will be loaded to the VC Glitcher's CPU.

Just like a Snippet, a custom glitch program is a java class that can be either written (or copied by hand, or prepared by the "New module wizard"). The main difference between custom Snippets and programs is that for Snippets the variable addresses are hidden (you just declare a name), while for programs the variables do have an absolute integer address. The process of creating a Perturbation Program with the wizard is very similar to creating a custom Snippet, so we refer to Section E.12.3, "SC Perturbation with Glitch Program" for more information.

The ProgramPerturbation module has four tabs. Compared to the other Perturbation modules there is no Glitch tab, where otherwise the glitching device has to be selected. Now, the VC Glitcher is the target. We focus on the Target tab. For the other tabs we refer to Section E.12.1, "SC Perturbation".

Target

Figure E.18. Target tab of Perturbation Advanced Program module



The Target tab of the ProgramPerturbation module is shown in Figure E.18, "Target tab of Perturbation Advanced Program module". The following options are available.

- **Glitcher:** Select the device. There should at least be a "dummy glitcher" device in the dropdown list. Any glitcher device that is connected to the computer should be visible here. If not, see if it is visible in the Hardware Manager of Inspector or in the Device Manager of Windows. Possibly the device driver was not properly installed.
- **Program:** Select the custom program that you want to load to the VC glitcher. Any java class in the user/Inspector/modules directory that extends BasicGlitchProgram will be listed here. If the list is empty, then close the module and go to "File" -> "New Module Wizard..." and create a new "Program".
- **Glitch mode:** The VC glitcher supports different modes, but the choice depends on the type of glitching and the type of target. The basic options are voltage glitching (VCC) and clock glitching (CLK). If you want to apply optical glitching you can select the LASER option, but in that case we advice to extend this module yourself and include an XYZ Device tab and a Camera tab. For an embedded TOE the options EMBEDDED_VCC and EMBEDDED_LASER are available.
- **Clock speed:** Select the clock speed that the VC glitcher generates for the smartcard interface.

- *Time out*: The default time out value is 1000 milliseconds, but this can be altered by the user.
- *Glitch Pattern Settings*: The user has to specify the number of glitches per clock cycle.
- *Program Settings*: The user can add variables to his custom program.

E.12.5 SC Perturbation after Reset

Purpose

An ATR (Answer To Reset) attack can be accomplished with this module. The idea is that the (voltage) glitch attack is performed during the first communication phase of the target, where the chip outputs its usage information in reaction to the release of the reset line. Typically, this attack is applied to smart cards. This module is sometimes referred to as RawPerturbation, since it doesn't make assumptions about the target's protocol.

Inputs

This module is very similar to the Perturbation module, with two main differences. First, the protocol is not very relevant, since any smart card sends its ATR before actual (application-specific) communication is initiated. For this reason the RawPerturbation module comes with a single protocol called ReadAll, which simply reads all bytes that the target sends.

Second, on the Perturbation tab the "Wait cycles x" fields have a condition named "After trigger". This option should be selected for "Wait cycles 1". It means that the sequence of wait cycles will start relative to the release of the reset line, that is, after the boot. The user can further delay the start of the wait cycles by entering a positive value for the "Number of bytes to read" field on the Glitch tab. The first wait cycle will be scheduled when the specified the number of ATR bytes have been received.

E.12.6 EP Perturbation after Reset

Purpose

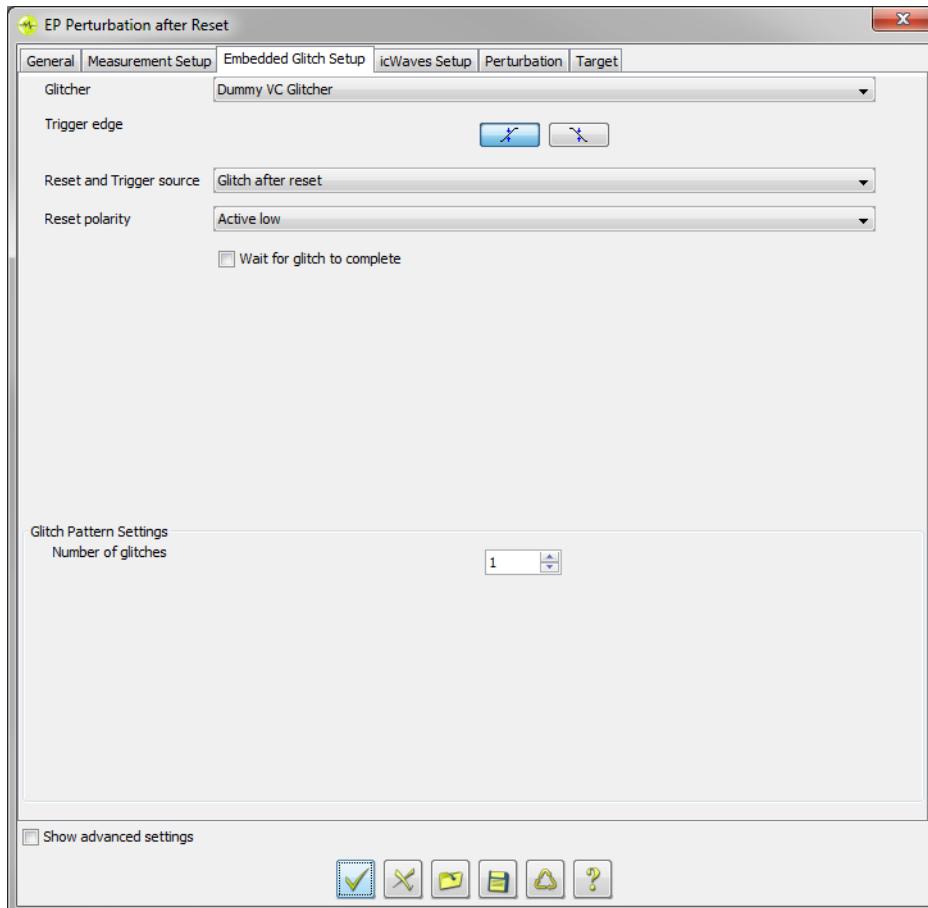
Attacking an embedded target is similar to attacking a smart card. The main differences are that we have no control over the clock and that the communication protocol is not standard (like for smart cards). This means that we can't specify a perturbation program for the VC Glitcher. Instead we have to define a glitch sequence: a list of <glitch offset, length> pairs in units of 2 nanoseconds. This provides more flexibility, but the concepts of wait cycles and glitch cycles are lost.

The glitch sequence will be started after the trigger. Since there is no perturbation program and no controlled clock signal we have to link the trigger to either the (hard) reset or the trigger phase² indicated by the raw protocol. The default raw protocol ("ReadAll") simply reads bytes from the target, without any assumptions on the specific target. Analogous to the previous section, we sometimes refer to this module as RawEmbeddedPerturbation.

² See Protocol Phases [57] for more background on protocol phases.

Embedded Glitch Setup

Figure E.19. Glitch tab of Raw Embedded Perturbation module



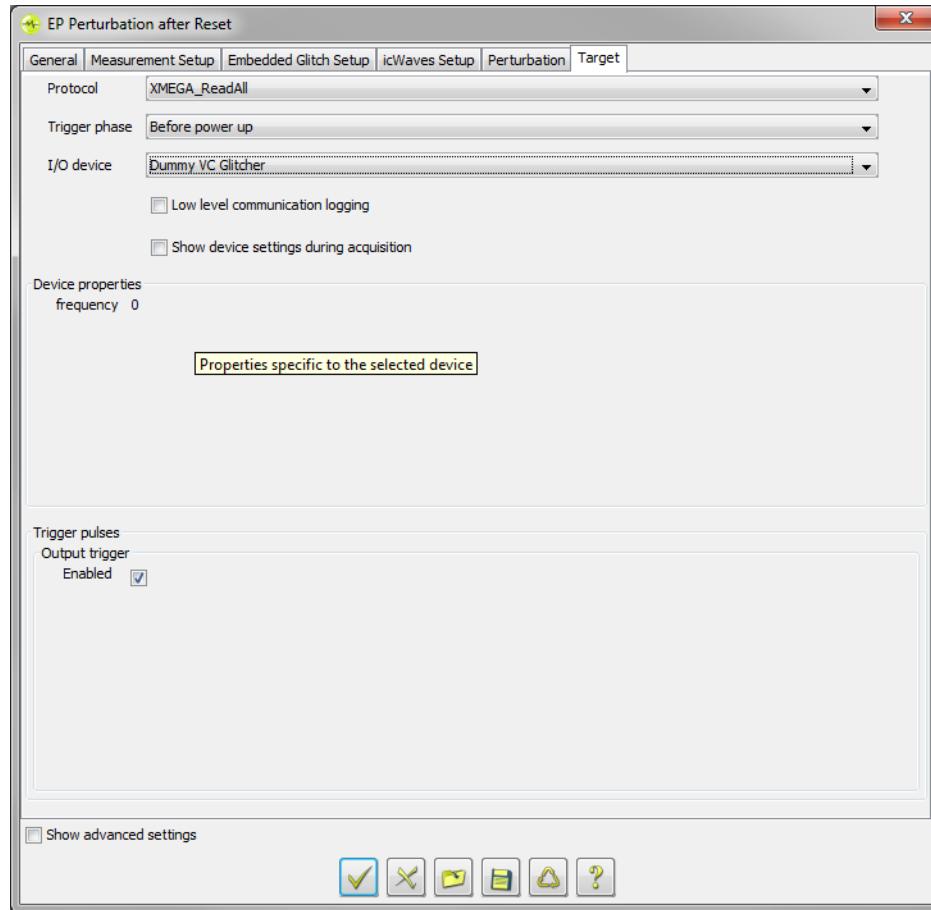
- **Glitcher:** The same as for the other perturbation modules (see Section E.12.1, “Glitch”). Note that the device should be an EmbeddedGlitcher device, i.e. a glitching device which supports the embedded mode.
- **Trigger source:** Since the embedded glitcher does not have a perturbation program we have to specify the (physical) trigger source. The trigger source can be a hard reset (Powerdown trigger), a soft trigger, an external trigger, or a CPU trigger.
- **Trigger edge:** The user can specify the glitch sequence should start at the rising or the falling edge of the trigger signal.
- **Reset polarity:** Depending on the ToE, the reset line can be either interpreted as active on the high signal or on the low signal.
- **Glitch Pattern Settings:** The user has to specify the number of glitches. Increasing the number will lead to additional "glitch offset" and "glitch length" parameters on the Perturbation tab.

Target

The (embedded) device is conceptualized as a "raw target". This means that we interface directly with the device, instead of using a transport protocol. The default protocol for this

module is the ReadAll protocol, which simply reads all data that the device sends to Inspector. It doesn't have a distinction between an initialization phase and a command phase since there is no standard protocol. The user can write his own RawProtocol subclass if he requires more sophisticated communication or more control over the power and/or reset line. An example of this can be found in Section 9.9.2, "Embedded Boot Glitching With a Warm Reset".

Figure E.20. Target tab of Raw Embedded Perturbation module



- *Protocol*: The default protocol for this module is the ReadAll protocol, which simply reads all data that the device sends to Inspector.
- *Data generator*: The user can specify whether fixed or random data should be sent to the card. See Section 4.4.2, "Data Generator" for more details.
- *Trigger phase*: The glitcher can either start when the raw protocol says so (the ReadAll protocol only has a single trigger phase, but this can be customized by the user), when the device performs a power up, or when the device powers down.
- *I/O device*: Usually, the communication with the target passes through the VC Glitcher. However, the user can also specify that a COM port should be used for this.

E.13 Sort

This section contains the descriptions for all the sort modules.

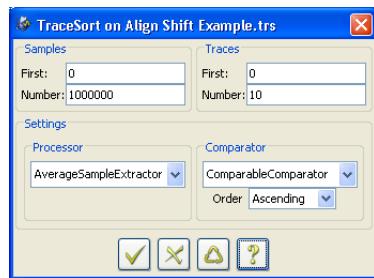
E.13.1 TraceSort

Purpose

To sort the *Traces* within a *TraceSet* on a specifiable criterium.

The *TraceSort* module provides a basic framework for sorting a *TraceSet*. This framework allows a *TraceSet* to be sorted on virtually any criterium, by allowing custom criteria to be plugged into the module.

Dialog



Input

- **Processor** Selects an implementation of the interface `sort.interfaces.TraceProcessor`. Implementations of this interface process a Trace and output a value that is later used to sort a *TraceSet* with, *the derivative*.
- The following processors are provided by default:
- **AverageSampleExtractor** Extracts/computes the average sample value of a Trace.
 - **DataExtractor** Extracts the data associated with a Trace. An offset (in bytes) in the associated data and the desired length (in bytes) of the data can also be specified.
 - **FirstSampleExtractor** Extracts (the value of) the first sample of a Trace.
 - **Comparator** Selects an implementation of the interface `sort.interfaces.TraceDerivativeComparator`. This interface extends `java.util.Comparator<Object>`. Implementations of this interface are able to compare derivatives with each other for the purpose of sorting.
 - **ComparableComparator** Compares all implementations of `java.util.Comparable<T>`. This implementation is compatible with the default provided TraceProcessors. It also provides the option of sorting a *TraceSet* in an ascending or descending order.

Results

A *TraceSet* sorted on the criterium specified by the selected *TraceProcessor*.

Background

TraceProcessor

In order to sort a TraceSet on a custom criterium, the interface `sort.interfaces.TraceProcessor` needs to be implemented. This interface defines one method:

```
Object process(Trace trace);
```

The TraceSet will be sorted on the output of the `process()`-method and this Object can be of any type. An example of an implementation of TraceProcessor is provided in the package `sort.example`. The class `FirstSampleExtractor` implements the TraceProcessor interface as follows:

```
public class FirstSampleExtractor implements TraceProcessor {  
    @Override  
    public Object process(Trace trace) {  
        float[] samples = trace.getSample();  
        // Pushes the trace to the end  
        if(samples == null || samples.length == 0) {  
            return Float.NaN;  
        }  
        else {  
            return trace.getSample(0);  
        }  
    }  
}
```

TraceDerivativeComparator

In order to define a custom sort order, the interface `sort.interfaces.TraceDerivativeComparator` needs to be implemented. This interface defines one method:

```
int compare(Object o1, Object o2)
```

The number returned by the `compare()`-method should be less than 0 when $o1 < o2$, equal to zero when $o1 = o2$ and greater than 0 if $o1 > o2$. An implementation of TraceDerivativeComparator, `ComparableComparator`, is provided in the package `sort.example`. This implementation is able to compare all implementations of `java.util.Comparable<T>`

E.14 Statistics

This section contains the descriptions for all the statistics modules.

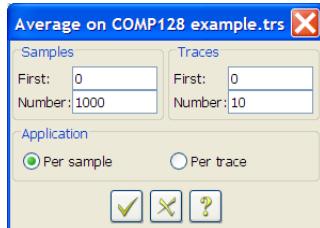
E.14.1 Average

Version: SCA + FI

Purpose

To compute an average trace from a set of multiple traces.

Dialog



Inputs

The averaging can be done either per sample or per trace. With the former option each sample of the resulting trace is the average of the samples at the same horizontal offset from all input traces. The latter option computes a resulting trace where each sample is an average sample of one input trace.

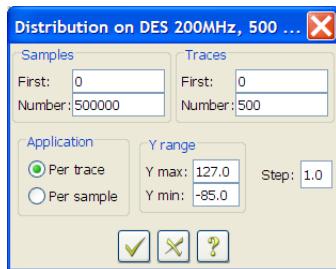
E.14.2 Distribution

Version: SCA + FI

Purpose

To compute the value distribution of samples per trace, or for a trace set. This analysis can be useful to observe noise or to distinguish separate coding paths.

Dialog



Input

The *Samples* and *Traces* panels define the scope of the distribution analysis.

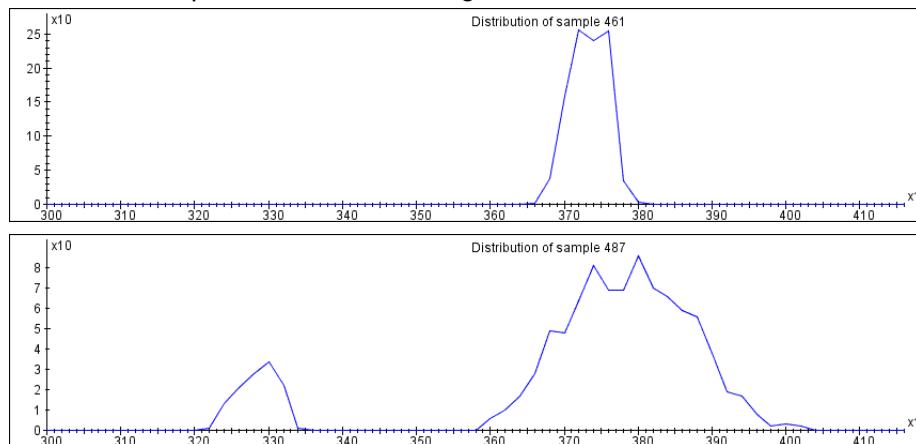
The *Application* panel defines whether the distribution analysis is performed on each individual trace, or on all selected traces. In the former case the result traces will show which values occur within a specific input trace. In the latter case the result traces will show which values occur for a specific sample analysed for all traces.

The Y range panel defines the range of values to be displayed in the result graph. Values that fall outside this range are included in the boundary values of the range.

The value of Step defines the range of values considered equal for the distribution computation.

Example

The figures below show how distribution graphs can reveal two different coding graphs. At sample 461 all traces represent the execution of the same code. At sample 487 the execution of two different parts of code are distinguished.



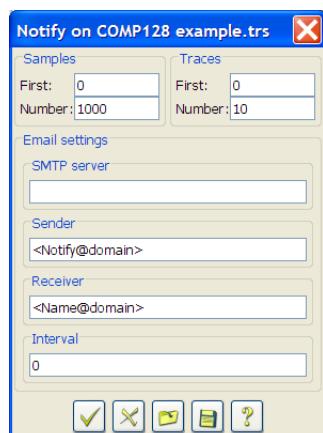
E.14.3 Notify

Version: SCA + FI

Purpose

To report the progress or result of an operation by e-mail. The module itself is a transparent filter: it returns traces unmodified.

Dialog



Background

This module should not be used alone, but in a chain after one or several filter modules. An analyst who is away from the acquisition or analysis setup can keep himself informed about the progress of the process. Note that this module only works if Inspector runs on a system connected to the internet.

Input

SMTP server: an internet server that will accept outgoing e-mail, for example smtp.mail.yahoo.com.

Sender: The sender of the e-mail message (e.g. name@mailaddress.com [mailto:e.g. %20name@mailaddress.com]).

Receiver: The recipient of the e-mail message (e.g. name@mailaddress.com [mailto:e.g. %20name@mailaddress.com]).

Interval: The number of traces after which to report. A zero value means 'report only when finished'. By setting a value of for instance 1000 the module will send an email to the receiver after processing each 1000 traces.

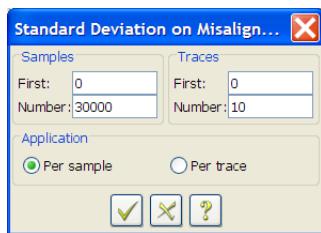
E.14.4 StandardDeviation

Version: SCA + FI

Purpose

To compute the standard deviation of a trace set. The result can be used to judge the need for alignment or verify the quality of an aligned trace set, because non-alignment causes a significant standard deviation. It can also be used to detect countermeasures or to get an indication of the noise level.

Dialog



Input

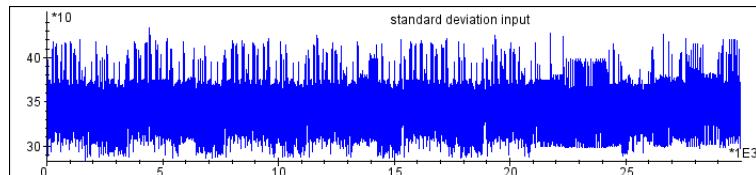
The *Application* panel selects whether the standard deviation is computed per sample or per trace. In the first case each sample in the resulting trace represents the standard deviation of the corresponding sample over all input traces. In the latter case each sample in the resulting trace represents the standard deviation of all samples in the corresponding input trace.

Result

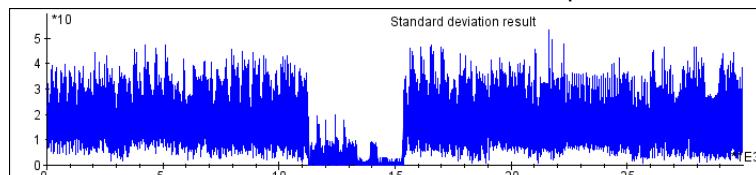
A trace set with a single trace, in which each sample represents the standard deviation of the corresponding samples (identical offset) in all input traces.

Example

The figure below shows the first trace of a trace set that has been aligned on the centre part:



The standard deviation of this trace set shows quite well where the trace is aligned:



Related tutorials

- **Second order AES Tutorial** [[..../tutorials/sectionsSecondOrderAEScard.html](#)][PDF] [[..../tutorials/Tutorials.pdf#sectionsSecondOrderAEScard](#)]

E.15 Configuration

This section contains the description of modules that are used to configure a device or a target. These modules do not perform acquisition or fault injection.

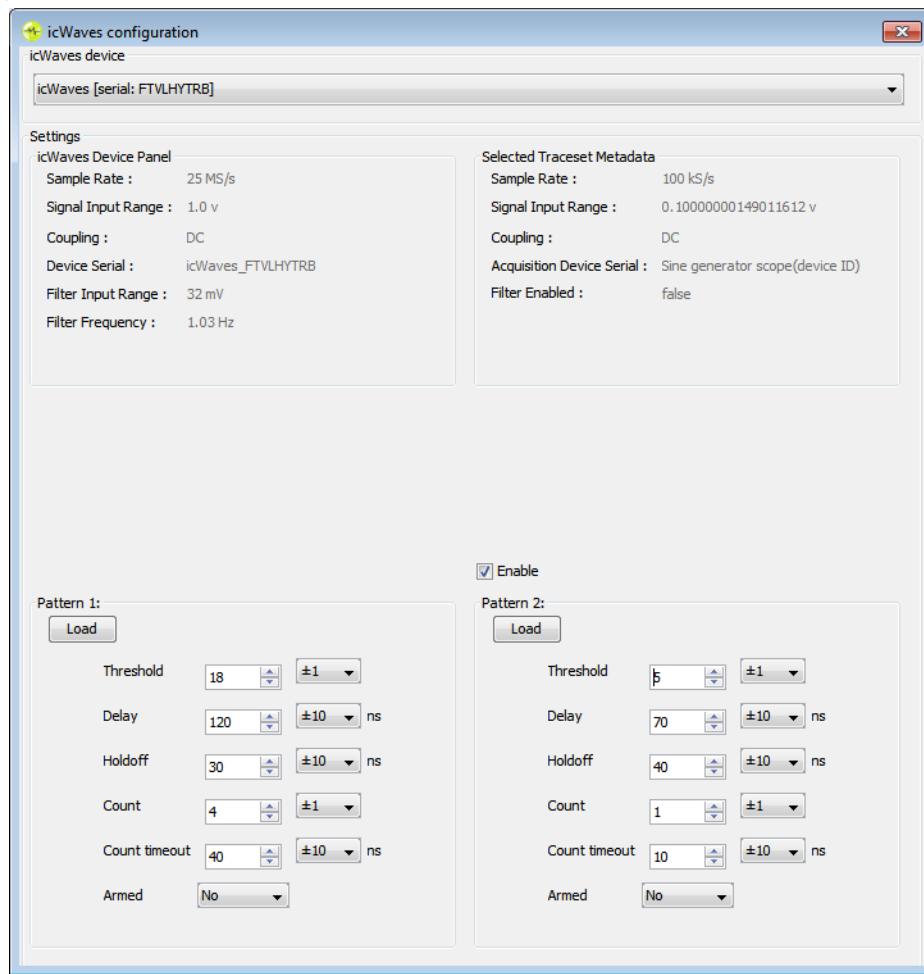
E.15.1 icWavesConfiguration

Version: SCA + FI

Purpose

This module allows the user of the icWaves to set the different parameters recognized by the device.

Dialog



Inputs

The module contains several panels:

- The icWaves Device Panel shows the info regarding the current icWaves device settings
- The Selected Traceset Metadata Panel shows the parameter settings of the icWaves at the time of acquiring this selected traceset
- The two pattern panels can be used to configure the two patterns. The configuration options are described in the section called "Triggering". The +100, +10 and +1 option buttons affect the step size of the spinners in this panel. The *Load* button can be used to load the selected part of a trace as a reference pattern.

The module remembers the previous settings that were used with the icWaves and automatically configures the device on startup. This means that if the user wants to reuse the same reference pattern, threshold and frequency, the only thing he has to do is open the module, select 'Yes' to the Arm option

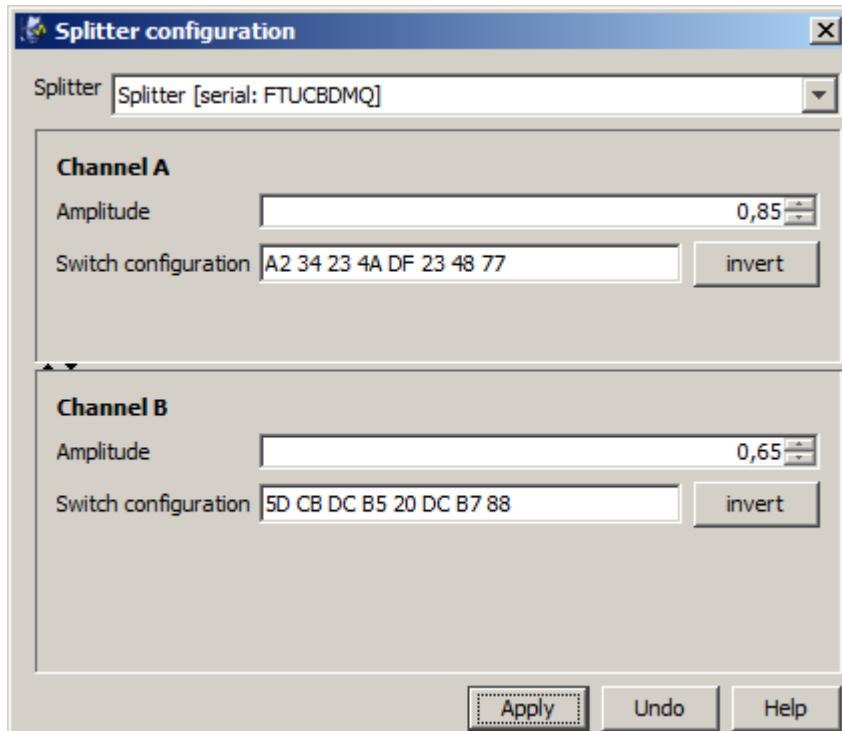
E.15.2 splitterConfiguration

Version: F1

Purpose

This module allows the user of the Splitter to set the different parameters recognized by the device.

Dialog



Inputs

The module contains several components:

- The *Splitter* drop-down list displays the connected Splitters. Selecting a Splitter from this list will display its settings and allow the user the settings. This drop-down list is automatically updated when Splitters are added or removed.
- The *Channel A* and *Channel B* panels allow setting the amplitude of the output (up to 1.0) and the switch configuration (a 64-bit array as hexadecimal string) for channels A and B respectively. The string is parsed LSB first, meaning that a configuration of "00 00 00 00 00 00 00 01" will only fire the laser for the first pulse. The switch configuration can be inverted by clicking the *invert* button. A change in the value configures the Splitter only after clicking the *Apply* button. The *Undo* button restores the chosen Splitter's settings to settings at the moment when *Apply* was last clicked for that Splitter.

The module remembers the previous settings that were used for a Splitter and automatically configures the device on startup.

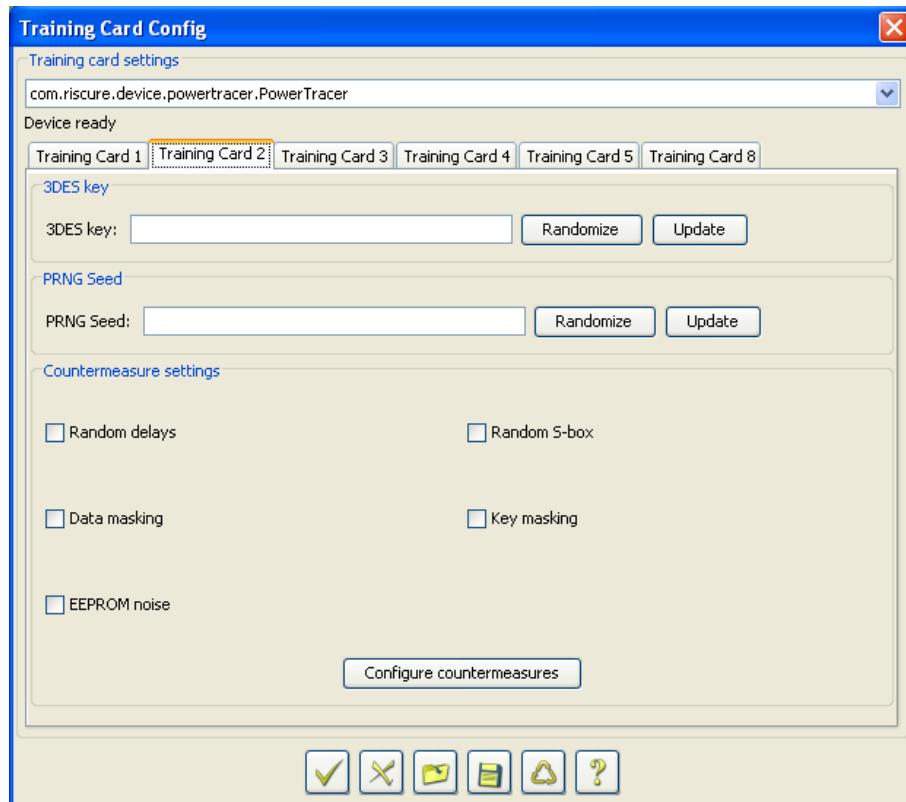
E.15.3 Training Card configuration

Version: SCA + FI**Purpose**

To configure the Riscure's training cards [..../tutorials/chaptertosamples.html] [PDF] [..../tutorials/Tutorials.pdf#chaptertosamples] that were provided for training purposes.

**Note**

This module is considered deprecated, since the training cards can be configured via the protocol classes since Inspector 4.6. It is still available via the "Classic" profile; NOT, however, in the "Full" profile.

Dialog**Input**

At the top of the dialog, a device panel allows the selection of the device used to configure the card. For these cards, one can use a PCSC reader or an Inspector Power Tracer.

Below this panel, a tabbed panel with several tabs is present. Each one of these tabs is used for the configuration of a particular kind of training card.

Each tab provides a simple panel for changing the keys used by the cards. This module is just an easy way to send the APDUs explained in the Description of training cards [..../tutorials/

[chaptertosamples.html](#) [PDF] [[..../tutorials/Tutorials.pdf#chaptertosamples](#)] chapter in the tutorials document.

For training cards 2 and 3, the module also provides a way to modify the countermeasures enabled in the card. The available countermeasures for training card 2 are:

- Random delays: Generate misalignment by means of randomly occurring delays.
- Random Sbox: Randomize the S-box look-up in every round
- Data masking: Mask the data throughout the algorithm, making it resistant to first order DPA, but not to second order DPA
- Key masking: Equivalent to the data masking, but for the key scheduling algorithm
- EEPROM noise: Adds amplitude noise coming from the EEPROM to the traces.

For training card 3 the following countermeasures are available:

- Random delays: Generate misalignment by means of randomly occurring delays.
- Random Sbox: Randomize the S-box look-up in every round
- Data masking: Mask the data throughout the algorithm, making it resistant to first order DPA, but not to second order DPA
- EEPROM noise: Adds amplitude noise coming from the EEPROM to the traces.

Note



It is possible that Inspector refuses to start if the card is inserted into a PCSC driver. Please remove the card from the reader before starting Inspector to solve the issue.

E.16 XYZ

This section contains the descriptions of the modules related to an XY scan.

E.16.1 AveragePlot

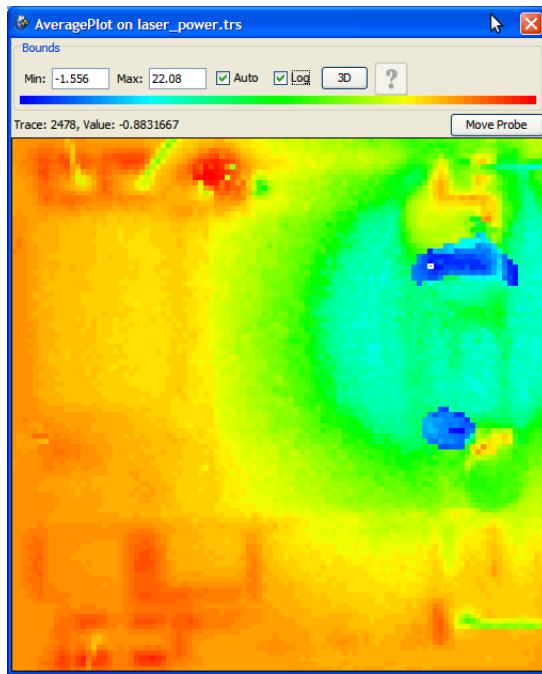
Version: F1

Purpose

The *AveragePlot* module can be used to analyse spatially obtained measurements. It plots for each spatial measurement the average value of the samples of the selected samples.

An example application is visualising a chip surface by stimulating different locations with a laser pulse. The chip's power usage is affected differently depending on the location. A trace set with power measurements can be input to this module.

Dialog



Input and results

Please refer to the SpectrallIntensity module for further explanation of the usage of this module.

E.16.2 SpectrallIntensity

Version: SCA

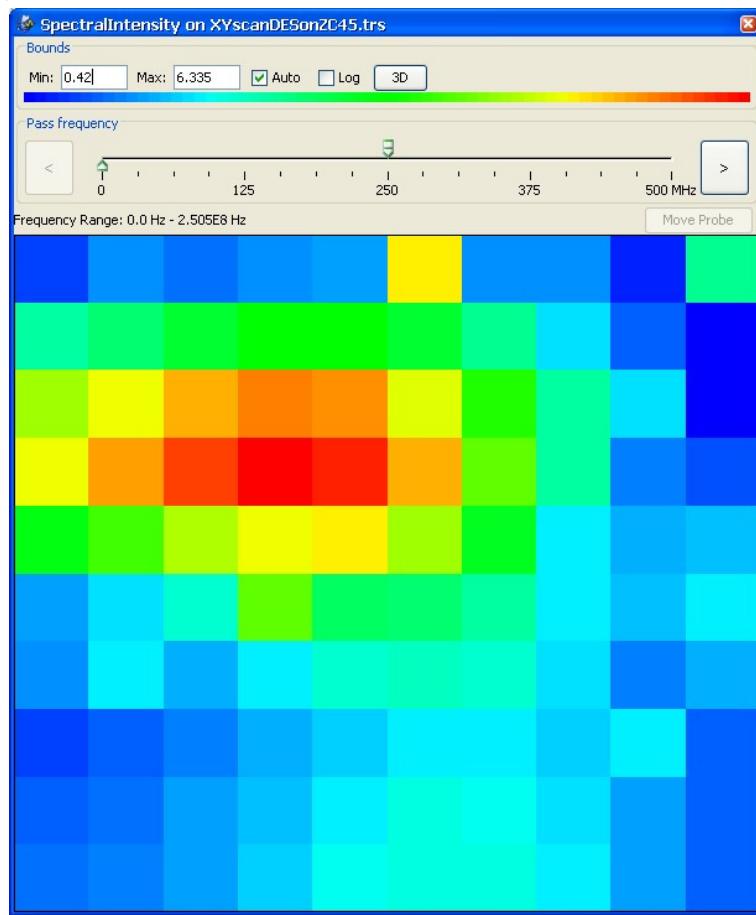
Purpose

The *SpectrallIntensity* module can be used to present and analyse the resulting trace set of an EM scan of a chip area which is performed using XYAcquisiton.

The module presents the Intensity of the measured signal at the X x Y grid positions and calculated for a selected frequency range. High intensity is indicated by a red square and low intensity is indicated by a blue square, with intermediate values indicated by the rainbow colors in between blue and red. The color scaling can be adapted by the user by setting the bounds. Also the selected frequency range can be tuned by the user.

The analyst may select the optimum position for the EM probe based on the analysis of the EM scan. With the EM probe at this location, the analyst can acquire the trace set that is necessary for SEMA or DEMA.

Dialog



Input

The input is grouped in two panels: ‘Bounds’ for the color scaling and ‘Pass frequency’ for the selection of the analyzed frequency range.

- The 3D button starts the 3D presentation of the spectral intensity levels. The level is indicated by the color and the height. Clicking on the 3D plot and moving your mouse will rotate the 3D plot. Also one of the grid point may be selected in the 3D plot.
- The Min field sets the level that corresponds to blue in the color plot
- The Max field sets the level that corresponds to red in the color plot
- The Auto check box toggles between manual and automatic scaling of the colors
- The Log check box toggles between linear and logarithmic scaling of the spectral intensity levels
- The Pass frequency sliders select the frequency range. The spectral intensity level is only plotted for this frequency range. There is low bound slider to set the lower bound of the frequency range, a high bound slider to set the upper bound of the frequency range and a total slider to shift the frequency range. The left and right arrows shift the frequency range down or up by the smallest possible frequency step.

- Selecting one of X x Y grid positions will high-light that corresponding color square and will pop-up the corresponding trace in the trace set window.
- After selection of a X x Y grid positions, the Move Probe button can be pressed to move the probe to the corresponding position.

Results

The *SpectralIntensity* module presents the intensity of the measured signal at the X x Y grid positions and calculated for a selected frequency range. High intensity is indicated by a red square and low intensity is indicated by a blue square, with intermediate values indicated by the rainbow colors in between blue and red. The color scaling can be adapted by the user by setting the bounds. Also the selected frequency range can be tuned by the user.

Pressing the 3D button will give a 3D presentation of the spectral intensity levels over the grid. This usually gives a better insight in the level differences than the 2D color plot.

F Hardware Drivers

This appendix describes the available hardware drivers in inspector.

F.1 I/O Devices

I/O devices allow Inspector to exchange information with the target at the application protocol level. An IO device implements a OSI layer-2 protocol, for example T=0 or T=1 for ISO-7816 smart cards.

F.1.1 PC/SC reader

F.1.1.1 Introduction

Inspector can access all PC/SC compatible card readers that are known to the Windows OS on which Inspector is installed.

F.1.1.2 Creation

Once plugged in the PC/SC reader is automatically recognized and added as a device in Inspector, assuming the appropriate PC/SC drivers are installed. Most PC/SC devices comply with the USB CCID standard for which drivers are included with Windows, therefore no additional drivers are required.

F.1.1.3 Configuration

No configuration options are available for PC/SC readers.

F.1.2 Power Tracer

F.1.2.1 Introduction

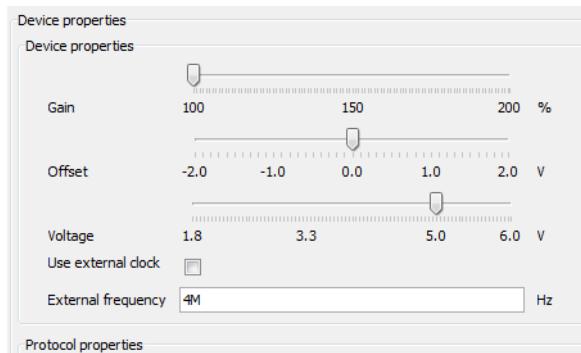
The Power Tracer driver controls the Riscure Power Tracer.

F.1.2.2 Creation

The Power Tracer is automatically detected when plugged into the system and will show up in the Hardware Manager after a delay of a few seconds.

F.1.2.3 Configuration

Figure F.1, “Power Tracer properties” shows the configuration dialog of the Power Tracer.

Figure F.1. Power Tracer properties

'Gain' and 'Offset' influence the power measurements before they are output. This allows the user to boost the signal or change the offset in order to reduce the range on the oscilloscope.

'Voltage' will adjust the voltage provided to the smart card. According to ISO 7816 the smart cards must initially operate at 5 V but can desire to continue operating at 3 or 1.8 V. Voltage control is entirely in the hands of the user, the Power Tracer will *not* adjust it automatically. Reducing the voltage to the lowest level at which the smart card still functions correctly improves the quality of the side channel measurement.

The Power Tracer provides a clock of exactly 4 MHz to the smart card. This makes it easier to filter the clock and its harmonics from the signal. However certain cards may not operate optimally at this clock frequency. Therefore it is possible to attach a custom clock generator to the 'clock input' on the Power Tracer. To enable the use of the externally provided clock check 'Use external clock'. In order for the Power Tracer to communicate with the smart card it needs to be aware of the exact frequency of the external clock. Therefore when using the external clock the frequency must be set in the 'External frequency' field.

The remaining configuration deals with the properties of the Section I.1, "ISO/IEC 7816" protocol. Please refer to that section for details.

The Power Tracer can generate a pulse immediately after the communication with the smart card which can be used by an oscilloscope as a trigger. Figure F.2, "Power Tracer pulse properties" shows the configuration properties for the pulse.

Figure F.2. Power Tracer pulse properties

Checking 'Enabled' will enable generation of the pulse after the command of interest has been send to the smart card. Setting 'Delay' will delay the pulse for the specified number of seconds. E.g. checking 'Enabled' and setting 'Delay' to 10m will result in a pulse 10 milliseconds after the command of interest has been send to the card.

F.1.3 Protocol Device

Most modules in Inspector require a device from the "I/O Device" category for target communications. For example, the PowerTracer is found in this category as it implements

the communications protocols T=0 and T=1 for its target (ISO7816). Devices that have no knowledge of the communications protocols used for the target are categorized as Raw I/O Devices. The Protocol Device combines a Raw I/O Device with a communications protocol to create an I/O Device that can be used with the acquisition modules.

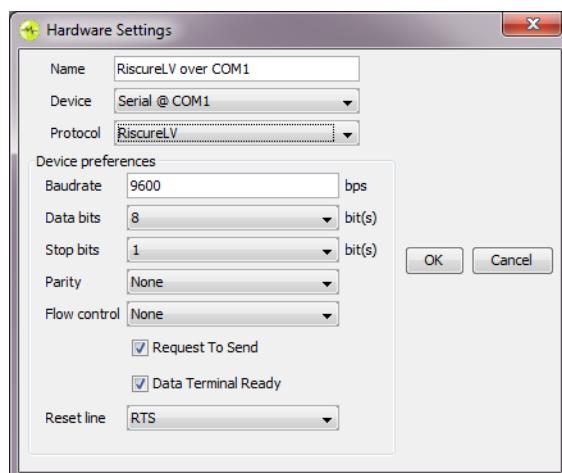
F.1.3.1 Setup and use

A Protocol Device is added using the Hardware Manager "Add Device" button. The steps mentioned here will create a protocol device that uses RiscureLV over the power tracer serial line.

- Open the hardware manager dialog and click "Add Device",
- Select "Protocol Device" from the drop-down menu,
- A dialog (Figure F.3, "Protocol device dialog") appears that allows you to enter a name for the device, a protocol and a device,
- For the name, enter a meaningful description, for example "RiscureLV over COM1",
- Select a Raw IO Device from the drop-down list, for example "Serial @ COM1",
- Select a Protocol, for example "RiscureLV". The dialog is extended with the parameters for the protocol,
- Select "OK".

A new device appears in the "I/O Devices" category. This device can now be used by the modules in Inspector.

Figure F.3. Protocol device dialog



F.2 Raw In/Output devices

Raw IO devices provide direct access to a hardware interface. Data send to and received from this interface is not interpreted by the device driver. Raw IO devices are combined with a transport protocol to construct an IO Device that can be used by Inspector.

F.2.1 Serial Port

F.2.1.1 Introduction

The serial port driver exposes both physical and virtual serial ports present in the system.

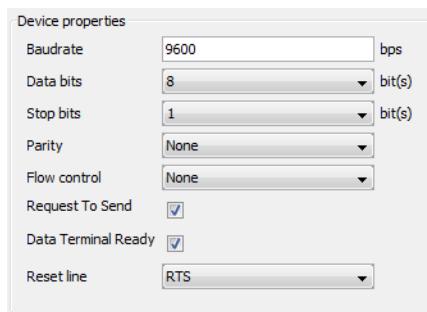
F.2.1.2 Creation

Serial ports are automatically added on start up. Virtual ports, such as USB-to-serial converters, add automatically added when plugged into the system.

F.2.1.3 Configuration

Figure F.4, “Serial port properties” shows the configuration dialog for a serial port.

Figure F.4. Serial port properties



'Baudrate', 'Data bits', 'Stop bits', and 'Parity' are standard configuration properties of a serial port. 'Flow control' can be set to 'None' or variations of 'RTS/CTS' or 'XON/XOFF'. The default value of the 'Request To Send' (RTS) and 'Data Terminal Ready' (DTR) can be set by using the checkboxes. Checking the box will mean the line is set to high on connect.

Finally certain serial devices allow for resetting the device by triggering the RTS or DTR line. 'Reset line' selects the appropriate one. When a reset is send the selected line will be inverted from its normal state as specified by the checkboxes above for a short period of time.

F.2.2 Serial card readers (e.g. SASEBO-W)

F.2.2.1 Introduction

Serial card readers connect the smart card directly to the serial port allowing for low level communication. Support for this kind of device is implemented in Inspector by combining a serial port with the ISO 7618 protocol.

F.2.2.2 Creation

See Protocol Device.

F.2.2.3 Configuration

The configuration of the serial port depends on the type of card reader used. However the most common configuration for the serial properties is:

Baudrate: 9600

Data bits: 8

Stop bits: 1

Parity: Odd

Flow control: None

Request To Send: On

Data Terminal Ready: On

Reset line: RTS

The ISO 7816 properties depend on the particular card used. However the ISO 7816 standard specifies a number of baudrates (combinations of F and D) which are often not supported by a serial port. In such a case the closest matching baudrate will be used. If the closest matching baudrate is not close enough communication with the card may contain errors or fail all together. In case of failure check the "Use first offered" option in protocol properties. This will attempt to communicate with the card in a compatibility mode which should work at the default baudrate of 9600 bps supported by all serial devices. Note that certain cards may not support the compatibility mode, if this is the case the Power Tracer should be used.

G Error Codes

This chapter describes Inspector error codes and possible solutions.

G.1 E03629

No templates generated

This error has the following known causes and solutions:

- Cause: The Covariance matrix is not invertible.

Solution: Increase the number of traces or switch to a different model.

- Cause: The Covariance matrix' elements are too large.

Solution: Switch to a different model.

- Cause: There is no noise in the selected traces.

Solution: (Co)variance will not work without noise, switch to Mean.

H Additional API information

This appendix describes additional information for APIs included in Inspector.

H.1 VC Glitcher API

The VCGlitcher is controlled using a perturbation program which is loaded by an Inspector perturbation module. This section describes all methods available on the `PerturbationProgram` class, see Section 8.3.6, “Writing perturbation modules” for more information on developing a perturbation module.

Registers

The VC Glitcher CPU holds four general-purpose registers. Each register is 32 bits wide. The names of the registers are: R0, R1, R2 and R3.

Data Memory

The VC Glitcher CPU has an embedded RAM of 1 KB .This memory is accessible using the `storeToMemory` and `loadFromMemory` instructions. A perturbation module can access this memory using the `readDataMemory` and `writeDataMemory` methods. The memory can be used to exchange parameters and results between the perturbation module and the perturbation program.

Counter

The VC Glitcher CPU has an embedded continuously running 32 bits counter. Every CPU clock cycle, the counter is incremented by one. The timer can be reset using the `resetCounter()` instruction and read using the `copyCounter()` instruction.

Using parameters

Some of the instructions support using Inspector parameters as arguments. This way a better integration with the GUI can be reached. For example, suppose the default `glitchCycles` parameter is initialized as:

```
glitchCycles = new Parameter(PARAM_GLITCH_CYCLES, "Glitch cycles 1", "The number of consecutive cycles to glitch", ParameterType.FIXED, ParameterFormat.INT, 1, 20);
```

From the GUI in the Perturbation window, this parameter can be set to be a random or a range between two values. The glitch program shall contain an instruction that uses that parameter:

```
p.glitchCycles(glitchCycles);
```

Once the program is executed, Inspector will supply the VC Glitcher with an appropriate value for the parameter, either the next value in the range or a random number.

Power down methods

The features described in this section are available since VC Glitcher version 1.1.

The VC Glitcher supports several methods for powering down the smart card.

- Direct power down
- Power down input
- Current limiter

First of all, the smart card can be powered down using the *powerDown* instruction. This will immediately power down the smart card.

The second method is to use the second trigger input. Note that this input has to be enabled using the *enableTriggerPowerDown* instruction. It can be disabled again using the *disableTriggerPowerDown* instruction. This method is extremely useful when combined with icWaves. For example, in order to prevent the smart card from writing EEPROM, icWaves can be programmed to trigger on a pattern that represents an EEPROM write.

The third method is to use the current limiter. Note that the current limiter has to be enabled using the *enableCurrentLimiter* instruction. It can be disabled again using the *disableCurrentLimiter* instruction.

H.1.1 Instruction set

This section lists all the supported instructions, together with a description of its effects, the list of the affected registers and the value of the Program Counter after the execution of the instruction. Furthermore the table specifies whether the instruction supports parameters: if so, its arguments can be replaced by parameters set by inspector. For more details on how to use parameters see the previous section.

add(Register r, int data)

Adds an 8-bits immediate to the register R_d . The result is stored into R_d .

| Operation | Operands | Program counter | Parameters supported |
|--------------------|---|-----------------|----------------------|
| $R_d = R_d + data$ | $0 \leq d \leq 3$ $0 \leq data \leq 255$ | $PC = PC + 1$ | No |

add(Register a, Register b, Register destination)

Adds register R_a to R_b . The result is stored into $R_{\text{destination}}$.

| Operation | Operands | Program counter | Parameters supported |
|--------------------------------------|---|-----------------|----------------------|
| $R_{\text{destination}} = R_a + R_b$ | $0 \leq \text{destination} \leq 3, 0 \leq a \leq 3,$ $0 \leq b \leq 3$ | $PC = PC + 1$ | No |

and(Register a, Register b, Register destination)

Performs a bitwise AND operation between R_a and R_b . The result is stored into $R_{\text{destination}}$.

| Operation | Operands | Program counter | Parameters supported |
|---------------------------------------|---|-----------------|----------------------|
| $R_{\text{destination}} = R_a \& R_b$ | $0 \leq \text{destination} \leq 3, 0 \leq a \leq 3,$ $0 \leq b \leq 3$ | $PC = PC + 1$ | No |

clearRegister(Register r)

Sets the content of the specified register to 0.

| Operation | Operands | Program counter | Parameters supported |
|------------------------------|------------------------------------|-----------------------------|----------------------|
| $R_{\text{destination}} = 0$ | $0 \leq \text{destination} \leq 3$ | $\text{PC} = \text{PC} + 1$ | No |

copyCounter(Register destination)

Copies the current value of the counter into $R_{\text{destination}}$.

| Operation | Operands | Program counter | Parameters supported |
|---|------------------------------------|-----------------------------|----------------------|
| $R_{\text{destination}} = \text{counter}$ | $0 \leq \text{destination} \leq 3$ | $\text{PC} = \text{PC} + 1$ | No |

end()

Ends the glitch program. No instructions after this will be processed. Note that this instruction is not mandatory: the compiler will automatically insert an *end* instruction in the end of the prepared glitch program.

| Operation | Operands | Program counter | Parameters supported |
|------------------|----------|--------------------|----------------------|
| Ends the program | None | The CPU is stopped | No |

glitch(boolean enabled)

Enables or disables the glitch pattern in the next clock cycle.

| Operation | Operands | Program counter | Parameters supported |
|-----------------------------------|---|-----------------------------|----------------------|
| Enable/disable the glitch pattern | $\text{enabled} = \text{true}$ to enable the glitch pattern. $\text{enabled} = \text{false}$ to disable the glitch pattern. | $\text{PC} = \text{PC} + 1$ | No |

glitchCycle()

Enables the glitch pattern for a single clock cycle.

| Operation | Operands | Program counter | Parameters supported |
|---|----------|-----------------------------|----------------------|
| Enables the glitch pattern for a single clock cycle | None | $\text{PC} = \text{PC} + 1$ | No |

glitchCycles(int numberOfCycles)

Enables the glitch pattern for the specified number of clock cycles.

| Operation | Operands | Program counter | Parameters supported |
|---|--|-----------------------------|----------------------|
| Enables the glitch pattern for the specified number of clock cycles | $0 < \text{numberOfCycles} < 4294967296$ | $\text{PC} = \text{PC} + 1$ | Yes |

jump(String label)

Sets the program counter to the address specified by *label*. If *label* is not defined in the glitch program, an exception will be thrown during compilation.

| Operation | Operands | Program counter | Parameters supported |
|--------------------------------|---|---------------------------------------|----------------------|
| Jumps to the specified address | Label = a label defined in the glitch program | $\text{PC} = \text{address of label}$ | No |

jumplfEqual(Register a, Register b, String label)

Sets the program counter to the address specified by *label* if R_a is equal to R_b . If *label* is not defined in the glitch program, an exception will be thrown during compilation.

| Operation | Operands | Program counter | Parameters supported |
|--|--|---|----------------------|
| Jumps to the specified address if $R_a == R_b$ | $0 \leq a \leq 3, 0 \leq b \leq 3$ Label = a label defined in the glitch program | PC = address of <i>label</i> if $R_a == R_b$. PC = PC + 1 otherwise. | No |

jumplfGreaterThan (Register a, Register b, String label)

Sets the program counter to the address specified by *label* if R_a is greater than R_b . If *label* is not defined in the glitch program, an exception will be thrown during compilation.

| Operation | Operands | Program counter | Parameters supported |
|---|--|--|----------------------|
| Jumps to the specified address if $R_a > R_b$ | $0 \leq a \leq 3, 0 \leq b \leq 3$ Label = a label defined in the glitch program | PC = address of <i>label</i> if $R_a > R_b$. PC = PC + 1 otherwise. | No |

jumplfGreaterThanOrEqual (Register a, Register b, String label)

Sets the program counter to the address specified by *label* if R_a is greater than or equal to R_b . If *label* is not defined in the glitch program, an exception will be thrown during compilation.

| Operation | Operands | Program counter | Parameters supported |
|--|--|---|----------------------|
| Jumps to the specified address if $R_a \geq R_b$ | $0 \leq a \leq 3, 0 \leq b \leq 3$ Label = a label defined in the glitch program | PC = address of <i>label</i> if $R_a \geq R_b$. PC = PC + 1 otherwise. | No |

jumplfLessThan (Register a, Register b, String label)

Sets the program counter to the address specified by *label* if R_a is less than R_b . If *label* is not defined in the glitch program, an exception will be thrown during compilation.

| Operation | Operands | Program counter | Parameters supported |
|---|--|--|----------------------|
| Jumps to the specified address if $R_a < R_b$ | $0 \leq a \leq 3, 0 \leq b \leq 3$ Label = a label defined in the glitch program | PC = address of <i>label</i> if $R_a < R_b$. PC = PC + 1 otherwise. | No |

jumplfLessThanOrEqual (Register a, Register b, String label)

Sets the program counter to the address specified by *label* if R_a is less than or equal to R_b . If *label* is not defined in the glitch program, an exception will be thrown during compilation.

| Operation | Operands | Program counter | Parameters supported |
|--|--|---|----------------------|
| Jumps to the specified address if $R_a \leq R_b$ | $0 \leq a \leq 3, 0 \leq b \leq 3$ Label = a label defined in the glitch program | PC = address of <i>label</i> if $R_a \leq R_b$. PC = PC + 1 otherwise. | No |

jumplfNotEqual(Register a, Register b, String label)

Sets the program counter to the address specified by *label* if R_a is not equal to R_b . If *label* is not defined in the glitch program, an exception will be thrown during compilation.

| Operation | Operands | Program counter | Parameters supported |
|--|--|---|----------------------|
| Jumps to the specified address if $R_a \neq R_b$ | $0 \leq a \leq 3, 0 \leq b \leq 3$ Label = a label defined in the glitch program | PC = address of <i>label</i> if $R_a \neq R_b$, PC = PC + 1 otherwise. | No |

jumpIfNotZero(Register a, String label)

Sets the program counter to the address specified by *label* if R_a is not equal to 0. If *label* is not defined in the glitch program, an exception will be thrown during compilation.

| Operation | Operands | Program counter | Parameters supported |
|--|---|---|----------------------|
| Jumps to the specified address if $R_a \neq 0$ | $0 \leq a \leq 3$ Label = a label defined in the glitch program | PC = address of <i>label</i> if $R_a \neq 0$, PC = PC + 1 otherwise. | No |

jumpIfZero(Register a, String label)

Sets the program counter to the address specified by *label* if R_a is equal to 0. If *label* is not defined in the glitch program, an exception will be thrown during compilation.

| Operation | Operands | Program counter | Parameters supported |
|--|---|---|----------------------|
| Jumps to the specified address if $R_a == 0$ | $0 \leq a \leq 3$ Label = a label defined in the glitch program | PC = address of <i>label</i> if $R_a == 0$, PC = PC + 1 otherwise. | No |

label(String label)

The label instruction is a pseudo-instruction which inserts a label at the current address. The program code will not be modified.

| Operation | Operands | Program counter | Parameters supported |
|-------------------------------------|---|-----------------|----------------------|
| Sets a label at the current address | Label = the name of the label to be defined | Not modified | No |

load(int value, Register destination)

Loads an immediate integer value into a register.

| Operation | Operands | Program counter | Parameters supported |
|---------------------------|-----------------------------|-----------------|----------------------|
| $R_{destination} = value$ | $0 \leq destination \leq 3$ | PC = PC + 1 | No |

loadFromMemory(byte address, Register destination)

Loads the 32-bits long data at the specified data memory address into the specified register.

| Operation | Operands | Program counter | Parameters supported |
|-----------------------------------|-----------------------------|-----------------|----------------------|
| $R_{destination} = data[address]$ | $0 \leq destination \leq 3$ | PC = PC + 1 | No |

loadFromMemory(Register source, Register destination)

Loads a value in the data memory pointed by a register into another register.

| Operation | Operands | Program counter | Parameters supported |
|--------------------------------------|---|-----------------|----------------------|
| $R_{destination} = data[R_{source}]$ | $0 \leq destination \leq 3, 0 \leq source \leq 3, 0 \leq R_{source} \leq 255$ | PC = PC + 1 | No |

loadParameter(Parameter p, Register destination)

Load Inspector parameter in the specified register. Internally Inspector assigns an address to each parameter.

| Operation | Operands | Program counter | Parameters supported |
|---|------------------------------------|-----------------------------|----------------------|
| $R_{\text{destination}} = \text{data}[R_{\text{par. address}}]$ | $0 \leq \text{destination} \leq 3$ | $\text{PC} = \text{PC} + 1$ | Yes |

move(Register source, Register destination)

Moves a value from one register to another.

| Operation | Operands | Program counter | Parameters supported |
|--|--|-----------------------------|----------------------|
| $R_{\text{destination}} = R_{\text{source}}$ | $0 \leq \text{destination} \leq 3$ $0 \leq \text{source} \leq 3$ | $\text{PC} = \text{PC} + 1$ | No |

nop()

No operation performed.

| Operation | Operands | Program counter | Parameters supported |
|-----------|----------|-----------------------------|----------------------|
| None | None | $\text{PC} = \text{PC} + 1$ | No |

not(Register source, Register destination)

Performs a bitwise negation of a register.

| Operation | Operands | Program counter | Parameters supported |
|---|--|-----------------------------|----------------------|
| $R_{\text{destination}} = \sim R_{\text{source}}$ | $0 \leq \text{destination} \leq 3$ $0 \leq \text{source} \leq 3$ | $\text{PC} = \text{PC} + 1$ | No |

or(Register a, Register b, Register destination)

Performs a bitwise OR of two registers.

| Operation | Operands | Program counter | Parameters supported |
|---|---|-----------------------------|----------------------|
| $R_{\text{destination}} = R_a \mid R_b$ | $0 \leq \text{destination} \leq 3$ $0 \leq a \leq 3$, $0 \leq b \leq 3$ | $\text{PC} = \text{PC} + 1$ | No |

disableCurrentLimiter()

Disable the current limiter. This will make sure the smart card is not powered down when the current limiter reached its threshold.

| Operation | Operands | Program counter | Parameters supported |
|------------------------------|----------|-----------------------------|----------------------|
| Disable the current limiter. | None | $\text{PC} = \text{PC} + 1$ | No |

disableTriggerPowerDown()

Disable trigger power down. This will make sure the smart card is not powered down on a trigger pulse.

| Operation | Operands | Program counter | Parameters supported |
|---------------------------------------|----------|-----------------|----------------------|
| Disable the power down trigger input. | None | PC = PC + 1 | No |

enableCurrentLimiter()

Enable the current limiter. This will make sure the smart card is powered down when the current limiter reached its threshold.

| Operation | Operands | Program counter | Parameters supported |
|-----------------------------|----------|-----------------|----------------------|
| Enable the current limiter. | None | PC = PC + 1 | No |

enableTriggerPowerDown()

Enable trigger power down. This will make sure the smart card is powered down on a trigger pulse.

| Operation | Operands | Program counter | Parameters supported |
|--------------------------------------|----------|-----------------|----------------------|
| Enable the power down trigger input. | None | PC = PC + 1 | No |

powerDown()

Power down. This will enable the power down of the smart card immediately.

| Operation | Operands | Program counter | Parameters supported |
|-------------------------|----------|-----------------|----------------------|
| Power down immediately. | None | PC = PC + 1 | No |

powerDownIfNotTriggered(int timeout, Register res)

Power down the smart card if a trigger is not received within a specified amount of time. Note that the hardware counter value will be modified after the execution of this instruction. The register "res" will be set to "1" if the trigger has been received, or "0" otherwise.

| Operation | Operands | Program counter | Parameters supported |
|--|--|-----------------|----------------------|
| Wait for a trigger and power down if the trigger is not received | 0 ≤ timeout ≤ 4294967295 (in us) 0 ≤ res ≤ 3 | PC = PC + 1 | No |

receive(Register destination)

Receives a byte from the smart card and stores it in the specified register. This instruction is blocking, therefore it will not return until a byte is received.

| Operation | Operands | Program counter | Parameters supported |
|--|---------------------|-----------------|----------------------|
| R _{destination} = received byte from smart card | 0 ≤ destination ≤ 3 | PC = PC + 1 | No |

receiveAnswer(Command command,int getResponse)

Receives the answer to the specified command. The command must have been previously sent by using the sendCommand instruction. getResponse can either hold the value PerturbationProgram. GET_RESPONSE (an additional GET_RESPONSE command with

CLA = APDU class will be sent in case the communication protocol is T=0 and the smart card signals it has some data ready to be sent) or PerturbationProgram. NO_RESPONSE (no additional commands will be sent). The received data will not be stored anywhere, but it will be logged and reported in the Inspector report.

| Operation | Operands | Program counter | Parameters supported |
|--|---|-----------------|----------------------|
| Receives the answer to a previously sent command | Command: a previously sent command. getResponse = PerturbationProgram. GET_RESPONSE or PerturbationProgram. NO_RESPONSE | PC = PC + 1 | No |

receiveAnswer(Command command,int getResponse, int getResponseClass)

Like `receiveAnswer(Command command,int getResponse)`, but it allows to specify the value of the CLA byte to be sent in the GET_RESPONSE command. This might be needed for some applications that require a specific CLA byte for the GET_RESPONSE APDU. For example, `receiveAnswer(readCommand,PerturbationProgram. GET_RESPONSE, 0)` will use 00 as CLA for the GET_RESPONSE command, instead of using the same CLA byte as `readCommand`.

| Operation | Operands | Program counter | Parameters supported |
|--|--|-----------------|----------------------|
| Receives the answer to a previously sent command | Command: a previously sent command. getResponse = PerturbationProgram. GET_RESPONSE or PerturbationProgram. NO_RESPONSE getResponseClass = the CLA byte to be used in the GET_RESPONSE command. | PC = PC + 1 | No |

receiveATR()

Resets the smart card and receives the ATR. Once the ATR has been received, the appropriate communication parameters (ETU, convention, protocol) will be set automatically.

| Operation | Operands | Program counter | Parameters supported |
|--|----------|-----------------|----------------------|
| Resets the smart card and receives the ATR | None | PC = PC + 1 | No |

receiveBytes(int count)

Receives the specified number of bytes from the smart card. The received data will not be stored in a register or data memory, but it will be logged and reported in the Inspector report.

| Operation | Operands | Program counter | Parameters supported |
|--|------------------------|-----------------|----------------------|
| Receives the specified number of bytes | 0 < count < 4294967296 | PC = PC + 1 | Yes |

resetCounter()

Resets the CPU internal counter to 0.

| Operation | Operands | Program counter | Parameters supported |
|--------------------------------------|----------|-----------------|----------------------|
| Resets the CPU internal counter to 0 | None | PC = PC + 1 | No |

resetSmartCard()

Resets the smart card using a pulse (800 clock cycles) on the smart card reset line.

| Operation | Operands | Program counter | Parameters supported |
|-----------------------|----------|-----------------|----------------------|
| Resets the smart card | None | PC = PC + 1 | No |

send(byte data)

Sends the specified byte to the smart card. This call is non-blocking, thus if the program needs to wait for the end of the transmission before performing further actions, a `waitForTxBusy(false)` instruction must be executed afterwards (see Section4.6.59). Note that this function first sends two stop bits, followed by a start bit, 8 data bits and the parity bit. This is done to return control to the perturbation program as soon as possible.

| Operation | Operands | Program counter | Parameters supported |
|--------------------------------|-------------------------------|-----------------|----------------------|
| Sends a byte to the smart card | $0 \leq \text{data} \leq 255$ | PC = PC + 1 | No |

send(Register source)

Sends the least significant byte of the specified register to the smart card. This call is non-blocking, thus if the program needs to wait for the end of the transmission before performing further actions, a `waitForTxBusy(false)` instruction must be executed afterwards (see Section4.6.59). Note that this function first sends two stop bits, followed by a start bit, 8 data bits and the parity bit. This is done to return control to the perturbation program as soon as possible.

| Operation | Operands | Program counter | Parameters supported |
|---|-------------------------------|-----------------|----------------------|
| Sends $R_{\text{source}} \& 0xFF$ to the smart card | $0 \leq \text{source} \leq 3$ | PC = PC + 1 | No |

sendCommand(Command command)

Sends the specified command to the smart card.

| Operation | Operands | Program counter | Parameters supported |
|---|---------------------------------------|-----------------|----------------------|
| Sends the specified command to the smart card | Command: a previously defined command | PC = PC + 1 | No |

setBaudrate(int baudrate)

Sets the specified baud rate (specified in bps) for the smart card communication. This command is not usually needed as the VC Glitcher automatically detects the right communication speed.

| Operation | Operands | Program counter | Parameters supported |
|-----------------------------|-----------------------------------|-----------------|----------------------|
| Sets the specified baudrate | $0 < \text{baudrate} \leq 115200$ | PC = PC + 1 | No |

setDirectConvention()

Set the VC Glitcher to use direct convention for smart card communication. This command is not usually needed as the VC Glitcher automatically detects the right convention.

| Operation | Operands | Program counter | Parameters supported |
|---|----------|-----------------|----------------------|
| Sets the VC Glitcher to use direct convention | None | PC = PC + 1 | No |

setInverseConvention()

Set the VC Glitcher to use inverse convention for smart card communication. This command is not usually needed as the VC Glitcher automatically detects the right convention.

| Operation | Operands | Program counter | Parameters supported |
|--|----------|-----------------|----------------------|
| Sets the VC Glitcher to use inverse convention | None | PC = PC + 1 | No |

setSmartcardResetLine(boolean value)

Sets the smart card reset line. If value is set to false then the smart card is held in reset, otherwise it is functioning normally.

| Operation | Operands | Program counter | Parameters supported |
|--------------------------------|---|-----------------|----------------------|
| Sets the smart card reset line | value = <i>false</i> for reset value = <i>true</i> for normal operation | PC = PC + 1 | No |

setTriggerOutput(boolean value)

Sets the trigger output of the VC Glitcher. If value is set to false then output is set to 0V, otherwise it is set to a +3.3V.

| Operation | Operands | Program counter | Parameters supported |
|--------------------------------|---|-----------------|----------------------|
| Sets the smart card reset line | value = <i>false</i> for reset value = <i>true</i> for normal operation | PC = PC + 1 | No |

shiftLeft(Register destination, int numberOfBits)

Shifts the specified register left the specified number of bits.

| Operation | Operands | Program counter | Parameters supported |
|---|--|-----------------|----------------------|
| $R_{destination} = R_{destination} \ll$ <i>numberOfBits</i> | $0 \leq destination \leq 3$ $0 \leq$ <i>numberOfBits</i> ≤ 31 | PC = PC + 1 | No |

shiftRight(Register destination, int numberOfBits)

Shifts the specified register right the specified number of bits.

| Operation | Operands | Program counter | Parameters supported |
|---|--|-----------------|----------------------|
| $R_{destination} = R_{destination} \gg$ <i>numberOfBits</i> | $0 \leq destination \leq 3$ $0 \leq$ <i>numberOfBits</i> ≤ 31 | PC = PC + 1 | No |

sleep(int us)

Causes the program to sleep the specified number of microseconds.

| Operation | Operands | Program counter | Parameters supported |
|---|---------------------------|-----------------|----------------------|
| Causes the program to sleep the specified number of microseconds. | $0 \leq us \leq 85899345$ | $PC = PC + 1$ | No |

storeToMemory(byte address, Register source)

Store the contents of a register in the data memory at the specified address.

| Operation | Operands | Program counter | Parameters supported |
|------------------------------|--|-----------------|----------------------|
| $data[address] = R_{source}$ | $0 \leq source \leq 3 \quad 0 \leq address \leq 256$ | $PC = PC + 1$ | No |

storeToMemory(Register destination, Register source)

Store the data from a register in the data memory at the address pointed by another register.

| Operation | Operands | Program counter | Parameters supported |
|--------------------------------------|--|-----------------|----------------------|
| $data[R_{destination}] = R_{source}$ | $0 \leq source \leq 3 \quad 0 \leq destination \leq 3$ | $PC = PC + 1$ | No |

subtract(Register r, int data)

Subtracts an 8-bits immediate to the register R_d . The result is stored into R_d .

| Operation | Operands | Program counter | Parameters supported |
|--------------------|--|-----------------|----------------------|
| $R_d = R_d - data$ | $0 \leq d \leq 3 \quad 0 \leq data \leq 255$ | $PC = PC + 1$ | No |

subtract(Register a, Register b, Register destination)

Subtracts register R_b to R_a . The result is stored into $R_{destination}$.

| Operation | Operands | Program counter | Parameters supported |
|-------------------------------|---|-----------------|----------------------|
| $R_{destination} = R_a - R_b$ | $0 \leq destination \leq 3, 0 \leq a \leq 3, 0 \leq b \leq 3$ | $PC = PC + 1$ | No |

synchronize()

Synchronizes the CPU with the pattern generators. This is useful to synchronize the CPU with the smart card.

| Operation | Operands | Program counter | Parameters supported |
|--|----------|-----------------|----------------------|
| Synchronizes the CPU with the pattern generators | None | $PC = PC + 1$ | No |

trigger()

Generates a rising edge on the *Trigger out* SMB connector. The falling edge will be generated after each perturbation run.

| Operation | Operands | Program counter | Parameters supported |
|---|----------|-----------------|----------------------|
| Generates a rising edge on the trigger output | None | PC = PC + 1 | No |

waitForTrigger(boolean value)

Waits for a specific value in the trigger input signal. If *value* is set to *false*, the program will wait for a low signal. If *value* is set to *true*, the program will wait for a high signal.

| Operation | Operands | Program counter | Parameters supported |
|--|--|-----------------|----------------------|
| Waits for a specific value in the trigger input signal | value = <i>false</i> for low signal value = <i>true</i> for high signal | PC = PC + 1 | Yes |

waitForTimer(Register a)

Waits for the number of CPU clock cycles specified in the register R_a .

| Operation | Operands | Program counter | Parameters supported |
|--|--|-----------------|----------------------|
| Waits for the number of CPU clock cycles | $0 \leq a \leq 3$ $0 < R_a < 4294967296$ | PC = PC + 1 | No |

waitForTxBusy(boolean value)

Waits for a specific state in the Tx line. If *value* is set to *false*, the program will wait until a transmission ends. If *value* is set to *true*, the program will wait until a transmission starts.

| Operation | Operands | Program counter | Parameters supported |
|---|---|-----------------|----------------------|
| Waits for a specific state in the Tx line | value = <i>false</i> for end of transmit value = <i>true</i> for start of transmit | PC = PC + 1 | No |

waitForTxSignal(boolean value)

Wait for a specific state of the transmit signal (i.e. the transmit line to the smart card).

| Operation | Operands | Program counter | Parameters supported |
|------------------------------|--|-----------------|----------------------|
| Wait for the transmit signal | value = <i>false</i> for the transmission of a "0" value = <i>true</i> for the transmission of a "1" | PC = PC + 1 | No |

xor(Register a, Register b, Register destination)

Performs a bitwise XOR between registers R_b and R_a . The result is stored into $R_{\text{destination}}$.

| Operation | Operands | Program counter | Parameters supported |
|---|---|-----------------------------|----------------------|
| $R_{\text{destination}} = R_a \wedge R_b$ | $0 \leq \text{destination} \leq 3, 0 \leq a \leq 3,$ $0 \leq b \leq 3$ | $\text{PC} = \text{PC} + 1$ | No |

I Transport Protocols

This section describes the available transport protocols in Inspector.

I.1 ISO/IEC 7816

I.1.1 Introduction

ISO/IEC 7816 protocol is used primarily in contact smartcards. This section is mainly concerned with the transport part of the protocol as specified in part 3 of the specification. The protocol supports the third edition as published on November 1, 2006.

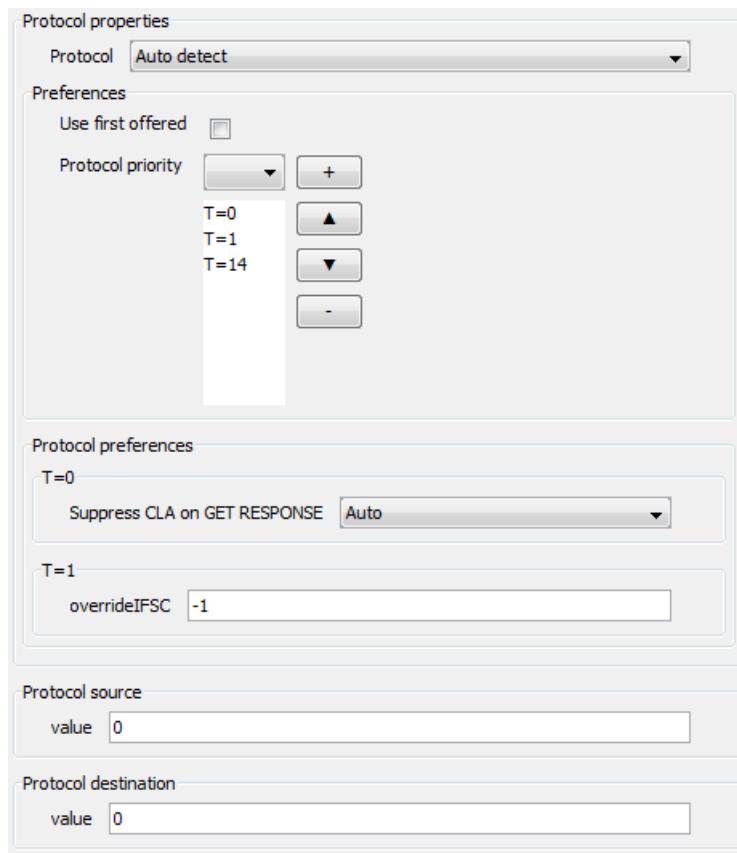
The standard specifies two sub protocols, one called T=0, which is character based, the other T=1, which is block based. Both sub protocols are implemented. In addition another sub protocol referred to as PPS is implemented to configure smartcards which support multiple sub protocols.

I.1.2 Implementation details

The implementation aims to follow the standard very strictly for transmitting data and attempts to be lenient when receiving data. However a significant number of different interpretation of the standard exist in existing implementations. In order to support as many smartcards as possible this implementation can be configured to behave in a way that other interpretations of the standard expect. The default, however, is to strictly follow the standard unless it can be unequivocally determined that the smart card expects a non strict behaviour, in that case the implementation is automatically configured to meet the expectations of the card. In all other cases the user will have to manually configure any deviations from the strict interpretation of the standard.

I.1.3 Configuration

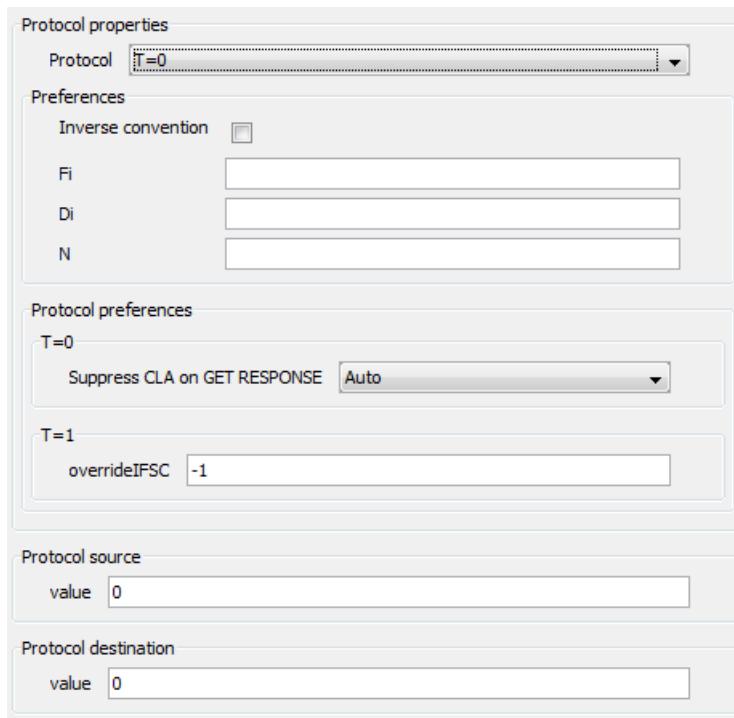
Figure I.1, “ISO 7816 Properties” shows the configuration dialog for the ISO 7816 protocol stack.

Figure I.1. ISO 7816 Properties

Protocol allows you to select the sub protocol to use. 'Auto detect' will detect the protocol(s) supported by the card and choose one based on the configuration specified by the next few items. 'Use first offered' will select the first protocol the card specifies it can support. If this option is unchecked the priority will be determined based on the order of the protocols in the 'Protocol priority' list. Protocols higher in the list will be attempted first.

Note that T=14 is specified by the standard as a sub protocol which is not standardized. Neither does the stack support T=14. If the user wants to use T=14 he will have to provide an implementation himself.

Figure I.2, "ISO 7816 Specific protocol" shows the dialog when a specific protocol is selected.

Figure I.2. ISO 7816 Specific protocol

When a specific protocol is selected all configuration parameters by the card are ignored and the ones provided in the dialog are used. If no values are provided the defaults, as specified by the standard are used.

Below the 'Preferences' box, regardless of whether auto detect or a specific protocol is selected, a number of 'Protocol preferences' appear. These are the preferences for the available sub protocols, providing the sub protocols have options to configure.

'Suppress CLA on GET RESPONSE' allows the protocol to work with a specific deviation in certain T=0 implementations. According to section 12.2.5 of the ISO standard in case of a case 4 APDU the GET RESPONSE APDU must contain the CLA byte of the original request. However certain cards expect the CLA byte to be set to zero. When this setting is set to 'Auto' the protocol will exhibit the specified behaviour unless a card is detected, based on historical bytes of known cards with this deviation. When set to 'Yes' this deviation will always be enabled. When set to 'No' the strict interpretation will always be used even if cards are detected which will only work with the deviation.

'Override IFSC' allows the user to override the default IFSC setting. If set to -1 the default will be used.

'Protocol source' and 'Protocol destination' allow the user to specify a value between 0 and 7 which will indicate the source node address (SAD) and destination node address (DAD) respectively. These values have no meaning in T=0.

I.2 RiscureLV

This is the embedded protocol that is used by Inspector for Embedded devices. All encoding is in network-byte order (MSB first)

I.2.1 Command packet

The protocol exists of command and response packets. The command packet has a structure of a 1 byte tag, a 2 byte length field, and optional data (0-65535 bytes). The tag encodes the command, and the length is a 16-bit unsigned integer containing the length of the data field.

Tags are defined by the application protocol, for example embeddedDES or embeddedAES.

I.2.2 Response packet

To each command packet, the target will respond with a response packet. Each packet is prefixed with 2 byte header containing the length of the remainder of the packet.

I.2.3 Example flow

This is an example command-response sequence of the RiscureLV protocol

| | |
|--------------|--|
| Inspector => | 01 00 10 11 22 33 44 55 66 77 88 99 00 AA BB CC DD EE FF |
| | T Len Value |
| Target <= | 00 01 00 |
| | Len data |

J Application Protocols

This section describes the available application protocols in Inspector.

J.1 EMV

The EMV protocol implemented in Inspector adheres to EMV specification version 4.2. The current implementation does not support digital signatures (SDA, ODA/DDA, CDA). Information provided by the card is used to feed equivalent fields that are considered to be terminal resident, for example information related to the currency of the transaction.

There are different methods of selecting an EMV application. The default method is to select the first EMV application that is declared by the Payment System Directory information. If this is undesired, a name or a RID/ AID can be specified. The name is case-sensitive and must be identical to the names found in the Payment System Directory.

If selection using PSE is not supported, the module will resort to Static AID selection. In this case, the AID or RID must be specified.

Quick Start feature

The standard specifies that the terminal should read all application data prior to beginning a transaction. Assuming that the relevant parts of the application data are not changing between calls (i.e. static information), this procedure may be skipped on subsequent calls to transactions. The "Quick Start" feature (enabled by default) selects the payment application immediately after reset and resumes the transaction flow on "offline data authentication".

Cardholder Verification Method

CVM selection is based on the rules specified by the application and what the implementation supports. If no PIN is provided, the implementation will claim not to support PIN CVM. The first rule that can be supported by the implementation will be used. The implementation will always support SIGNATURE/PAPER methods, if supported by the application.

Please note that CVM verification failure causes an error in Inspector. If the error handling in Inspector is set to restart the acquisition on errors, the CVM method may block due to multiple CVM failures.

J.2 GlobalPlatform SCP

The GlobalPlatform SCP is the implementation of the Secure Channel Protocol versions 01 and 02, according to the GlobalPlatform Card Specification version 2.2. This application protocol can be used for analysis of the setup of a secure channel using SCP01 or SCP02. Two modes of operation are supported: using random data or using a known key to complete the setup of a Secure Channel. All configurable options for Secure Channel Protocols 01 and 02 are available in the settings dialog of this protocol.

Please note that selection of SCP01 or SCP02 is performed by the target. It can be implicitly selected by using key version and key identity values that are linked to SCP02 keys. This is specific to the target configuration.

J.3 MRTD

Machine Readable Travel Documents are standardized by the International Civil Aviation Organisation (ICAO). In these standards the protocols *Active Authentication* (AA) and *Extended Access Control (Chip Authentication)* (EAC CA) are defined.

Both AA and EAC run only after the Basic Access Control (BAC) protocol is run. After BAC the passport is assured that the terminal has knowledge of some printed details on the passport, being the passport number, the date of birth of the holder and the passport's expiration date (these three parameters must be configured in the settings GUI in order to run the MRTD protocols). An additional result of BAC are session keys that used to protect the integrity and confidentiality of all following data exchange (i.e. a secure channel).

The implementation of the MRTD protocols in Inspector is based on an open source implementation of MRTD in Java, JMRTD [<http://jmrtd.org>].

J.3.1 Active Authentication

Active Authentication is described in MACHINE READABLE TRAVEL DOCUMENTS, PKI for Machine Readable Travel Documents offering ICC Read-Only Access, Version - 1.1 Date - October 01, 2004.

An excerpt from Appendix D2 reads the following: *Active Authentication is performed using the ISO7816 INTERNAL AUTHENTICATE command. The input is a nonce (RND.IFD) that MUST be 8 bytes. The ICC computes a signature, when an integer factorization based mechanism is used, according to ISO9796-2 Digital Signature scheme 1 ([R17], ISO/IEC 9796-2, Information Technology – Security Techniques – Digital Signature Schemes giving message recovery – Part 2: Integer factorisation based mechanisms, 2002.).*

In short, in Active Authentication a private RSA key in the passport is used to sign data composed of random data generated by the passport and data offered by the terminal.

The data that can be retrieved after each run of the protocol is the RSA encrypted output and the plaintext input.

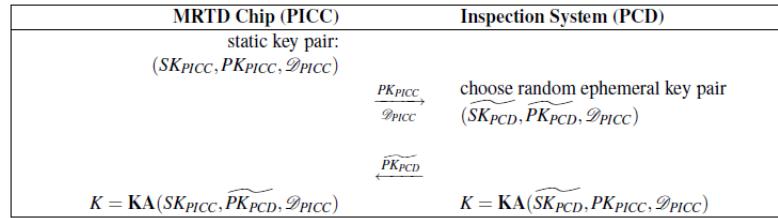
J.3.2 Extended Access Control

Extended Access Control is described in Technical Guideline TR-03110 Advanced Security Mechanisms for Machine Readable Travel Documents – Extended Access Control (EAC) Version 1.11.

Like Active Authentication, EAC is mapped onto ISO7816 APDUs. EAC consists of two parts, Chip Authentication (CA) and Terminal Authentication (TA). Protocol EAC CA is an alternative to AA and allows a terminal to verify it is communicating with an authentic passport. Protocol EAC TA allows a passport to verify it is communicating with an authentic terminal. For the latter protocol no secret key material stored on the passport is used, and therefore this protocol is not implemented.

J.3.2.1 Chip Authentication

Chip Authentication is based on Diffie-Hellman with optional ECC. The protocol flow is displayed in Figure J.1, “EAC Chip Authentication” below, taken from Figure 3.1 in the protocol specification document.

Figure J.1. EAC Chip Authentication

The passport sends the terminal its public key and domain parameters. Based on these parameters the terminal computes random ephemeral keys and returns the ephemeral public component to the passport. From the ephemeral data the passport and the terminal can now both compute a shared secret. From this secret new sessions keys for the secure channels are computed. The session keys computed during BAC are replaced!

In addition, to assure the terminal communicates with an authentic chip, the terminal must verify the chain of trust for the public key of the passport it received in the first step of the protocol. Note that our implementation of the protocol does not perform such verification.

The data that can be retrieved after each run of the protocol are the ephemeral public keys and the emphemeral private key of the terminal.

J.4 OTA (SCP80)

OTAProtocol is the implementation for sending SCP80 datagrams to a SAT/UICC enabled smartcard using SMS-PP. SCP80 is the secure channel protocol as defined in the ETSI TS102.225 and TS131.115 (3GP 31.115) standards.

The term 'SCP80' is designated by GlobalPlatform, and '03.48' is also often used to describe the SCP80 protocol.

A SCP80 datagram cannot be offered as such to the smart card. It has to be offered as if a text message was received by the ME (cellphone) and forwarded to the UICC/USIM (target). In order to simulate this event, the SCP80 packet needs to be wrapped in an SMS-TPDU (see ETSI TS123.040). The SMS-TPDU then needs to be wrapped in an ENVELOPE APDU (TS102-223, TS131-111). This ENVELOPE APDU is what is transmitted to the target.

The OTAProtocol implementation in Inspector provides an exhaustive number of configuration items for all the fields present in the APDU.

There are some restrictions in the current implementation of the OTA Protocol:

- The SMS messages constructed here do not support 'concatenation control', i.e. the SCP80 datagram should not exceed the 140 bytes.
- The SCP80 stack itself is not implemented, i.e. no encryption and no MACing is performed.
- The smart card response data is not analyzed.

J.5 Training card protocols

Description

This Riscure-proprietary protocol is meant to trigger various cryptographic operations (AES, DES, RSA, etc) on Riscure training cards. It sends randomized input data to execute a cryptographic primitive on, and receives the processed data from the card.

Settings

- **Key:** Optional field for programming the card key. If left empty, no key is programmed.

Phases

- **Select:** Application selection.
- **SetKey:** Setting the key.
- **Crypto:** Running the crypto.

Output fields

- **Input:** Input to the crypto.
- **Output:** Output after crypto.

J.6 SIM/USIM Authentication

The USIM Protocol is an implementation of the GSM / 3G network authentication protocols, specifically TS 151.011 (previous GSM11.11) and TS 131.102 (release 8). The protocol first attempt to initialize the USIM application according to the TS131.102 standard. If this does not succeed, it reverts to TS151.011 (classic GSM mode).

Cardholder Verification is only performed if required by the (U)SIM application.

Random numbers are used for feeding the authentication mechanisms of the SIM card. As a result of this, the AUTHENTICATE for 3G applications will always fail, as the AUTN parameter is also randomized.

J.7 Embedded Protocol

The Embedded Protocol is a protocol based on Tag-Length-Value components. This protocol can easily be extended to support proprietary functions.

J.7.1 Specification

J.7.1.1 Protocol datagram format

An embedded protocol datagram is a concatenation of two mandatory fields and one optional field. these fields are :

- **Tag :** a field with a fixed length of one octet, describing the type of information conveyed in the 'value' field.
- **Length:** A field of variable length, containing the number of octets conveyed in the 'value' field. The number is encoded as an unsigned integer value, most significant byte first. The encoding of the Length field follows the Basic Encoding Rules for TLV objects:

The most significant bit of the first (left-most) octet of the Length field specifies the usage of the first octet. If it is clear ('0'), the remaining bits of the first octet encode the length of the value field. If it is set ('1'), the remaining bits (6-0) encode the number of octets used to encode the length of the value field. i.e. a value of 0x82 0x02 0x48 would be interpreted as 'the value field will contain 584 octets'. The maximum number of octets to use for encoding the value should be two octets (encoding a length field of at most 16 bits)

- Value: The value field contains the number of octets specified by the Length field. If the number of octets encoded by the Length field is equal to 0, the Value field shall be absent.

J.7.1.2 Communications

The inspector workstation shall transmit a single COMMAND-TLV, containing one or more data elements. The Target shall respond to each COMMAND-TLV with a RESPONSE-TLV. The RESPONSE-TLV shall contain the result of the COMMAND-TLV processing.

The Target may send unsolicited messages to the Inspector workstation, using a MESSAGE-TLV. MESSAGE-TLVs will not be acknowledged by Inspector.

J.7.1.3 Protocol TLVs

Three protocol TLVs are defined: COMMAND-TLV, RESPONSE-TLV and MESSAGE-TLV. This section describes the mandatory fields in these protocol TLVs.

COMMAND-TLV

The Value field of the COMMAND-TLV shall contain at least one FUNCTION-TLV.

RESPONSE-TLV

The Value field of the RESPONSE-TLV shall contain exactly one RESPONSE-CODE-TLV and may contain one or more responses encoded in FUNCTION-TLVs.

MESSAGE-TLV

The Value field of the MESSAGE-TLV shall contain exactly one MESSAGE-TYPE-TLV and exactly one MESSAGE-TEXT-TLV.

J.7.1.4 Function TLVs

All the TLVs defined in this section are considered FUNCTION-TLV objects.

INITIALIZE_APPLICATION

This function will cause the target to initialize. There shall be no Value field.

STOP_APPLICATION

This function will cause the target to stop. There shall be no Value field.

RESET_APPLICATION

This function will cause the target to reset. There shall be no Value field.

GET_ID

This function requests the application to identify itself. The value field shall be empty.

GET_INFO

This function requests the application to send information regarding supported features. The value field shall be empty.

SET_PARAMETER

The SET_PARAMETER function provides arbitrary parameters to the application. The parameters shall be conveyed in the value field.

SET_KEY

The SET_KEY function provides key data to the application. The value field shall contain exactly one of each of the following DATA-TLVs : ALGORITHM, KEY_LENGTH and KEY_DATA.

SET_ALGO

The SET_ALGO function sets the algorithm that is to be used in subsequent calls to the RUN_ALGO function. The value field shall contain exactly one ALGORITHM DATA-TLV, and optionally a ALGO_MODE DATA-TLV.

RUN_ALGO

The RUN_ALGO function runs the active algorithm as selected by the SET_ALGO mode. The function shall contain exactly one INPUT DATA-TLV of a pre-defined length. The length of the INPUT data is specified by the application or algorithm

J.7.1.5 Data TLVs

All the TLVs defined in this section are considered DATA-TLV objects.

ALGORITHM

The ALGORITHM field is a one-octet reference value to a cryptographic algorithm. The following algorithms have been defined:

Table J.1. Algorithm field values

| Value | Algorithm |
|-----------|-------------------------|
| 0x00 | DES |
| 0x01 | 3DES |
| 0x02 | RSA |
| 0x04-0x3F | Reserved for future use |
| 0x40-0xFF | Customer definable |

ALGO_MODE

The ALGO_MODE field defines the operating mode of the selected algorithm. The selected algorithm is not required to support all algorithms specified here.

The field shall be a one-octet reference value. The following operating modes have been defined:

Table J.2. Algorithm mode field values

| Value | Mode |
|-----------|-------------------------|
| 0x00 | Encrypt |
| 0x01 | Decrypt |
| 0x02 | Sign |
| 0x03 | Verify |
| 0x04-0x3F | Reserved for future use |
| 0x40-0xFF | Customer definable |

KEY_LENGTH

The content of the KEY_LENGTH field is to be interpreted by the algorithm.

KEY_DATA

The content of the KEY_DATA field is to be interpreted by the algorithm.

INPUT_DATA

The content of the INPUT_DATA field is to be processed by the algorithm.

OUTPUT_DATA

The content of the OUTPUT_DATA field is the result of the algorithm.

MESSAGE_TYPE

The MESSAGE_TYPE field has a length of exactly one octet, referencing one of the following message types:

Table J.3. MESSAGE_TYPE field values

| Value | Meaning |
|-----------|------------------------|
| 0x00 | Informational |
| 0x01 | Debugging |
| 0x02 | Error |
| 0x03 | Warning |
| 0x04-0xFF | Reserve for future use |

MESSAGE_TEXT

The MESSAGE_TEXT field contains a US-ASCII text field.

RESPONSE_CODE

The RESPONSE_CODE field contains a result code with a length of exactly one octet, referencing one of the following result codes:

Table J.4. Response code values

| Response code | Meaning |
|---------------|---|
| 0x00 | OK |
| 0x01 | Command Rejected (not implemented or not supported) |
| 0x02 | Parameter error |
| 0x03 | Command failed |

KEY_STRUCT

A KEY_STRUCT tag contains one or more TLVs specifying additional key information.

RSA_EXPONENT

The field of this TLV contains the RSA Exponent field encoded according to the algorithm specification.

RSA_MODULUS

The field of this TLV contains the RSA Modulus field encoded according to the algorithm specification.

J.7.1.6 List of assigned tags

The tag values assigned by Riscure are specified in the following tables.

Table J.5. Frame tags

| Frame | Tag |
|--------------|------|
| COMMAND-TLV | 0x01 |
| RESPONSE-TLV | 0x02 |
| MESSAGE-TLV | 0x03 |

Table J.6. Function tags

| Function | Tag |
|------------------------|------|
| INITIALIZE_APPLICATION | 0x08 |
| STOP_APPLICATION | 0x09 |
| RESET_APPLICATION | 0x0A |
| GET_ID | 0x0B |
| GET_INFO | 0x0C |
| SET_PARAMETER | 0x0D |
| SET_KEY | 0x0E |
| SET_ALGO | 0x0F |
| RUN_ALGO | 0x10 |

Table J.7. Parameter tags

| Parameter | Tag |
|-----------|------|
| ALGORITHM | 0x40 |

| Parameter | Tag |
|----------------|------|
| ALGORITHM_MODE | 0x41 |
| KEY_LENGTH | 0x42 |
| KEY_DATA | 0x43 |
| INPUT_DATA | 0x44 |
| OUTPUT_DATA | 0x45 |
| PARAMETER | 0x46 |
| MESSAGE_TYPE | 0x47 |
| MESSAGE_TEXT | 0x48 |
| RESPONSE_CODE | 0x50 |
| KEY_STRUCTURE | 0x51 |
| RSA_EXPONENT | 0x52 |
| RSA_MODULUS | 0x53 |

Tag values starting from 0x80 are customer definable.

J.7.2 Application protocol development

By default, two protocols are provided with Inspector:

- *EmbeddedDES*
- *EmbeddedRSA*

Protocols for other algorithms can be created based on the provided examples. All new protocols should implement the abstract class *EmbeddedProtocol*.

In order to develop a protocol for a new algorithm, the following methods should be implemented:

- Constructor. In the constructor, the super constructor should be called with the length of the randomized input data as an argument. In addition, the settings bean should be set. For Example:

```
public EmbeddedDES() throws IOException {
    super(8);
    setSettingsBean(BeanUtils.instantiate(getClass().getClassLoader(),
    DESSettings.class, EmbeddedDES.class.getName()));
}
```

The *setSettingsBean* method sets the settings bean. This is used for generating the user interface that allows configuring the application protocol.

- The *prepareAlgorithm* method. In this method, all operations that are needed to prepare the implementation of the algorithm should be performed. In general, this requires TLV commands to be sent to the target. For example:

```
@Override
public void prepareAlgorithm() throws IOException {
    // Set algorithm command...
    CompositeTLV setAlgorithmCommand = new CompositeTLV(Tag.PROTOCOL_COMMAND);
    CompositeTLV setAlgorithmFunction = new
    CompositeTLV(Tag.FUNCTION_SET_ALGORITHM);
```

```
TLV algorithm = new SimpleTLV(Tag.TLV_ALGORITHM, Algorithm.DES.value());
TLV algorithmMode = new SimpleTLV(Tag.TLV_ALGORITHM_MODE,
settings.getMode().value());
setAlgorithmFunction.add(algorithm, algorithmMode);
setAlgorithmCommand.add(setAlgorithmFunction);
sendToTarget(new TLVCommand(setAlgorithmCommand), preparePhase);

// Set key...
if (settings.isSetKeyDuringInit()) {
    CompositeTLV setKeyCommand = new CompositeTLV(Tag.PROTOCOL_COMMAND);
    CompositeTLV setKeyFunction = new CompositeTLV(Tag.FUNCTION_SET_KEY);
    TLV keyLength = new SimpleTLV(Tag.TLV_KEY_LENGTH, 64);
    TLV keyData = new SimpleTLV(Tag.TLV_KEY_DATA,
Conversions.toByteArray(settings.getKey()));
    setKeyFunction.add(algorithm, keyLength, keyData);
    setKeyCommand.add(setKeyFunction);
    sendToTarget(new TLVCommand(setKeyCommand), preparePhase);
}
}
```

In the above code, we can distinguish two types of TLV objects which are both based on the *TLV* interface. *CompositeTLV* objects can contain other TLV objects. It's *add* function can be used to add one or more TLV objects. *SimpleTLV* is a TLV object that contains data, but cannot contain other TLV objects. The *sendToTarget* method can be used to send TLV commands to the currently selected target.

Please note that commands should be constructed according to the embedded protocol specification.

- The *getSettingsBean* method. This method should simply return the *settings* object.

```
@Override
public Object getSettingsBean() {
    return settings;
}
```

- The *setSettingsBean* method. This method should set the *settings* object and register it as shown below.

```
@Override
public void setSettingsBean(Object settings) {
    this.settings = (DESSettings) settings;
    BeanUtils.register(this.settings, EmbeddedDES.class.getName());
}
```

Each embedded protocol can have a protocol specific settings bean. This is a simple Java class with some private fields, getters and setters. For example:

```
package acquisition2.protocol.embedded;

import javax.validation.constraints.Pattern;
import acquisition2.protocol.embedded.lib.AlgorithmMode;
import com.riscure.beans.annotation.DisplayName;

public class DESSettings {

    @DisplayName("Key")
    @Pattern(regexp="[0-9a-fA-F]{16}", message="Only hexadecimal characters allowed")
    private String key;
```

```
@DisplayName("Set key during initialization")
private boolean setKeyDuringInit;

@DisplayName("Mode")
private AlgorithmMode mode;

public void setKey(String key) {
    this.key = key;
}

public String getKey() {
    return key;
}

public void setMode(AlgorithmMode mode) {
    this.mode = mode;
}

public AlgorithmMode getMode() {
    return mode;
}

public void setSetKeyDuringInit(boolean setKeyDuringInit) {
    this.setKeyDuringInit = setKeyDuringInit;
}

public boolean isSetKeyDuringInit() {
    return setKeyDuringInit;
}
}
```

Inspector will automatically generate a user interface for this settings object.

K Cipher Suites

This appendix contains a listing of the Cipher Suites provided in Inspector.

- Section K.1, "AES"
- Section K.2, "ARIA"
- Section K.3, "Camellia"
- Section K.4, "DES"
- Section K.5, "DSA"
- Section K.6, "ECDSA"
- Section K.7, "HMAC"

K.1 AES

AES is a block cipher standard adopted by the U.S government. AES uses a block size of 128 bits, and is able to use 128-bit, 192-bit and 256-bit keys. It consists of 10, 12 or 14 rounds respectively, and is designed based on a Substitution Permutation Network.

References

<http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.

K.1.1 Analysis steps

The Inspector crypto modules take the following approach: key bytes are guessed by simply assuming a value (a candidate) and then computing intermediate data after data addition and substitution. The correlation of the intermediate data with the collected traces is computed and the candidate resulting in the best correlation is assumed to be correct.

Within AES, every round takes a round key whose length is equal to the block size. The discovery of one round key is insufficient for computing the full key if the full key size is larger than the block size. In that case, a second round must be attacked.

When finished, a (round/full) key is presented, along with a list of correlation values.

K.1.1.1 Output

We will give an example of an attack on an AES implementation.

- "HD SBox in/out round X" targets the Hamming Distance between the input and the output of the S-boxes in round X.
- "HW SBox out round X" targets the Hamming Weight of the output of the S-boxes in round X. This target uses the input data in encryption mode and the output data in decryption mode.

- "HW SBox in round X" targets the Hamming Weight of the input of the S-boxes in round X. This target uses the output data in encryption mode and the input data in decryption mode.

If the key is 128 bits, attacking the first round (or the last round) already provides enough information to recover the full key. Otherwise, a second attack on the second (respectively the last but one) round needs to be performed.

```
...
Best correlation Key byte 14:
Key byte candidate: 113, value: 0.6620, at position: 371
Key byte candidate: 168, value: 0.4690, at position: 199
Key byte candidate: 104, value: 0.4393, at position: 352
Key byte candidate: 20, value: 0.4255, at position: 417
Best correlation Key byte 15:
Key byte candidate: 17, value: 0.6620, at position: 167
Key byte candidate: 85, value: 0.4430, at position: 211
Key byte candidate: 25, value: 0.4081, at position: 111
Key byte candidate: 114, value: 0.3930, at position: 397
Best correlation Key byte 16:
Key byte candidate: 62, value: 0.6848, at position: 146
Key byte candidate: 12, value: 0.4840, at position: 104
Key byte candidate: 144, value: 0.4247, at position: 245
Key byte candidate: 204, value: 0.4169, at position: 168
AES key: D8 1E 94 E5 D2 ED 7B ED 3E 2C 3A E3 6B 71 11 3E
```

The table above shows the correlation for each key candidate (in decimal notation). Note that the best value differs significantly from the second-best value (e.g. 0.662 vs. 0.469). This is an indication for the reliability of the results. The resulting full AES key is presented at the bottom in hexadecimal coding. Note that (part of) the first round key equals (part of) the full key.

Example

The AESsimulation.trs file, included on the DVD with Inspector training trace sets, contains 100 simulated power consumption traces, as well as randomly chosen AES input and output data. The simulated traces contain a little noise that will still allow the statistical properties to be observed. Note that real-life traces will contain much more noise that will significantly hamper the analysis. An analyst will have to use many more traces and align them in order to achieve any practical results.

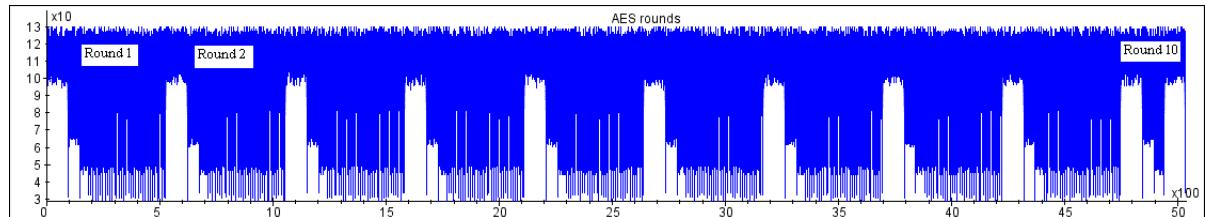
Related tutorials

- **AES Analysis Tutorial (simulation)** [\[../tutorials/sectionsAESSimTutorial.html\]\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsAESSimTutorial\]](#)
- **Analysis of an AES implementation** [\[../tutorials/sectionsAESCardTutorial.html\]\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsAESCardTutorial\]](#)
- **Analysis of an AES implementation with random delays** [\[../tutorials/sectionsAESCardRndDelaysTutorial.html\]\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsAESCardRndDelaysTutorial\]](#)
- **Analysis of an AES implementation with S-box randomization** [\[../tutorials/sectionsAesCardRndSboxTutorial.html\]\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsAesCardRndSboxTutorial\]](#)

K.1.2 Background

The AES algorithm was developed just before the year 2000, and is becoming the world's leading symmetric security algorithm, replacing the popular variations of the DES algorithm. The AES algorithm is used, for example, in the authentication scheme for 3G mobile networks (Milenage).

The AES algorithm can take keys and block sizes of either 128, 192 or 256 bits. The number of rounds is either 10, 12 or 14, depending on the key and block sizes. As a first step a key schedule is created where the full key is used to derive N+1 round keys for N rounds. The figure below shows a trace including 10 rounds.



All rounds (except for the last one) have the following structure:

- **key addition**

The round key is added modulo-two to the data.

- **substitute**

Each data byte is replaced by the value given by an 8-bit substitution table.

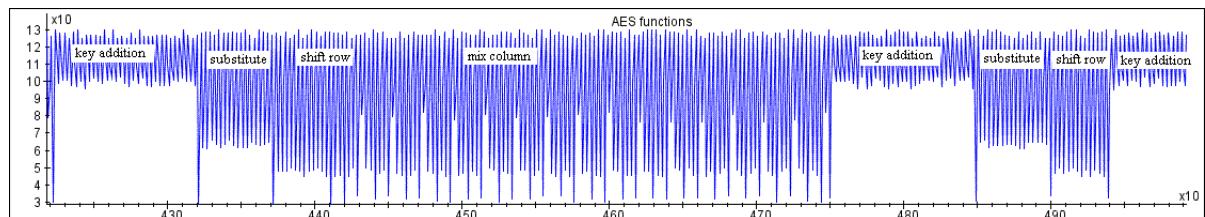
- **shift row**

The order of bytes in the data array is changed.

- **mix column**

Data bytes are mixed according to a mathematical function.

The last round is a little different because it skips the mix column function, but adds an additional key addition step. The figure below shows a trace including the last but one and last rounds.



K.2 ARIA

ARIA is a general-purpose involutational SPN block cipher algorithm designed in 2003 by a group of Korean researchers. In 2004, ARIA was established as a Korean Standard block cipher algorithm (KS X 1213) by the Ministry of Knowledge Economy. The algorithm uses a substitution-permutation network structure based on AES. The interface is the same as AES:

128-bit block size with key size of 128, 192, or 256 bits. The number of rounds is 12, 14, or 16, depending on the key size. ARIA uses two 8×8 -bit S-boxes and their inverses in alternate rounds; one of these is the Rijndael S-box. The key schedule processes the key using a 3-round 256-bit Feistel cipher. Most operations that ARIA uses are simple, byte-oriented ones like XOR in order to be competent in performance in lightweight environments.

Figure K.1. ARIA encryption and decryption processes.

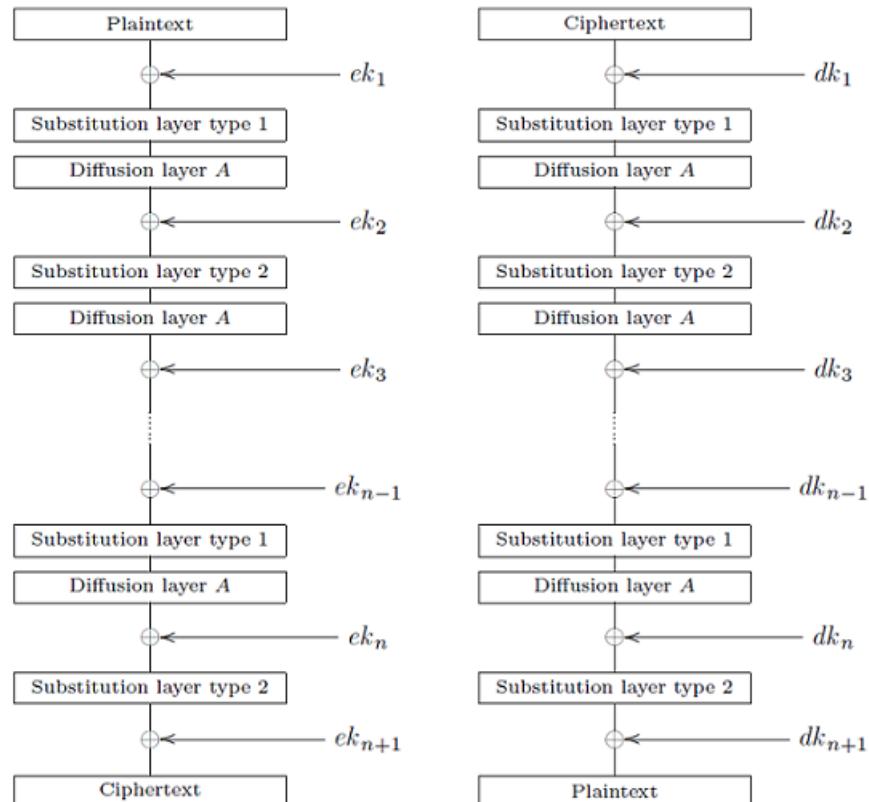


Figure K.2. ARIA substitution layer, type 1.

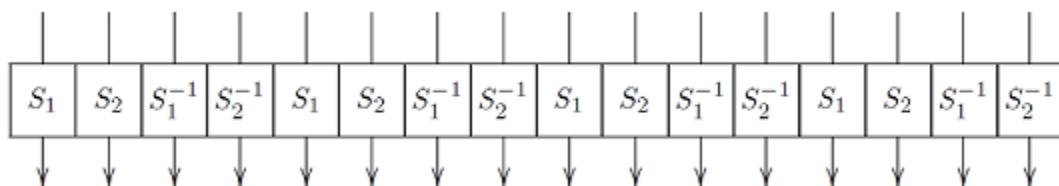
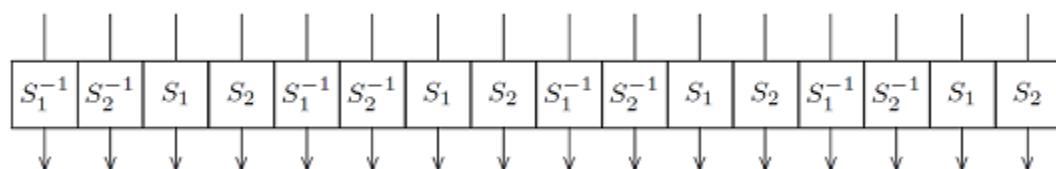


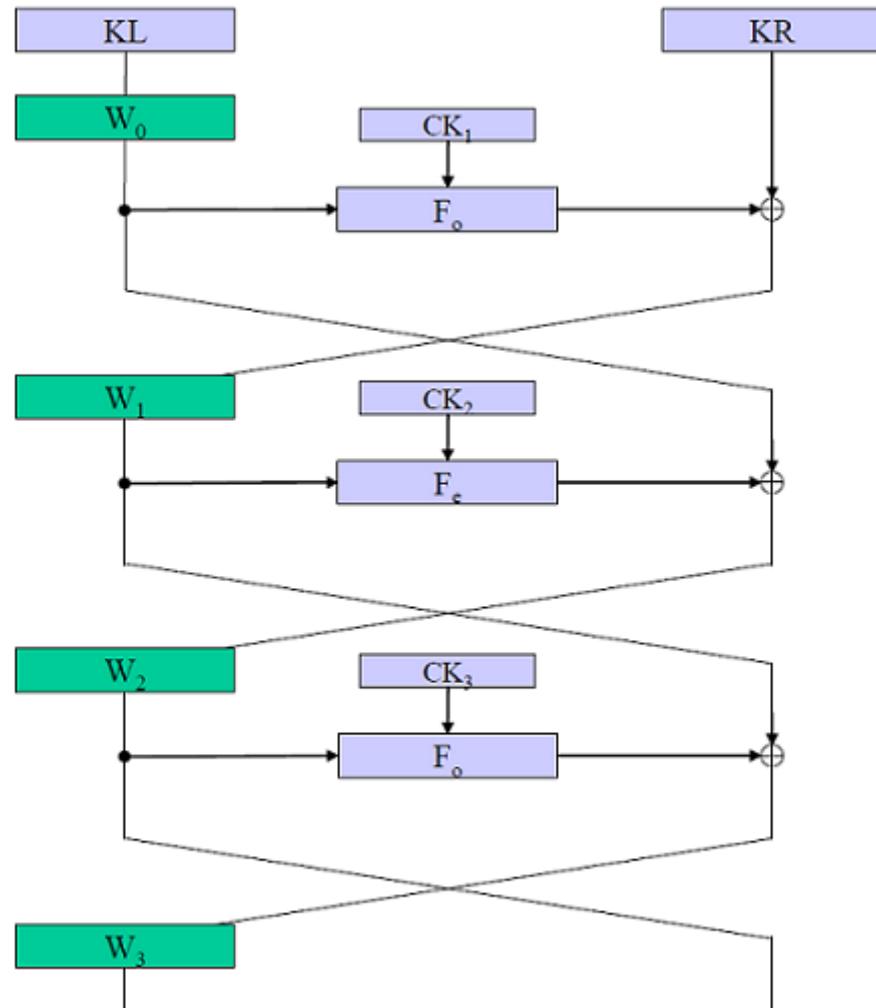
Figure K.3. ARIA substitution layer, type 2.



The ARIA key expansion consists of two parts, which are initialization and round key generation.

In the initialization part, four 128-bit values W_0, W_1, W_2 , and W_3 are generated from the master key MK, by using a 3-round 256-bit Feistel cipher. The MK can be of 128-, 192-, or 256-bit. We first fill out the 128-bit value KL with bits from MK and use what is left of MK (if any) on the 128-bit value KR. The space remaining on KR (if any) is filled with zeros.

Figure K.4. Initialization for ARIA key expansion.



In the round key generation, the four values W_i are combined to generate the encryption round keys ek_i and the decryption round keys dk_i . The following is the definition of ek_i .

Table K.1. ARIA round key generation.

| | |
|--------------------------------|--------------------------------|
| $ek_1 = (W_0) \# (W_1 \gg 19)$ | $ek_2 = (W_2) \# (W_3 \gg 19)$ |
| $ek_3 = (W_2) \# (W_3 \gg 19)$ | $ek_4 = (W_3) \# (W_0 \gg 19)$ |
| $ek_5 = (W_0) \# (W_1 \gg 31)$ | $ek_6 = (W_1) \# (W_2 \gg 31)$ |
| $ek_7 = (W_2) \# (W_3 \gg 31)$ | $ek_8 = (W_3) \# (W_0 \gg 31)$ |

| | |
|------------------------------------|------------------------------------|
| $ek_9 = (W_0) \# (W_1 \lll 61)$ | $ek_{10} = (W_1) \# (W_2 \lll 61)$ |
| $ek_{11} = (W_2) \# (W_3 \lll 61)$ | $ek_{12} = (W_3) \# (W_0 \lll 61)$ |
| $ek_{13} = (W_0) \# (W_1 \lll 31)$ | $ek_{14} = (W_1) \# (W_2 \lll 31)$ |
| $ek_{15} = (W_2) \# (W_3 \lll 31)$ | $ek_{16} = (W_3) \# (W_0 \lll 31)$ |
| $ek_{17} = (W_0) \# (W_1 \lll 19)$ | |

The number of rounds we use are 12, 14, or 16, corresponding to the key size 128, 192, or 256 of the master key, respectively. Since there is one more key addition layer after the last round, in fact the number of round keys need is 13, 15, or 17.

The decryption round keys are different from the encryption round keys, and derived from the encryption round keys. The ordering of round keys are reversed followed by the output of the diffusion layer A to all round keys except for the first and the last. The following gives the definition of dk_i : $dk_1 = ek_{n+1}$, $dk_2 = A(ek_n)$, $dk_3 = A(ek_{n-1})$, ..., $dk_n = A(ek_2)$, $dk_{n+1} = ek_1$.

K.2.1 Analysis steps

The DPA attack on ARIA takes 4 steps, attacking 4 different round keys using either the plaintext (to recover the first round keys) or the ciphertext (to recover the last round keys). The full MK can be recovered once the first 4 or the last 4 round keys have been found. No bits of MK can be recovered using fewer round keys. This holds for all key lengths.

This module offers the following targets and leakage models for the analysis:

- Hamming Weight (HW) of the S-box output (using plaintext)
- Hamming Weight (HW) of the S-box input (using ciphertext)
- Hamming Distance (HD) between the S-box input and S-box output (using plaintext or ciphertext)

Each of these targets depends on 8 bits of the key, and therefore 256 candidates are tested for each S-box.

The analysis can be performed both in encryption mode and decryption mode.

K.2.2 Output

After the first step, the attack produces a listing providing the best candidates for each S-box:

```

Algorithm: ARIA-128 ENCRYPT
Possible attacks: [HD SBox in/out round 1 (using input), HD SBox in/out round 12
(using output),
HW SBox in round 12 (using output), HW SBox out round 1 (using input)]
Attacking: HW SBox out round 1 (using input)
Key targets: key[00][00] => HW(sboxOut[01][00]), key[00][01] => HW(sboxOut[01][01]),
key[00][02] => HW(sboxOut[01][02]), key[00][03] => HW(sboxOut[01][03]), key[00][04]
=>
HW(sboxOut[01][04]), key[00][05] => HW(sboxOut[01][05]), key[00][06] =>
HW(sboxOut[01][06]),
key[00][07] => HW(sboxOut[01][07]), key[00][08] => HW(sboxOut[01][08]), key[00][09]
=>
HW(sboxOut[01][09]), key[00][10] => HW(sboxOut[01][10]), key[00][11] =>
HW(sboxOut[01][11]), key[00][12] =>
```

```
HW(sboxOut[01][12]), key[00][13] => HW(sboxOut[01][13]), key[00][14] =>
HW(sboxOut[01][14]), key[00][15] =>
HW(sboxOut[01][15])
Best correlation key[00][00]:
0, key[00][00]=81, value: 1.0000, at position: 32
1, key[00][00]=cd, value: -0.4325, at position: 0
2, key[00][00]=60, value: 0.4261, at position: 484
3, key[00][00]=7b, value: -0.4215, at position: 380
Best correlation key[00][01]:
0, key[00][01]=d6, value: 1.0000, at position: 33
1, key[00][01]=fb, value: -0.4548, at position: 75
2, key[00][01]=e5, value: 0.4270, at position: 256
3, key[00][01]=f5, value: 0.4050, at position: 56
Best correlation key[00][02]:
0, key[00][02]=c9, value: 1.0000, at position: 34
1, key[00][02]=d2, value: 0.4119, at position: 236
2, key[00][02]=2a, value: -0.4115, at position: 66
3, key[00][02]=61, value: -0.4071, at position: 276
Best correlation key[00][03]:
0, key[00][03]=bc, value: 1.0000, at position: 35
1, key[00][03]=f1, value: -0.4230, at position: 532
2, key[00][03]=a2, value: 0.4194, at position: 380
3, key[00][03]=bd, value: 0.4049, at position: 159
Best correlation key[00][04]:
0, key[00][04]=5e, value: 1.0000, at position: 36
1, key[00][04]=df, value: -0.4595, at position: 546
2, key[00][04]=6d, value: 0.4225, at position: 317
3, key[00][04]=2b, value: 0.4200, at position: 36
Best correlation key[00][05]:
0, key[00][05]=0f, value: 1.0000, at position: 37
1, key[00][05]=20, value: 0.4082, at position: 540
2, key[00][05]=2a, value: 0.4063, at position: 171
3, key[00][05]=ce, value: -0.4020, at position: 537
Best correlation key[00][06]:
0, key[00][06]=9a, value: 1.0000, at position: 38
1, key[00][06]=03, value: 0.4167, at position: 366
2, key[00][06]=69, value: 0.4149, at position: 218
3, key[00][06]=fa, value: 0.4148, at position: 185
Best correlation key[00][07]:
0, key[00][07]=98, value: 1.0000, at position: 39
1, key[00][07]=1f, value: -0.4749, at position: 149
2, key[00][07]=52, value: 0.4120, at position: 319
3, key[00][07]=da, value: 0.4073, at position: 29
Best correlation key[00][08]:
0, key[00][08]=7f, value: 1.0000, at position: 40
1, key[00][08]=74, value: -0.4645, at position: 24
2, key[00][08]=13, value: 0.4425, at position: 545
3, key[00][08]=b4, value: -0.4329, at position: 545
Best correlation key[00][09]:
0, key[00][09]=1c, value: 1.0000, at position: 41
1, key[00][09]=40, value: 0.4162, at position: 41
2, key[00][09]=0c, value: -0.4111, at position: 107
3, key[00][09]=bf, value: -0.4022, at position: 199
Best correlation key[00][10]:
0, key[00][10]=71, value: 1.0000, at position: 42
1, key[00][10]=36, value: -0.4483, at position: 226
2, key[00][10]=e1, value: -0.4222, at position: 49
3, key[00][10]=af, value: -0.4168, at position: 529
Best correlation key[00][11]:
0, key[00][11]=71, value: 1.0000, at position: 43
```

```
1, key[00][11]=35, value: 0.4229, at position: 424
2, key[00][11]=b5, value: 0.4213, at position: 454
3, key[00][11]=eb, value: 0.4169, at position: 389
Best correlation key[00][12]:
0, key[00][12]=1b, value: 1.0000, at position: 44
1, key[00][12]=02, value: 0.4580, at position: 392
2, key[00][12]=c4, value: -0.4360, at position: 336
3, key[00][12]=89, value: 0.4315, at position: 298
Best correlation key[00][13]:
0, key[00][13]=bd, value: 1.0000, at position: 45
1, key[00][13]=da, value: 0.4101, at position: 28
2, key[00][13]=f7, value: 0.4041, at position: 413
3, key[00][13]=81, value: 0.4034, at position: 339
Best correlation key[00][14]:
0, key[00][14]=d7, value: 1.0000, at position: 46
1, key[00][14]=b1, value: 0.4596, at position: 550
2, key[00][14]=05, value: 0.4127, at position: 313
3, key[00][14]=57, value: -0.4117, at position: 95
Best correlation key[00][15]:
0, key[00][15]=d7, value: 1.0000, at position: 47
1, key[00][15]=0c, value: -0.4394, at position: 111
2, key[00][15]=2f, value: 0.4232, at position: 547
3, key[00][15]=b9, value: -0.4113, at position: 436
Key entropy: 128
Key info:
No master key found. Need more round keys
Entropy too large to print or bruteforce key candidates
```

In addition to this listing the module produces a set of correlation traces.

A similar output is produced for subsequent steps. A listing with the top candidates for each S-box is produced.

In the last (4th) step of the attack, the listing of the 4th round key is followed by a candidate for the complete key. For instance, for analysis of ARIA-128, this results in the output below.

```
Algorithm: ARIA-128 ENCRYPT
Possible attacks: [HD SBox in/out round 4 (using input), HW SBox out round 4 (using
input), HD SBox in/out round 3 (using input), HW SBox out round 3 (using input), HD
SBox in/out round 2 (using input), HW SBox out round 2 (using input), HD SBox in/out
round 1 (using input), HD SBox in/out round 12 (using output), HW SBox in round 12
(using output), HW SBox out round 1 (using input)]
Attacking: HW SBox out round 4 (using input)
Key targets: key[03][00] => HW(sboxOut[04][00]), key[03][01] => HW(sboxOut[04]
[01]), key[03][02] => HW(sboxOut[04][02]), key[03][03] => HW(sboxOut[04][03]),
key[03][04] => HW(sboxOut[04][04]), key[03][05] => HW(sboxOut[04][05]),
key[03][06] => HW(sboxOut[04][06]), key[03][07] => HW(sboxOut[04][07]),
key[03][08] => HW(sboxOut[04][08]), key[03][09] => HW(sboxOut[04][09]), key[03]
[10] => HW(sboxOut[04][10]), key[03][11] => HW(sboxOut[04][11]), key[03][12]
=> HW(sboxOut[04][12]), key[03][13] => HW(sboxOut[04][13]), key[03][14] =>
HW(sboxOut[04][14]), key[03][15] => HW(sboxOut[04][15])
Best correlation key[03][00]:
0, key[03][00]=7f, value: 1.0000, at position: 176
1, key[03][00]=3d, value: 0.4528, at position: 190
2, key[03][00]=b5, value: -0.4404, at position: 486
3, key[03][00]=e7, value: 0.4338, at position: 77
Best correlation key[03][01]:
0, key[03][01]=5b, value: 1.0000, at position: 177
1, key[03][01]=03, value: 0.4167, at position: 419
2, key[03][01]=12, value: 0.3981, at position: 464
```

```
3, key[03][01]=0b, value:  0.3959, at position: 61
Best correlation key[03][02]:
0, key[03][02]=4b, value:  1.0000, at position: 178
1, key[03][02]=04, value: -0.4357, at position: 354
2, key[03][02]=51, value:  0.4239, at position: 281
3, key[03][02]=b2, value: -0.4138, at position: 287
Best correlation key[03][03]:
0, key[03][03]=4b, value:  1.0000, at position: 179
1, key[03][03]=d9, value: -0.4439, at position: 116
2, key[03][03]=4c, value: -0.4225, at position: 364
3, key[03][03]=8e, value: -0.4158, at position: 276
Best correlation key[03][04]:
0, key[03][04]=9b, value:  1.0000, at position: 180
1, key[03][04]=42, value: -0.4367, at position: 243
2, key[03][04]=78, value:  0.4344, at position: 239
3, key[03][04]=ee, value: -0.4289, at position: 409
Best correlation key[03][05]:
0, key[03][05]=fa, value:  1.0000, at position: 181
1, key[03][05]=08, value:  0.4725, at position: 181
2, key[03][05]=5f, value:  0.4268, at position: 475
3, key[03][05]=4d, value:  0.4063, at position: 446
Best correlation key[03][06]:
0, key[03][06]=17, value:  1.0000, at position: 182
1, key[03][06]=ad, value:  0.4422, at position: 149
2, key[03][06]=a3, value:  0.4261, at position: 201
3, key[03][06]=12, value: -0.4153, at position: 150
Best correlation key[03][07]:
0, key[03][07]=fb, value:  1.0000, at position: 183
1, key[03][07]=5a, value:  0.4518, at position: 369
2, key[03][07]=15, value: -0.4011, at position: 564
3, key[03][07]=5f, value: -0.3995, at position: 159
Best correlation key[03][08]:
0, key[03][08]=c2, value:  1.0000, at position: 184
1, key[03][08]=6f, value: -0.4636, at position: 416
2, key[03][08]=cf, value: -0.4327, at position: 431
3, key[03][08]=9e, value: -0.4208, at position: 411
Best correlation key[03][09]:
0, key[03][09]=c2, value:  1.0000, at position: 185
1, key[03][09]=70, value: -0.4217, at position: 169
2, key[03][09]=f0, value:  0.4181, at position: 352
3, key[03][09]=e7, value: -0.4165, at position: 307
Best correlation key[03][10]:
0, key[03][10]=9a, value:  1.0000, at position: 186
1, key[03][10]=c9, value: -0.4537, at position: 278
2, key[03][10]=eb, value: -0.4145, at position: 551
3, key[03][10]=d3, value: -0.4123, at position: 276
Best correlation key[03][11]:
0, key[03][11]=c5, value:  1.0000, at position: 187
1, key[03][11]=6e, value: -0.4403, at position: 473
2, key[03][11]=f8, value: -0.4283, at position: 386
3, key[03][11]=36, value: -0.4001, at position: 251
Best correlation key[03][12]:
0, key[03][12]=ff, value:  1.0000, at position: 188
1, key[03][12]=ba, value: -0.4282, at position: 379
2, key[03][12]=99, value: -0.4169, at position: 466
3, key[03][12]=d6, value: -0.4136, at position: 249
Best correlation key[03][13]:
0, key[03][13]=b8, value:  1.0000, at position: 189
1, key[03][13]=a8, value:  0.4368, at position: 375
2, key[03][13]=4d, value:  0.4251, at position: 157
```

```
3, key[03][13]=ab, value: -0.4180, at position: 173
Best correlation key[03][14]:
0, key[03][14]=75, value: 1.0000, at position: 190
1, key[03][14]=b6, value: -0.4285, at position: 158
2, key[03][14]=54, value: 0.4242, at position: 472
3, key[03][14]=8f, value: -0.4230, at position: 89
Best correlation key[03][15]:
0, key[03][15]=8b, value: 1.0000, at position: 191
1, key[03][15]=33, value: -0.4641, at position: 130
2, key[03][15]=a8, value: -0.4346, at position: 165
3, key[03][15]=27, value: 0.4140, at position: 198
Key entropy: 0
Key info:
Found compatible round keys; inverted master key candidate:
0xcafebabecafebabecafebabecafebabe
Trying key candidates...
Correct key found: cafebabecafebabecafebabecafebabe
```

K.2.3 Validation

When the module attempts to compute the complete key, a validation step is applied in order to provide a hint to the analyst on whether the key is correct or not. For ARIA, this validation consists of computing the key schedule based on the resulting key(s) and testing it against the keys obtained with DPA.

This results in a candidate for the master key if succeeded and no candidate otherwise. If the DPA attack did not succeed, the module will first test a number of different key candidates using the top candidates for each S-box. If the input and output of the algorithm are known, the module also verifies whether the recovered key generates the expected output. A report like the one below will be printed in the output window if no validated key is found.

```
Algorithm: ARIA-128 ENCRYPT
Possible attacks: [HD SBox in/out round 4 (using input), HW SBox out round 4 (using
    input), HD SBox in/out round 3 (using input), HW SBox out round 3 (using input), HD
    SBox in/out round 2 (using input), HW SBox out round 2 (using input), HD SBox in/out
    round 1 (using input), HD SBox in/out round 12 (using output), HW SBox in round 12
    (using output), HW SBox out round 1 (using input)]
Attacking: HD SBox in/out round 4 (using input)
Key targets: key[03][00] => HD(sboxIn[04][00], sboxOut[04][00]), key[03][01]
    => HD(sboxIn[04][01], sboxOut[04][01]), key[03][02] => HD(sboxIn[04][02],
    sboxOut[04][02]), key[03][03] => HD(sboxIn[04][03], sboxOut[04][03]), key[03]
[04] => HD(sboxIn[04][04], sboxOut[04][04]), key[03][05] => HD(sboxIn[04][05],
    sboxOut[04][05]), key[03][06] => HD(sboxIn[04][06], sboxOut[04][06]), key[03]
[07] => HD(sboxIn[04][07], sboxOut[04][07]), key[03][08] => HD(sboxIn[04][08],
    sboxOut[04][08]), key[03][09] => HD(sboxIn[04][09], sboxOut[04][09]), key[03]
[10] => HD(sboxIn[04][10], sboxOut[04][10]), key[03][11] => HD(sboxIn[04][11],
    sboxOut[04][11]), key[03][12] => HD(sboxIn[04][12], sboxOut[04][12]), key[03][13] =>
    HD(sboxIn[04][13], sboxOut[04][13]), key[03][14] => HD(sboxIn[04][14], sboxOut[04]
[14]), key[03][15] => HD(sboxIn[04][15], sboxOut[04][15])
Best correlation key[03][00]:
0, key[03][00]=24, value: -0.3468, at position: 95
1, key[03][00]=d0, value: -0.3291, at position: 103
2, key[03][00]=ae, value: 0.3275, at position: 97
3, key[03][00]=00, value: -0.3269, at position: 103
Best correlation key[03][01]:
0, key[03][01]=62, value: -0.3856, at position: 99
1, key[03][01]=a2, value: -0.3523, at position: 99
2, key[03][01]=48, value: -0.3346, at position: 97
```

```
3, key[03][01]=99, value: 0.3321, at position: 101
Best correlation key[03][02]:
0, key[03][02]=9d, value: 0.4131, at position: 97
1, key[03][02]=05, value: 0.3671, at position: 103
2, key[03][02]=7f, value: -0.3480, at position: 95
3, key[03][02]=76, value: -0.3299, at position: 99
Best correlation key[03][03]:
0, key[03][03]=d6, value: -0.3511, at position: 98
1, key[03][03]=50, value: 0.3353, at position: 103
2, key[03][03]=a5, value: -0.3120, at position: 96
3, key[03][03]=aa, value: 0.3048, at position: 100
Best correlation key[03][04]:
0, key[03][04]=a3, value: -0.3641, at position: 96
1, key[03][04]=9b, value: -0.3584, at position: 98
2, key[03][04]=d7, value: 0.3279, at position: 101
3, key[03][04]=c8, value: 0.3046, at position: 99
Best correlation key[03][05]:
0, key[03][05]=7a, value: 0.3468, at position: 95
1, key[03][05]=ff, value: 0.3365, at position: 97
2, key[03][05]=a2, value: 0.3315, at position: 96
3, key[03][05]=34, value: 0.3215, at position: 103
Best correlation key[03][06]:
0, key[03][06]=04, value: -0.3307, at position: 103
1, key[03][06]=7f, value: -0.3093, at position: 100
2, key[03][06]=9f, value: -0.3047, at position: 100
3, key[03][06]=44, value: 0.2992, at position: 95
Best correlation key[03][07]:
0, key[03][07]=81, value: 0.3508, at position: 96
1, key[03][07]=b8, value: 0.3170, at position: 95
2, key[03][07]=4a, value: 0.2970, at position: 96
3, key[03][07]=b1, value: -0.2948, at position: 96
Best correlation key[03][08]:
0, key[03][08]=13, value: -0.3179, at position: 101
1, key[03][08]=c7, value: 0.3093, at position: 97
2, key[03][08]=3d, value: -0.3065, at position: 102
3, key[03][08]=be, value: -0.3003, at position: 96
Best correlation key[03][09]:
0, key[03][09]=ea, value: 0.3421, at position: 96
1, key[03][09]=55, value: 0.3203, at position: 102
2, key[03][09]=70, value: 0.3104, at position: 97
3, key[03][09]=86, value: 0.3091, at position: 97
Best correlation key[03][10]:
0, key[03][10]=d0, value: -0.3831, at position: 97
1, key[03][10]=fe, value: 0.3285, at position: 101
2, key[03][10]=17, value: -0.3254, at position: 95
3, key[03][10]=be, value: 0.3124, at position: 102
Best correlation key[03][11]:
0, key[03][11]=70, value: -0.3616, at position: 100
1, key[03][11]=32, value: -0.3521, at position: 96
2, key[03][11]=d0, value: 0.3478, at position: 100
3, key[03][11]=06, value: -0.3440, at position: 101
Best correlation key[03][12]:
0, key[03][12]=cd, value: -0.3119, at position: 97
1, key[03][12]=d6, value: -0.3051, at position: 100
2, key[03][12]=fb, value: 0.3036, at position: 101
3, key[03][12]=a0, value: 0.2985, at position: 101
Best correlation key[03][13]:
0, key[03][13]=5f, value: 0.3510, at position: 102
1, key[03][13]=52, value: -0.3204, at position: 101
2, key[03][13]=70, value: 0.3189, at position: 102
```

```

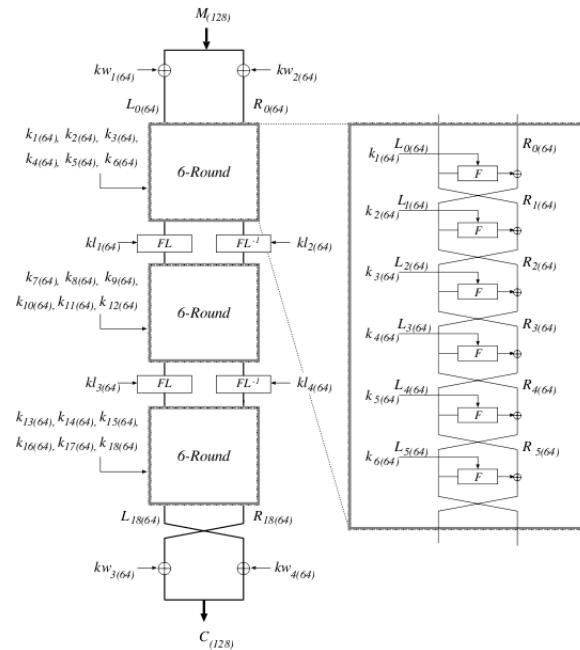
3, key[03][13]=59, value: 0.3102, at position: 102
Best correlation key[03][14]:
0, key[03][14]=4e, value: -0.3565, at position: 100
1, key[03][14]=01, value: 0.3229, at position: 99
2, key[03][14]=f4, value: 0.3087, at position: 95
3, key[03][14]=35, value: 0.3079, at position: 99
Best correlation key[03][15]:
0, key[03][15]=f6, value: 0.3312, at position: 98
1, key[03][15]=be, value: -0.3293, at position: 98
2, key[03][15]=b6, value: -0.3223, at position: 103
3, key[03][15]=56, value: -0.3159, at position: 102
Tried 65536 key candidates, no consistent candidates found, key is invalid
Key entropy: 128
Key info:
No compatible round keys found. Retry attack.

```

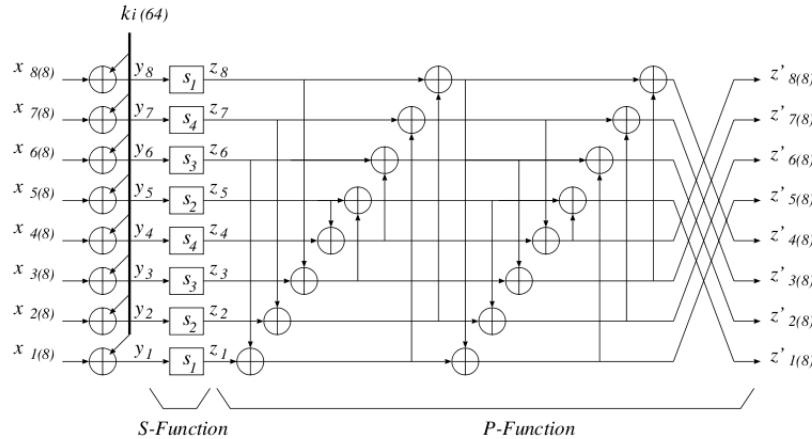
K.3 Camellia

Camellia is a block cipher developed jointly by Mitsubishi and NTT, with a similar design to Misty1. Camellia uses a block size of 128 bits, and is able to use 128-bit, 192-bit and 256-bit keys. It uses 18 rounds (for 128-bit keys) or 24 rounds (for 192 and 256 bits), based on a Feistel structure. The input is divided in two halves of 64 bits and pre-whitening is applied XORing key material with these halves. Every 6 rounds, the FL and its inverse operation are applied to the left and right half respectively. Finally, post-whitening is also applied by means of an extra XOR operation with key related information.

Figure K.5. Camellia encryption with 128 bit keys (18 rounds).



The key schedule is based on producing 2 or 4 extended keys, depending on whether a 128-bit key is used or a 192/256-bit key is used. Rotated versions of these keys are used in the subsequent rounds, using 64 bits per round. The round function of Camellia is based on 8 bit S-boxes and a P operation in which each output byte is a combination of 4 input bytes.

Figure K.6. Camellia's Feistel function.

K.3.1 Analysis steps

The DPA attack takes up to 6 steps, attacking 6 different round keys. Due to the structure of the Camellia cipher, masked round keys are recovered for each step. These keys are a combination of the actual round keys and the whitening keys applied to the input or output of the cipher. After enough masked round keys have been recovered, the module can recover the original key.

Depending on the selected version of the algorithm (key length and encryption or decryption) and input/output options, a number of available targets is presented. This module offers the following targets for each round:

- Hamming Weight (HW) of the S-box output
- Hamming Distance (HD) between the S-box input and S-box output

These targets depend on 8 bits of the key, and therefore 256 candidates are tested for each S-box.

When attacking a 128 bit key in encryption mode from the input, 4 steps are required. When attacking it from the output, 6 steps are required instead. For decryption mode, 4 steps are required for attacks based on the output and 6 steps for attacks based on the input data. For larger key sizes 6 steps are always required.

K.3.2 Output

After the first step, the attack produces a listing providing the best candidates for each S-box, as well as the recovered masked key:

```

Performing: HW SBox out round 0 (using input)
Best correlation key 0:
0, Key candidate: 4 (0x04), value: 1.0000, at position: 0
1, Key candidate: 64 (0x40), value: 0.4661, at position: 107
2, Key candidate: 227 (0xE3), value: 0.4548, at position: 269
3, Key candidate: 120 (0x78), value: 0.4308, at position: 281

```

```
Best correlation key 1:  
0, Key candidate: 173 (0xAD), value: 1.0000, at position: 1  
1, Key candidate: 20 (0x14), value: -0.4306, at position: 105  
2, Key candidate: 57 (0x39), value: 0.4118, at position: 310  
3, Key candidate: 9 (0x09), value: 0.4110, at position: 54  
Best correlation key 2:  
0, Key candidate: 189 (0xBD), value: 1.0000, at position: 2  
1, Key candidate: 146 (0x92), value: 0.4748, at position: 239  
2, Key candidate: 114 (0x72), value: -0.4692, at position: 217  
3, Key candidate: 13 (0x0D), value: 0.4231, at position: 0  
Best correlation key 3:  
0, Key candidate: 159 (0x9F), value: 1.0000, at position: 3  
1, Key candidate: 92 (0x5C), value: 0.4154, at position: 298  
2, Key candidate: 12 (0x0C), value: -0.3888, at position: 242  
3, Key candidate: 174 (0xAE), value: -0.3879, at position: 68  
Best correlation key 4:  
0, Key candidate: 166 (0xA6), value: 1.0000, at position: 4  
1, Key candidate: 168 (0xA8), value: -0.4364, at position: 216  
2, Key candidate: 217 (0xD9), value: 0.4223, at position: 219  
3, Key candidate: 27 (0x1B), value: 0.3788, at position: 14  
Best correlation key 5:  
0, Key candidate: 65 (0x41), value: 1.0000, at position: 5  
1, Key candidate: 218 (0xDA), value: 0.4251, at position: 216  
2, Key candidate: 3 (0x03), value: 0.4086, at position: 320  
3, Key candidate: 155 (0x9B), value: -0.3993, at position: 107  
Best correlation key 6:  
0, Key candidate: 33 (0x21), value: 1.0000, at position: 6  
1, Key candidate: 4 (0x04), value: -0.4141, at position: 66  
2, Key candidate: 169 (0xA9), value: 0.4115, at position: 21  
3, Key candidate: 210 (0xD2), value: -0.4001, at position: 325  
Best correlation key 7:  
0, Key candidate: 124 (0x7C), value: 1.0000, at position: 7  
1, Key candidate: 214 (0xD6), value: 0.4245, at position: 83  
2, Key candidate: 201 (0xC9), value: -0.4172, at position: 289  
3, Key candidate: 142 (0x8E), value: -0.4098, at position: 351  
Key entropy: 256  
Key info:  
Known subkeys:  
Masked k0: 4abd9fa641217c
```

Entropy too large to print or bruteforce key

If the *Traces* check box was selected, in addition to this listing the module produces a set of correlation traces.

A similar output is produced for subsequent steps. A listing with the best candidates for each S-box is produced, and a list of recovered masked keys is printed at the end. For instance, the following listing shows the output of the third step of the attack:

```
Performing: HW SBox out round 2 (using input)  
Best correlation key 0:  
0, Key candidate: 92 (0x5C), value: 1.0000, at position: 32  
1, Key candidate: 134 (0x86), value: -0.4480, at position: 311  
2, Key candidate: 0 (0x00), value: -0.4277, at position: 123  
3, Key candidate: 83 (0x53), value: -0.4256, at position: 225  
Best correlation key 1:  
0, Key candidate: 146 (0x92), value: 1.0000, at position: 33  
1, Key candidate: 246 (0xF6), value: -0.4530, at position: 211  
2, Key candidate: 192 (0xC0), value: 0.4331, at position: 21
```

```
3, Key candidate: 137 (0x89), value: -0.4306, at position: 33
Best correlation key 2:
0, Key candidate: 156 (0x9C), value: 1.0000, at position: 34
1, Key candidate: 155 (0x9B), value: -0.4477, at position: 283
2, Key candidate: 13 (0x0D), value: 0.4199, at position: 30
3, Key candidate: 131 (0x83), value: -0.4195, at position: 34
Best correlation key 3:
0, Key candidate: 60 (0x3C), value: 1.0000, at position: 35
1, Key candidate: 44 (0x2C), value: 0.5204, at position: 35
2, Key candidate: 146 (0x92), value: -0.4707, at position: 35
3, Key candidate: 25 (0x19), value: 0.4110, at position: 68
Best correlation key 4:
0, Key candidate: 3 (0x03), value: 1.0000, at position: 36
1, Key candidate: 115 (0x73), value: -0.4178, at position: 40
2, Key candidate: 47 (0x2F), value: 0.4137, at position: 198
3, Key candidate: 122 (0x7A), value: 0.4107, at position: 7
Best correlation key 5:
0, Key candidate: 47 (0x2F), value: 1.0000, at position: 37
1, Key candidate: 251 (0xFB), value: -0.3895, at position: 41
2, Key candidate: 45 (0x2D), value: 0.3867, at position: 156
3, Key candidate: 160 (0xA0), value: 0.3855, at position: 332
Best correlation key 6:
0, Key candidate: 141 (0x8D), value: 1.0000, at position: 38
1, Key candidate: 116 (0x74), value: 0.4490, at position: 344
2, Key candidate: 29 (0x1D), value: 0.4365, at position: 281
3, Key candidate: 127 (0x7F), value: -0.4104, at position: 235
Best correlation key 7:
0, Key candidate: 201 (0xC9), value: 1.0000, at position: 39
1, Key candidate: 222 (0xDE), value: 0.4605, at position: 165
2, Key candidate: 52 (0x34), value: -0.4454, at position: 292
3, Key candidate: 213 (0xD5), value: 0.4443, at position: 187
Key entropy: 256
Key info:
Known subkeys:
    Masked k0: 4adbd9fa641217c
    Masked k1: a8c104981d05747f
    Masked k2: 5c929c3c032f8dc9
```

Entropy too large to print or bruteforce key

In the last step of the attack, the listing of masked keys is followed by a candidate for the complete key. After a 256-bit attack, this results in the listing below:

```
Best correlation key 0:
0, Key candidate: 6 (0x06), value: 1.0000, at position: 80
1, Key candidate: 103 (0x67), value: 0.4251, at position: 128
2, Key candidate: 222 (0xDE), value: 0.4093, at position: 177
3, Key candidate: 36 (0x24), value: 0.4067, at position: 85
Best correlation key 1:
0, Key candidate: 11 (0x0B), value: 1.0000, at position: 81
1, Key candidate: 130 (0x82), value: 0.5008, at position: 164
2, Key candidate: 120 (0x78), value: 0.4158, at position: 136
3, Key candidate: 136 (0x88), value: -0.4153, at position: 81
Best correlation key 2:
0, Key candidate: 220 (0xDC), value: 1.0000, at position: 82
1, Key candidate: 231 (0xE7), value: -0.4372, at position: 175
2, Key candidate: 139 (0x8B), value: 0.4334, at position: 324
3, Key candidate: 76 (0x4C), value: 0.4284, at position: 150
Best correlation key 3:
```

```
0, Key candidate: 43 (0x2B), value: 1.0000, at position: 83
1, Key candidate: 241 (0xF1), value: 0.4230, at position: 266
2, Key candidate: 95 (0x5F), value: 0.4212, at position: 83
3, Key candidate: 7 (0x07), value: -0.4018, at position: 72
Best correlation key 4:
0, Key candidate: 109 (0x6D), value: 1.0000, at position: 84
1, Key candidate: 9 (0x09), value: -0.4255, at position: 1
2, Key candidate: 58 (0x3A), value: 0.4195, at position: 340
3, Key candidate: 98 (0x62), value: 0.4171, at position: 7
Best correlation key 5:
0, Key candidate: 144 (0x90), value: 1.0000, at position: 85
1, Key candidate: 139 (0x8B), value: 0.4154, at position: 244
2, Key candidate: 155 (0x9B), value: 0.4123, at position: 29
3, Key candidate: 10 (0x0A), value: 0.4091, at position: 298
Best correlation key 6:
0, Key candidate: 100 (0x64), value: 1.0000, at position: 86
1, Key candidate: 64 (0x40), value: 0.4238, at position: 362
2, Key candidate: 23 (0x17), value: 0.4046, at position: 88
3, Key candidate: 226 (0xE2), value: -0.4039, at position: 332
Best correlation key 7:
0, Key candidate: 15 (0x0F), value: 1.0000, at position: 87
1, Key candidate: 181 (0xB5), value: -0.4492, at position: 40
2, Key candidate: 76 (0x4C), value: -0.4422, at position: 234
3, Key candidate: 0 (0x00), value: -0.4203, at position: 143
Key entropy: 0
Key info:
Known subkeys:
Masked k0: 4adbd9fa641217c
Masked k1: a8c104981d05747f
Masked k2: 5c929c3c032f8dc9
Masked k3: 98770001f53b3245
Masked k4: 24a1d7bfec44e4c4
Masked k5: 60bdc2b6d90640f
Remaining key candidates: 1
Remaining key candidate:
6c364e6a8eab39b498d87d8a931d9a0450826149a4ad1b0968fa015efb16cc4d
```

K.3.3 Validation

After the module attempts to compute the complete key, a validation step is applied in order to provide a hint to the analyst on whether the key is correct or not. For Camellia, this validation consists of computing the key schedule based on the resulting key(s) and testing it against the keys obtained with DPA.

For 128 bit keys, this results on a list with either 0 or 2 key candidates being returned to the DifferentialAnalysis module. Most likely, the result will be an empty list if the DPA attack did not succeed.

However, when 2 candidates are returned, the DifferentialAnalysis module will try to verify if input and output are available. Otherwise it will print the two candidates. The listing below shows the result when two candidates are returned and input and output are known:

```
WARNING: Key verification failed, recovered keys do not create round keys matching
DPA results.
Key entropy: 1
Key info:
```

```
Known subkeys:  
Masked k0: 2ccc63354d77595  
Masked k1: 93cf467ed121a206  
Masked k2: f98825e0fc61f4e7  
Masked k3: c0b4faac9aacbc5  
Remaining key candidates: 2
```

```
Bruteforcing keys...  
Key could not be verified. Retry attack.
```

Finally, the following listing shows the output when no candidates are found:

```
WARNING: Key verification failed, recovered keys do not create round keys matching  
DPA results.  
Key entropy: 128  
Key info:  
Known subkeys:  
Masked k0: 2ccc63354d77595  
Masked k1: 93cf467ed121a206  
Masked k2: f98825e0fc61f4e7  
Masked k3: 78978353de855491  
Remaining key candidates: 0  
  
Entropy too large to print or bruteforce key (128)
```

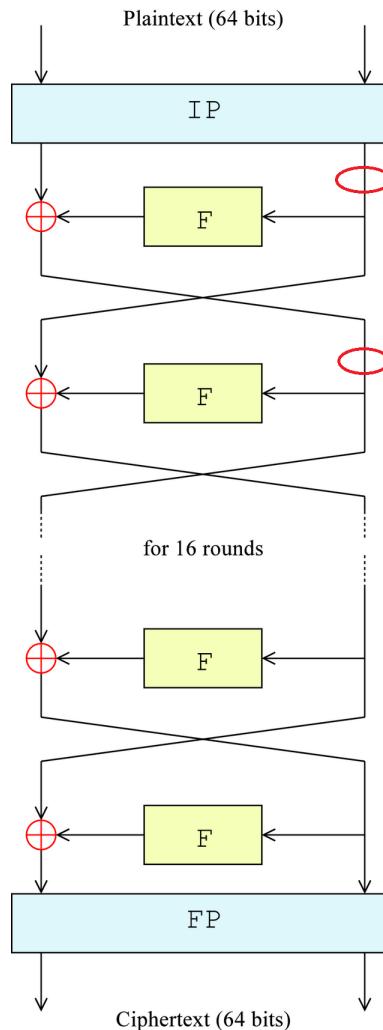
For 192 and 256 bit keys, only one key is returned, which will be printed by the Differential Analysis module. The same sort of verification process is applied, and a similar warning is shown when it is not successful as shown below:

```
WARNING: Key verification failed, recovered key does not create round keys matching  
DPA results.  
Key entropy: 0  
Key info:  
Known subkeys:  
Masked k0: 4adb9fa641217c  
Masked k1: a8c104981d05747f  
Masked k2: 5c929c3c032f8dc9  
Masked k3: 98770001f53b3245  
Masked k4: 24a1d7bfec44e4c4  
Masked k5: bb6a8b553a0a4016  
Remaining key candidates: 1  
  
Remaining key candidate:  
d8c2cd179d22db733035eb0cd1a535980fbb08a0a2573c1aad755085d61a493c
```

K.4 DES

DES is a block cipher that was selected as a FIPS standard for the U.S in 1976. DES uses a block size of 64 bits and a key size of 56 bits, which is usually written as a 64 bit key where the lowest bit in each byte represents a parity bit or is simply ignored. It consists of 16 rounds in a structure known as a Feistel Network.

Figure K.7, “The Feistel structure of DES” shows the Feistel structure of DES.

Figure K.7. The Feistel structure of DES

Besides the original DES mode, several 3DES modes have been defined. 3DES is a modification of DES consisting of 3 DES operations: Encrypt-Decrypt-Encrypt or Decrypt-Encrypt-Decrypt depending on the mode. 3DES can use 112 bit keys (first and last key are set to the same value) or 168 keys (3 independent DES keys).

The current implementation is capable of attacking all 3DES modes as well as DES.

References

<http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>.

K.4.1 Analysis steps

Two rounds need to be analyzed for each DES operation with independent keys. This means analyzing 2 rounds for single DES mode, 4 rounds for 2-key 3DES (2TDES) and 6 for 3-key 3DES (3TDES). Further, if both input and output are known, the last round attack is substituted by an 8 bit brute force on the remaining key bits.

For each round, several types of targets are defined.

- "HD SBox in/out shift Y round X" targets the Hamming Distance between the input and the output of the S-boxes in round X. Because the input of an S-box consists of 6 bits whereas the output consists of 4 bits, the shifted value Y defines which 4 consecutive bits of the input is used to calculate the Hamming Distance against the output: Y is the offset of the selected bits counting from the least significant bit. For example, denoting the input bits using (b5, b4, b3, b2, b1, b0) and the output using (a3, a2, a1, a0) (most significant, ..., least significant), Y = 2 means Hamming Distance $\text{HD}((b5, b4, b3, b2), (a3, a2, a1, a0))$ is targeted, while Y = 0 means $\text{HD}((b3, b2, b1, b0), (a3, a2, a1, a0))$ is targeted.
- "HD left/right round X" targets the Hamming Distance between the 32-bit block that enters to the F function in round X (namely the right half of the round input), and the XOR of the output of the F function and the other 32-bit block of the input of round X (namely the left half of the round output). For example, the Hamming Distance between the two circled intermediate values shown in Figure K.7, "The Feistel structure of DES" corresponds to attack option "HD left/right round 1 (using input)".
- "HW SBox out round X" targets the Hamming Weight of the output of the S-boxes in round X.

The Feistel structure of DES ensures that the encryption and decryption are identical processes (but with reversed subkeys). Therefore, unlike e.g. AES the listed targets for DES are valid for both encryption and decryption.

K.4.2 Output

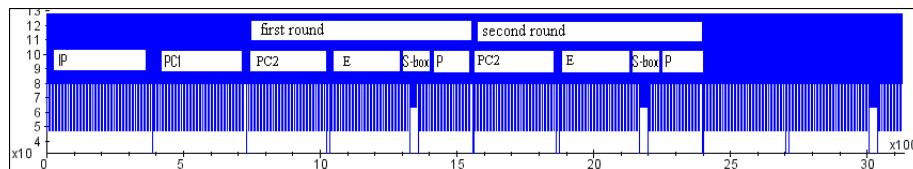
Using correlation to analyse a DES implementation, it may be possible to retrieve the complete DES key. The Inspector crypto modules take the following approach:

1. Intermediate crypto data is computed. For each S-box, a resulting bit (or nibble) is computed for each candidate sub key (64 options) and the available input data.
2. Correlation is computed between the available samples and the intermediate crypto data resulting in 8×64 correlation traces.
3. The correlation traces are scanned for the highest peaks, which indicate the correctness of the related candidate sub key.
4. The 8 best correlating sub keys are combined in a round key.
5. The process is repeated for a second round, to obtain another round key. The two round keys provide enough key material to reconstruct the entire DES key.
6. Optionally, the process can be repeated for a subsequent DES execution (in case of Triple DES).

Example

The file *DESSimulation.trs* contains 100 simulated power consumption traces, as well as the randomly chosen DES input and output data. Since the simulated traces do not contain noise, they clearly demonstrate the statistical properties. Note that real-life traces will contain much noise that will significantly hamper the analysis. An analyst will have to use many more traces and align them in order to achieve any practical results.

Per algorithm step, the *DESSimulation.trs* traces can be analysed fairly easily.



The labels in the graph show that the entire process can be divided into rounds with the respective DES permutations and substitutions. A proper target for differential analysis is the S-box or P permutation.

At completion, the log window shows the best correlating key candidates per S-Box:

```
Best correlation S-Box 1:  
sub key: 18, value: 0.4746, at position: 1334  
sub key: 48, value: 0.2182, at position: 1334  
sub key: 27, value: 0.2146, at position: 1334  
sub key: 33, value: 0.2062, at position: 1334  
Best correlation S-Box 2:  
sub key: 40, value: 0.4936, at position: 1337  
sub key: 33, value: 0.4293, at position: 1337  
sub key: 34, value: 0.4012, at position: 1337  
sub key: 22, value: 0.3410, at position: 1337
```

A correct candidate shows a significant difference in correlation between the best and second-best candidate. For instance, take the difference between 0.47 (sub key 18) and 0.21 (sub key 48) in the first S-Box.

When the first (or last) round key is found, the analysis can be repeated for the second (or last but one) round. You indicate this in the dialogue, along with the key value of the previous round that you have just found. After the second analysis, both round keys are compared and a resulting DES key is computed. Inspector detects bad analysis results by looking for mismatches between duplicated round key bits.

Related tutorials

- **Des Analysis tutorial** [\[../tutorials/sectionsDESSimTutorial.html\]](#)[\[PDF\]](#) [\[../tutorials/Tutorials.pdf#sectionsDESSimTutorial\]](#)

K.5 DSA

DSA is a signature scheme using public key cryptography. The algorithm first uses modular exponentiation of a random number, resulting in a number 'r'. Then it multiplies this number with the key, resulting in a number 'product'. Next, the message is hashed, and the hash is mixed with 'product', resulting in a number 's'. The signature is a concatenation of 'r' and 's'. The attack targets the multiplication of the key with 'r' (which is part of the output), and assumes that the multiplication is performed byte wise.

References

http://en.wikipedia.org/wiki/Digital_Signature_Algorithm.

K.5.1 Analysis steps

The attack targets the multiplication result bytes one by one, and considers each key byte multiplication as a round. Leakage is observed as the hamming weight of the multiplication

results. The attack starts with attacking the LSB (Least Significant Byte) of the key, which corresponds to the highest index of the key byte sequence. After retrieval of all the key bytes the key is reconstructed by a simple concatenation of these bytes.

K.5.2 Output

For each of the steps, the module outputs the best matching candidates for 8-bit blocks of the key. For the following example, a simulated trace set was used.

```
Best correlation subKey[19]:  
0, subKey[19]=14, value: 1.0000, at position: 0  
1, subKey[19]=0a, value: 0.9422, at position: 0  
2, subKey[19]=8a, value: 0.9329, at position: 0  
3, subKey[19]=28, value: 0.9244, at position: 0  
Key entropy: 152  
Key info:  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  
xxxxxxxxxxxxxxxxxxxxxx00010100/152 bits entropy  
(0x0000000000000000000000000000000000000014)  
Entropy too large to print or bruteforce key
```

Now that the final key byte (19) is known, the module can attack key byte 18.

During the analysis it may happen that a key byte of 0 is present. Note that a multiplication with 0 is always 0 for any data byte. Sometimes correlation with this intermediate is found, due to possible carries from other intermediates:

```
Key targets: subKey[15] => HW(mulState[15][00])  
Best correlation subKey[15]:  
0, subKey[15]=00, value: 1.0000, at position: 15  
1, subKey[15]=2b, value: 0.4508, at position: 387  
2, subKey[15]=3d, value: 0.3867, at position: 370  
3, subKey[15]=3c, value: 0.3836, at position: 324  
Key entropy: 120  
Key info:  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx000000  
0011001010111111010111010101111010/120 bits entropy  
(0x0000000000000000000000000000000000000000cafebab)
```

In other cases it may happen that it is not found; the correlation for key candidate 0 is very low. See e.g. the following output where 0 is not even present:

```
Attacking: HW key[14]*data[00] (using output)  
Key targets: subKey[14] => HW(mulState[14][00])  
Best correlation subKey[14]:  
0, subKey[14]=be, value: 0.8649, at position: 380  
1, subKey[14]=5f, value: 0.8090, at position: 380  
2, subKey[14]=cb, value: 0.5889, at position: 16  
3, subKey[14]=39, value: -0.4163, at position: 279  
Key entropy: 112  
Key info:  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx10111110000000  
00110010101111110101110101011110/112 bits entropy  
(0x000000000000000000000000000000be00cafebabe)  
Entropy too large to print or bruteforce key candidates
```

In this case there are two possibilities: manually setting this key byte to 0 using the UI, or trying to attack the same key byte again with another data byte:

```
Attacking: HW key[14]*data[19] (using output)  
Key targets: subKey[14] => HW(mulState[14][19])  
Best correlation subKey[14]:  
0, subKey[14]=00, value: 1.0000, at position: 299  
1, subKey[14]=80, value: 0.8910, at position: 299  
2, subKey[14]=40, value: 0.7658, at position: 337  
3, subKey[14]=c0, value: 0.7506, at position: 299  
Key entropy: 112  
Key info:  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx00000000000000  
00110010101111110101110101011110/112 bits entropy  
(0x00000000000000000000000000000000cafebabe)  
Entropy too large to print or bruteforce key candidates
```

K.6 ECDSA

ECDSA is a signature scheme using elliptic curve public key cryptography. It is quite similar to DSA, but uses scalar multiplication rather than modular exponentiation. The algorithm first uses elliptic curve scalar multiplication of a random number with a base point, resulting in a number 'r' (x-coordinate of resulting point). Then it multiplies this number with the key, resulting in a number 'product'. Next, the message is hashed, and the hash is mixed with 'product', resulting in a number 's'. The signature is a concatenation of 'r' and 's'. The attack targets the multiplication of the key with 'r' (which is part of the output), and assumes that the multiplication is performed byte wise.

References

http://en.wikipedia.org/wiki/Elliptic_Curve_DSA.

K.6.1 Analysis steps

The attack targets the multiplication result bytes one by one, and considers each key byte multiplication as a round. Leakage is observed as the hamming weight of the multiplication results. The attack starts with attacking the LSB (Least Significant Byte) of the key, which corresponds to the highest index of the key byte sequence. After retrieval of all the key bytes the key is reconstructed by a simple concatenation of these bytes. The process of this attack is identical to the DSA attack.

K.7 HMAC

HMAC (Hash-based Message Authentication Code), is a specific construction for calculating a message authentication code (MAC) involving a cryptographic hash function in combination with a secret key. It may be used to simultaneously verify both the data integrity and the

authenticity of a message. The module currently supports SHA1 only as the underlying hash algorithm.

The following pseudocode demonstrates how HMAC may be implemented.

```
function hmac (key, message)
    if (length(key) > blocksize) then
        key = hash(key) // keys longer than blocksize are shortened
    else if (length(key) < blocksize) then
        key = key || zeroes(blocksize - length(key)) // keys shorter than blocksize
        are zero-padded
    end if

    opad = [0x5c * blocksize] ^ key // Where blocksize is that of the underlying hash
    function
    ipad = [0x36 * blocksize] ^ key // Where ^ is exclusive or (XOR)

    return hash(opad || hash(ipad || message)) // Where || is concatenation
end function
```

Figure K.8. SHA1-HMAC overview.



K.7.1 Analysis steps

Since the original key is hashed, and its result is used as a key, it is not possible to recover its original value. Instead, two hashes will be recovered.

In order to recover the first hash value, we attack the first 4 rounds of the SHA1 involving the message, where the hash is combined with the message.

The following pseudocode represents the algorithm steps for the first 4 rounds

```
for i from 0 to 3
    f = (b and c) or ((not b) and d)
    k = 0x5A827999
    temp = (a leftrotate 5) + f + e + k + w[i]
    e = d
    d = c
    c = b leftrotate 30
    b = a
    a = temp
```

The w array holds the message, while a, b, c, d, e hold the hashed IPAD value. The following intermediates are then attacked in order to recover the key

- temp_0
- temp_1
- f_3

- f_2
- temp_2

For each step, a combination of several parts of the key is recovered. Once all the steps are completed, the system can be solved and the hashed key is printed

Since the operations involve an addition the key is recovered byte-wise: this means 20 steps are required for both the IPAD and OPAD

Due to the high linearity of the targeted operations, the difference in correlation between correct and wrong candidates is very low, especially while targeting the f function, therefore many traces are usually needed.

Experimental results show that enabling the CC option and using the "sum" distinguisher instead of "peak" often improve the results.

Once the value of the IPAD has been recovered, the input message for the second hash can be computed. The module then iterates the same process on the final hash to recover OPAD

Note on step 4

When attacking round 4, the following operation is targeted: $f = (t_0 \text{ and } (h_0 \text{ leftrotate } 30)) \text{ or } (!t_0 \text{ and } (h_1 \text{ leftrotate } 30))$

This means that when a byte of h_0 is the same as the byte in h_1 , the result of the operation is always constant and does not depend on the input. If this is the case, no correlation will be reported in that area. The analyst needs to check whether the correlation of step 4 is as strong as the correlation in step 3 for each of the 4 key bytes. Further, also the position where the correlation peaks are found should be checked. If no peaks appear in the expected location, probably two key bytes are the same. If this is the case, it is likely that the wrong key byte is guessed. In order to overcome this issue, the analyst can manually set the key byte value. As a further verification, if some of the bytes are not correctly guessed, the correlation of one or more of the last 4 key bytes will be much lower than the others.

K.7.2 Analysis settings

This algorithm offers the following settings:

- Word size: The word size to attack, either 8 or 32 bits. The module always attacks intermediate keys byte by byte. However, when 32 bits are selected, the values obtained for previous bytes of the same word are incorporated to the intermediate values. Thus, this option improves the correlation results when the target processor uses 32 bit arithmetics.
- Input size: This is the size of the input message to the HMAC implementation. Since HMACs can be applied to messages of arbitrary length, the analysis requires knowledge of the input size.

K.7.3 Output

After the first step, the attack produces a listing providing the best candidates for each the first IPAD key byte:

Algorithm: HMAC-SHA1 (32 bits)

```
Possible attacks: [HW T 0 step 1 keybyte 0 (using input)]
Performing: HW T 0 step 1 keybyte 0 (using input)
Best correlation:
0, Key candidate: 108 (0x6C), value: 0.4257, at position: 0
1, Key candidate: 236 (0xEC), value: 0.3310, at position: 0
2, Key candidate: 136 (0x88), value: -0.3117, at position: 3
3, Key candidate: 221 (0xDD), value: 0.3111, at position: 3
Key entropy: 320
Key info:
Masked H0 IPAD: xxxxxxxx6c
Masked H1 IPAD: xxxxxxxxx
Masked H2 IPAD: xxxxxxxxx
Masked H3 IPAD: xxxxxxxxx
Masked H4 IPAD: xxxxxxxxx
Masked H0 OPAD: xxxxxxxxx
Masked H1 OPAD: xxxxxxxxx
Masked H2 OPAD: xxxxxxxxx
Masked H3 OPAD: xxxxxxxxx
Masked H4 OPAD: xxxxxxxxx
```

Each H value is 32 bits long, and the module recovers 8 bytes at a time.

If the *Traces* checkbox was selected, in addition to this listing the module produces a set of correlation traces.

A similar output is produced for subsequent steps. A listing with the best candidates for the next key byte is presented. For instance, the following listing shows the output of the second step of the attack:

```
Algorithm: HMAC-SHA1 (32 bits)
Possible attacks: [HW T 0 step 1 keybyte 1 (using input), HW T 0 step 1 keybyte 0
(using input)]
Performing: HW T 0 step 1 keybyte 1 (using input)
Best correlation:
0, Key candidate: 113 (0x71), value: 0.6764, at position: 0
1, Key candidate: 241 (0xF1), value: 0.5740, at position: 0
2, Key candidate: 49 (0x31), value: 0.5492, at position: 0
3, Key candidate: 177 (0xB1), value: 0.5376, at position: 0
Key entropy: 320
Key info:
Masked H0 IPAD: xxxx716c
Masked H1 IPAD: xxxxxxxxx
Masked H2 IPAD: xxxxxxxxx
Masked H3 IPAD: xxxxxxxxx
Masked H4 IPAD: xxxxxxxxx
Masked H0 OPAD: xxxxxxxxx
Masked H1 OPAD: xxxxxxxxx
Masked H2 OPAD: xxxxxxxxx
Masked H3 OPAD: xxxxxxxxx
Masked H4 OPAD: xxxxxxxxx
```

In the 20th step of the attack, the value of the IPAD key is recovered and printed:

```
Performing: HW T 2 step 1 keybyte 19 (using input)
Best correlation:
0, Key candidate: 19 (0x13), value: 1.0000, at position: 4
1, Key candidate: 147 (0x93), value: 0.9397, at position: 4
2, Key candidate: 211 (0xD3), value: 0.9107, at position: 4
```

```
3, Key candidate: 83 (0x53), value: 0.9084, at position: 4
Key entropy: 160
Key info:
H0 IPAD: 91cd14b4
H1 IPAD: 50715a44
H2 IPAD: 13db60c8
H3 IPAD: 7d2027d2
H4 IPAD: 3f2ffb6f
Masked H0 OPAD: xxxxxxxx
Masked H1 OPAD: xxxxxxxx
Masked H2 OPAD: xxxxxxxx
Masked H3 OPAD: xxxxxxxx
Masked H4 OPAD: xxxxxxxx
```

Finally, in the last step of the attack for the OPAD key, the full key is recovered:

```
Best correlation:
0, Key candidate: 32 (0x20), value: 1.0000, at position: 9
1, Key candidate: 160 (0xA0), value: 0.9384, at position: 9
2, Key candidate: 224 (0xE0), value: 0.9088, at position: 9
3, Key candidate: 96 (0x60), value: 0.9009, at position: 9
Key entropy: 0
Key info:
H0 IPAD: 91cd14b4
H1 IPAD: 50715a44
H2 IPAD: 13db60c8
H3 IPAD: 7d2027d2
H4 IPAD: 3f2ffb6f
H0 OPAD: a3cae248
H1 OPAD: ade7f811
H2 OPAD: 20e4a51e
H3 OPAD: fdc2e7d2
H4 OPAD: 471776f8

Remaining key candidate: 91cd14b450715a4413db60c87d2027d23f2ffb6f
a3cae248ade7f81120e4a51efdc2e7d2471776f8
```

K.7.4 Validation

If input and output are known, the module verifies whether the recovered key generates the expected output. The validity of the IPAD key can be verified if correlation peaks are found for the OPAD attack, since the result of the IPAD+message hash is used as input for the OPAD hash.

K.8 RSA CRT

RSA is an algorithm for public-key cryptography that is based on the presumed difficulty of factoring large integers, the factoring problem.

Efficient implementations of RSA on computationally limited devices, such as smartcards, often use the Chinese Remainder Theorem (CRT) technique in combination with Garner's algorithm in order to make the computation of modular exponentiation as fast as possible.

References

Wikipedia [[http://en.wikipedia.org/wiki/RSA_\(algorithm\)](http://en.wikipedia.org/wiki/RSA_(algorithm))]

K.8.1 Analysis steps

Consider the RSA computation: $m = c^d \bmod n$

This formula breaks down to 3 phases in CRT Garner's algorithm:

1. Reduction: $c_p = c \bmod p$, $c_q = c \bmod q$, $d_p = d \bmod p-1$, $d_q = d \bmod q-1$, $p_{inv} = p^{-1} \bmod q$
2. Exponentiation: $m_p = c_p^{d_p} \bmod p$, $m_q = c_q^{d_q} \bmod q$
3. Recombination: $m_{qp} = m_q - m_p$, $h = p_{inv} * m_{qp} \bmod q$, $m = m_p + h * p$

A CRT implementation can normally be recognized in the power consumption profile by the two relatively long, equally sized, sections of increased power consumption.

This analysis method focuses on the *Recombination* phase, particularly where intermediate result h is multiplied by p . The product of h and p is quite close to the output message m because the length of m_p is only a half the length of m . Therefore we can use m as an estimation of $h*p$ and, if we know m , we can estimate $h \approx m/p$ by neglecting m_p .

Suppose that the most significant bytes p_{ms} of p is known. Then we can estimate the most significant bytes of h as: $h_{ms} \approx m/p_{ms}$. Now, we collect a significant number of side channel traces of the Recombination phase while we store the respective output values m with each trace.

If we knew for each trace the value of h we could have correlated these values with the samples taken from the Recombination phase and found out where h is used. To predict h , we first guess the first (most significant) byte of p , then estimate the corresponding first byte of h . Now, for each of the 256 possible values of the first byte of p we can estimate the first byte of h , and compute a correlation trace. The strongest correlation found in the 256 correlation traces yields the most likely value of the first byte of p .

Next, we use the value found for the first byte of p in a subsequent invocation where we vary the second byte of p to estimate the second byte of h and compute the corresponding correlation traces. By repeating the above procedure we recover p byte by byte until it is almost complete. The last few bytes of p are more difficult to recover as the estimation $h \approx m/p$ is disturbed by neglecting m_p .

With most of the byte of p known, we can brute force the remaining byte of p and obtain q using the following constraints:

- p is a prime number
- Modulus $n = p * q$ is a part of the public key and n must be divisible by p
- q is also a prime number
- A prime number must be in the form of $30*k + i$, with k being an integer and i being a co-prime of 30, i.e. $i = (1, 7, 11, 13, 17, 19, 23, 29)$. Reference: Wikipedia [http://en.wikipedia.org/wiki/Primality_test].

Once p and q are known, the private exponent d can be calculated by solving $e * d = 1 \bmod lcm(p-1, q-1)$ ¹, where e is the public key exponent.

¹Here we use modulus $lcm(p-1, q-1)$ according to the PKCS#1 [<http://en.wikipedia.org/wiki/PKCS1>] standard. For algorithms where other modulus are used, the private exponent reported by our attack may not be correct, but the actual private exponent d can be derived by the user based on the primes p and q found in our attack.

Given an n -byte prime p , the analysis takes $n-1$ steps. At each step 1 bytes of p is recovered. During the last step, a brute force is performed to test the last 11 bits of p according to the constraints listed above. (Here, we choose to test the last 11 bits of p instead of only the remaining 8 bits for the purpose of correcting the errors in the last recovered byte of p . See below for details.) Once the primes are found, the module automatically derives the private exponent d and verifies it using the input/output data given by the traces.

Correcting recovered byte

Since the attack is based on dividing the full m by the most significant part of p , the byte of p we find in every step of the attack is an upper bound of the actual byte of p . Sometimes this upper bound is the actual byte of p , but often the actual byte is smaller than this upper bound recovered by the attack. In case of the latter, we need to correct the last found byte of p before we can recover the next byte of p .

Cases that the $(i-1)$ -th byte of p might be incorrect can be recognized during the attempts to recover the i -th byte of p . Since all the candidates of the i -th byte are tested based on the same (incorrect) value assumed for the $(i-1)$ -th byte, we would observe that the resulting correlation values are low and close to each other for all the candidates of the i -th byte. In this case, it is suggested to lower the value of $(i-1)$ -th byte of p by 1, and repeat the attack for the i -th byte, until a distinctive correlation occurs for a particular candidate of the i -th byte.

For practical examples of DPA on RSA CRT, please refer to the following tutorials xxx:

- **RSA CRT Crypto2 Tutorial** [..../tutorials/sectionsRSACRTCrypto2Tutorial.html] ([PDF]) [..../tutorials/Tutorials.pdf#sectionsRSACRTCrypto2Tutorial]

K.8.2 Output

For each of the steps, the module outputs the best matching candidates for a byte of the prime p . Below shows the attack result for the first byte of p . Note that the partial key ignores the last 3 bits of uncovered bytes for they may be erroneous and require correction.

```
Results after 1780 traces
Best correlation p byte 0:
rank: 1, candidate: 252 (0xFC), confidence: 0.5014 at position: 2285
rank: 2, candidate: 250 (0xFA), confidence: 0.4912 at position: 2355
rank: 3, candidate: 249 (0xF9), confidence: 0.4580 at position: 2355
rank: 4, candidate: 251 (0xFB), confidence: 0.4514 at position: 2355
Key entropy: 246
Key info:
Partial key private prime: 11111xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx/249 bits entropy (0xf80000000000
000000000000000000000000000000000000000000000000000000000000)
Entropy too large to print or bruteforce key candidates
```

For the last step, the module also brute-forces the last 11 bits of p and derives d :

```
Results after 1780 traces
Best correlation p byte 30:
rank: 1, candidate: 107 (0x6B), confidence: 0.2912 at position: 2223
rank: 2, candidate: 106 (0x6A), confidence: 0.2810 at position: 2222
rank: 3, candidate: 108 (0x6C), confidence: 0.2317 at position: 2223
rank: 4, candidate: 109 (0x6D), confidence: 0.2111 at position: 2223
Key entropy: 9
```

Key info:

```
Partial key private prime: 1111110000011100110010000101100000000000111101001011  
001011011101100100101101100010010011110001101000000000011110010100101000  
111111011001000000001010101110110111010000101101011110101011010010001100  
00111110110000100101101011100001101xxxxxxxxx/9 bits entropy (0xfc1cc85800f4b2  
dd92dbb1278d003ca51fb200abb6e85adead230fb096b86800)
```

Trying key candidates...

```
Correct key found: 5d628e7e1508bc153dc6959cb81f9beca6c3b1e0fd172e167c92f39a4f9e  
376ca0c302fb8b873023c383f89d69be679bf474b10e71a76d8092ced5e1a67ce60f
```

L Trace set coding

Side-Channel Analysis traces include supplementary information that may be stored along with the samples. Inspector supports three formats for the storage of this information:

1. Sequence of bytes (*.bytes); this is the default coding applied to unknown extensions. Each byte in the file represents one sample.
2. Sequence of floats (*.floats); each group of 4 bytes represents one little endian (LSB first) coded sample in float coding (IEEE 754).
3. A structured but flexible format (*.trs); the file starts with a header, followed by a trace block containing a set of traces.

The '.trs' format implements a proprietary encoding in which each object value is preceded by a tag field and optionally a length field (i.e. TLV format). This section explains the coding of this format.

Sequences of recorded traces are stored in a so-called trace set file. Each trace includes the following information:

- The recorded samples
- Optionally, recorded (cryptographic) data related to the analysis process
- Optionally, a local title that represents the meaning of the trace

The global data includes:

- Number of traces
- Number of samples per trace
- Length of cryptographic data included in trace
- Sample coding (e.g. type and length in bytes of each sample)
- Title space reserved for local title in each trace
- Global title representing a general name for each trace
- Description of the trace set
- Offset in X-axis (x value of first sample)
- Label of X-axis (unit, e.g. 'seconds')
- Label of Y-axis (unit, e.g. 'volt')
- Scale value for X-axis
- Scale value for Y-axis
- Preferred representation, linear or logarithmic

Design considerations

Two important requirements are incorporated in the design of this encoding format:

- *Flexibility*: The global information shall be coded in a flexible way, such that new types of information can be added without redefining the trace set file coding. This ensures that existing trace set files can still be read when new global data is added.
- *Performance*: The trace related data shall be randomly accessible to allow quick reading and browsing through the trace set.

The first requirement is met by using a TLV (Tag, Length, Value) structure for the file header. Each object is identified by a tag. Its content is preceded by a length field. Applications can simply generate / process the supported objects and ignore others. The second requirement is met by using a flat structure for all traces containing fixed spaces for titles, data and samples. This allows applications to access individual traces at random in a very fast manner.

Header coding

The trace set file header defines the following objects:

Table L.1. Trace set objects in the trace set header

| Tag | Name | M/O | Type | Length | Default | Meaning |
|-------------|------|-----|--------|----------|---------|---|
| 0x41 | NT | M | int | 4 | | Number of traces |
| 0x42 | NS | M | int | 4 | | Number of samples per trace |
| 0x43 | SC | M | byte | 1 | | Sample Coding (see table) |
| 0x44 | DS | O | short | 2 | 0 | Length of cryptographic data included in trace |
| 0x45 | TS | O | byte | 1 | 0 | Title space reserved per trace |
| 0x46 | GT | O | byte[] | variable | "trace" | Global trace title |
| 0x47 | DC | O | byte[] | variable | None | Description |
| 0x48 | XO | O | int | 4 | 0 | Offset in X-axis for trace representation |
| 0x49 | XL | O | byte[] | variable | None | Label of X-axis |
| 0x4A | YL | O | byte[] | variable | None | Label of Y-axis |
| 0x4B | XS | O | float | 4 | 1 | Scale value for X-axis |
| 0x4C | YS | O | float | 4 | 1 | Scale value for Y-axis |
| 0x4D | TO | O | int | 4 | 0 | Trace offset for displaying trace numbers |
| 0x4E | LS | O | byte | 1 | 0 | Logarithmic scale |
| 0x4F - 0x5E | - | - | - | - | - | Reserved for future use |
| 0x5F | TB | M | none | 0 | | Trace block marker: an empty TLV that marks the end of the header |

The object coding always starts with the tag byte. The object length is coded in one or more bytes. If bit 8 (msb) is set to '0', the remaining 7 bits indicate the length of the object. If bit 8 is set to '1', the remaining 7 bits indicate the number of additional bytes in the length field. These additional bytes define the length in little endian coding (LSB first). The content of the object is stored in the subsequent number of bytes, indicated by length. A trace set file contains at least the mandatory objects and may contain any of the optional fields. The TB object is always the

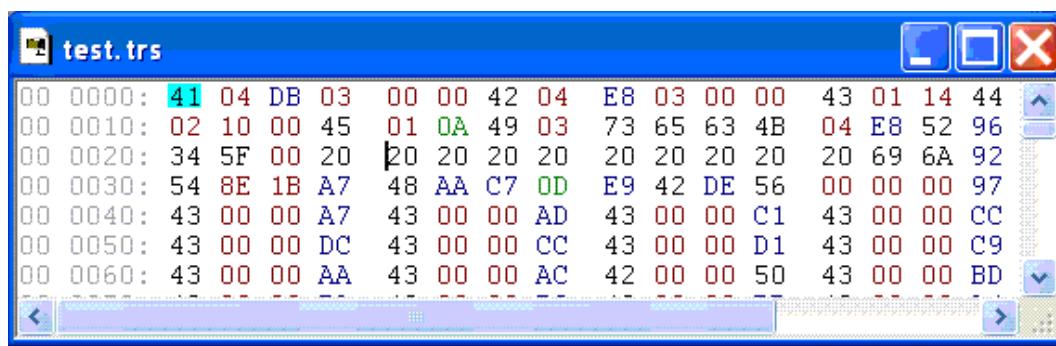
last object and marks the end of the header. The value of the numeric objects is coded little endian (LSB first). The float values use the IEEE 754 coding which is commonly supported by modern programming languages. Object SC defines the sample coding:

Table L.2. Sample coding

| | |
|---------|---|
| bit 8-6 | reserved, set to '000' |
| bit 5 | integer (0) or floating point (1) |
| bit 4-1 | Sample length in bytes (valid values are 1, 2, 4) |

Figure L.1, "Trace set example" shows the coding of a trace set file:

Figure L.1. Trace set example



The test.trs file contains the following objects:

0x41 (NT), length: 4, value: 0x3DB (987) number of traces

0x42 (NS), length: 4, value: 0x3E8 (1000) number of samples

0x43 (SC), length: 1, value: 0x14 (20), float coding, sampleSpace: 4 bytes per sample

0x44 (DS), length: 2, value: 0x10 (16), Data space: number of data bytes included

0x45 (TS), length: 1, value: 0x0A (10), 10 bytes title space per trace

0x49 (XL), length: 3, value: "sec", label X-axis

0x4B (XS), length: 4, value: 0x349652E8 (280E-9), time base of 280ns per sample

0x5F (TB), length: 0, beginning of trace block

987 times: 10 bytes space (title not present) 16 bytes data (e.g. 0x69 0x6A .. 0x56 0x00)
4000 bytes containing 1000 float samples of 4 bytes (e.g. 0x43970000 = 302)

Note that the header length is flexible, but always ends with the Trace Block Marker: 0x5F00.

Index

A

Abs module, 441
Acquisition devices, 41
Acquisition Protocol Wizard, 267
Acquisition2 Framework, 46
Acquisition2 Modules, 330
AES, 535
AES Acquisition module, 322
AES Advanced Analysis module, 368
AES Differential Fault Analysis module, 371
AES Known Key Analysis module, 373
AES Second Order Analysis module, 374
Align module, 334, 344
API, 507
Application Protocols, 524
ARIA, 537
AutoCorrelation module, 347
Average module, 489
AveragePlot module, 497

B

Binary module, 442

C

Camellia, 546
Chain module, 443
Cipher suites
 Ciphers, 535
CleanWave, 129
Communication devices, 41
Contents, 2
Coordinate Systems, 59
Correlation module, 349
CriExport module, 426
CriImport module, 427
CrossCorrelation module, 351
Current Probe, 87

D

Data Generator, 47
DataEdit module, 428
DataSort module, 429
DES, 551
DES Acquisition module, 323
DES Advanced Analysis module, 377
DES Differential Fault Analysis module, 381
DES Known Key Analysis module, 384
DES Masked Input Analysis module, 387
DES Partly Constant Analysis module, 388

DES Second Order Analysis module, 390
Device, 45
Diode Laser Station, 104
Distribution module, 490
drivers, 501
DSA, 554
DSA Acquisition module, 324
Dynamic Perturbation
 Conceptual overview, 56
 Protocols, 57
 Snippets/programs, 58
 Variables, 58

E

ECC Acquisition module, 324
ECCByteMultiplyAnalysis module, 393
ECDSA, 556
ECDSA Acquisition module, 325
ECNRPNonceAnalysis module, 395
ElasticAlign module, 337
ElasticAverage module, 340
ElasticRadius module, 341
EM Probe, 89
EM Probe Station, 88
EM-FI Transient Probe, 111
Embedded Protocol, 527
EP Perturbation after Reset Module, 485
Export module, 430

F

FI setup, 54
Field deflector, 91
FilterGuidance module, 444
Filters, 133
Frequency Band Decomposition Module, 446
Function modules, 11

G

General overview, 46
Gsm Acquisition module, 325

H

Harmonics module, 447
HMAC, 556
How do I implement a raw protocol?, 277
How do I implement an application protocol?, 267

I

icWaves, 113
icWaves configuration module, 493
Import module, 431
Installation and configuration, 23
InvNotchFilter, 448

IOCTL Acquisition module, 326
IVI-compliant oscilloscopes, 75

J

Java Card Targets, 135

K

Keyboard Shortcuts, 316

L

Legacy vs new acquisition, 40
Legacy vs new perturbation, 53
License, 28
LowPass module, 449

M

Matlab Export module, 432
Matlab import Module, 435
MicroPross MP300 TCL1/TCL2, 92
Microscope, 62
Migrating to the new acquisition, 40
Migrating to the new perturbation, 54
Module compatibility, 152
Module development, 150
Module development with Eclipse, 167
Module Methods, 153
MovingAverage module, 449
MP300 Device driver parameters, 317

N

Negate module, 450
Notify module, 491

O

Oscilloscope, 44
Oscilloscope module, 326
Oscilloscopes, 69
Overview, 1

P

PatternExtract module, 352
PatternMatch module, 354
PatternPad module, 451
PatternResample module, 360
PeakExtract module, 452
Perturbation Advanced Program Module, 483
Perturbation Module, 460
Power Tracer3, 79
Power Tracer4, 83
Preface, xvii
Prevent card damage using VC Glitcher and
icWaves, 129

Profiles, 18
Protocol Device, 502
Protocol phase, 57

R

Real-Time pattern matching, 139
Recover module, 436
Resample module, 363
Reverse module, 437
RF Tracer, 96
RFResample, 363
Round align, 343
RSA Acquisition module, 328
RSA CRT, 560
RSA High Order Analysis module, 405
RSA-CRT Differential Fault Analysis module, 400
RsaBinExpoAnalysis module, 397
RsaCorrelation module, 401
RsaCrtAnalysis module, 403
RsaNeighborCorrelation module, 410

S

Sample Card configuration module, 495
SampleEdit module, 437
Sasebo Acquisition module, 328
SC Perturbation after Reset Module, 485
SC Perturbation Module, 471
SC Perturbation with Glitch Program Module, 481
SC Single XYZ Perturbation Module, 476
Scan Area, 61
SEED Analysis module, 411
SegmentedChain module, 453
Select module, 439
Settings, 12
SideChannelAcquisition module, 319
Simulate module, 459
SmartCardOpticalPerturbation module, 465
SmartCardPerturbation module, 467
Spectral module, 454
SpectrallIntensity module, 498
Spectrogram module, 355
Spectrum module, 357
Split module, 439
Splitter, 109
Splitter configuration module, 494
StandardDeviation module, 492
Statistical Correction module, 455
Stretch module, 346
Sync Resample module, 365

T

TargetManager, 135
Targets

TargetManager, 135
Third party software, 28
Tools menu, 22
Trace set coding, 564
TraceMix module, 440
TraceSort module, 488
TrainingCardProtocol, 526
Transport Protocols, 520
Transpose module, 441
Triggered Perturbation, 55
TriggeredOpticalPerturbation module, 470
TriggeredPerturbation module, 470
Trim module, 441

U

UniqueData module, 358
USB Token Acquisition module, 330
User and System Space, 18
User interface, 8
Using the GPU computing power: CUDA, 25

V

VC Glitcher, 98, 507

W

WindowedResample module, 367
Writing perturbation modules, 174
Writing triggered perturbation modules, 181

X

XMEGA, 136, 136
XtalClear module, 457
XY(Z) Devices, 64
XYAcquisition module, 321