Ruhr-University Bochum, Germany
Department of Electrical Engineering and Information Sciences
Chair for Network and Data Security

Automatic Identification of Cryptographic Primitives in Software

Diploma Thesis of Felix Gröbert

First Examiner:	Prof. Dr. Jörg Schwenk (Ruhr-University Bochum)
Second Examiner:	Prof. Dr. Felix C. Freiling (University of Mannheim)
Supervisors:	Dr. Thorsten Holz (Vienna University of Technology) and
	Dipl. Inform. Carsten Willems (University of Mannheim)

February 7, 2010

Declaration / Erklärung

I hereby declare that the work presented in this thesis is my own work and that to the best of my knowledge it is original, except where indicated by references to other authors.

Hiermit versichere ich, dass ich meine Diplomarbeit eigenständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Bochum, February 7, 2010

Felix Gröbert

Contents

Contents i

List of Figures v

List of Tables vii

List of Code Listings ix

List of Algorithms xi

Acknowledgements xiii

Abstract / Zusammenfassung 1

1. Introduction 3
- 1.1. Motivation . . . 3
- 1.2. Thesis . . . 6
- 1.3. Assumptions and Limitations . . . 7
- 1.4. Contributions of this Work . . . 8
- 1.5. Outline . . . 8

2. Related Work 9
- 2.1. Static Approaches . . . 9
- 2.2. Finding Keys in Binary Data . . . 12
- 2.3. Dynamic Approaches . . . 12
- 2.4. Frameworks for Execution Tracing . . . 15
  - 2.4.1. Overview . . . 15
  - 2.4.2. Virtualization and Emulation . . . 16
  - 2.4.3. Dynamic Binary Instrumentation . . . 17
  - 2.4.4. Pin-focused Work . . . 17
- 2.5. Summary . . . 17

3. Prerequisites 19
- 3.1. Cryptographic Algorithms . . . 19
  - 3.1.1. Algorithmic Definition . . . 19
  - 3.1.2. Implemented Code . . . 21

3.1.3. Compiled Code 22
3.1.4. Algorithmic Composition 22
3.2. Execution Tracing 23
3.2.1. Data Reduction 24
3.2.2. Required Data 24
3.3. Basic Block Detection 25
3.4. Loop Detection 26
3.5. Control Flow Graph Generation 30
3.6. Memory Reconstruction 30
3.7. Summary 34

4. Observations and Methods **35**
4.1. Reducing of Code Search Space 35
4.2. Type of Identification 36
4.2.1. Signature-based Identification 36
4.2.2. Generic Identification 36
4.3. Observations 37
4.4. Identification Methods 39
4.4.1. Signature-based 40
4.4.2. Generic Bitwise Arithmetic-based 44
4.4.3. Generic Loop-based 46
4.4.4. Generic Memory-based 50
4.5. Summary 52

5. System Implementation **53**
5.1. Overview 53
5.2. Tracing 55
5.3. Analysis 58
5.3.1. Architecture 60
5.3.2. Process 62
5.4. Summary 63

6. Experimental Evaluation **65**
6.1. Evaluation Environment 65
6.2. Testing Applications 65
6.2.1. Published Methods 68
6.2.2. Signature-based Methods 70
6.2.3. Generic Methods 75
6.3. Off-the-Shelf Application 76
6.4. Modified Application 80
6.5. Summary 80

7. Conclusions **83**
7.1. Evaluation Performance 83

7.2. Further Work 84
7.3. Conclusion 86

Bibliography **87**

A. Digital Attachments **95**

B. Static Tools **97**

List of Figures

1.1. Auguste Kerckhoffs ⋆ 1835 † 1903 4

2.1. Cipher-block chaining mode 15

3.1. Control flow graph generated for the XOR application 31
3.2. Sliding window over candidate keys 34

4.1. OpenSSL implementation of DES 37
4.2. Beecrypt, Cryptopp, and OpenSSL implementations of MD5 38
4.3. Composition of sequence sets 40
4.4. Composition of (mnemonic, constant)-tuple datasets 42
4.5. Three-dimensional view of the loop differ 47
4.6. Verification of the algorithmic relation 51

5.1. Overview of the stages 53
5.2. Displaying an overview of implementation stage 1 54
5.3. Displaying an overview of implementation stage 2 55
5.4. Initialization of objects 62

6.1. Analysis and trace time/size costs, logarithmic scaled 66
6.2. Cumulative entropy for all testing applications 70
6.3. Memory constant identification method for all testing applications . . 72
6.4. Results of the signature matching using (mnemonic, constant)-tuples . 72
6.5. Results of the implementation signatures 73
6.6. Results of the unique signatures 74
6.7. The core XOR encryption packed with ASPack 81

List of Tables

2.1. Publicly available static tools 9
2.2. Detection performance among the static tools 11
2.3. False-positives among the static tools 11
2.4. Matches of the static tools for whole cryptographic libraries 11
2.5. Distinguishing between decryption and other routines by Wang et al. . . 13

3.1. Pseudo trace for example in Listing 3.1 29
3.2. Detected loops for the XOR test application 29
3.3. Shows an example for the memory block search algorithm 34

6.1. Overview of testing applications 67
6.2. Parameter size of testing applications 67
6.3. Analysis and trace time/size costs 68
6.4. Analysis performance for the published methods 69
6.5. Analysis performance for the signature-based identification methods . 71
6.6. Analysis performance for the generic identification methods 76
6.7. Analysis performance for the Curl trace 78

List of Code Listings

3.1. Sample loop code 28

4.1. API signatures 44
4.2. Loop differ check for XOR relations 48
4.3. Entropy calculation for the loop differ 48
4.4. Entropy check of the loop differ 49
4.5. Sample output of the XOR check 50

5.1. Running the Pintool 55
5.2. Example of the trace file 57
5.3. Running the analysis tool 58
5.4. Sample output of the analysis tool 59
5.5. Sample experimentation session using the analysis console 59

6.1. Intersecting constants of the implementations 71
6.2. Excerpt of found counters 75
6.3. Command line options for tracing Curl 77
6.4. Decrypted HTTPS client request 78
6.5. Detected function symbols in Curl 79
6.6. Sample output for the XOR check 80

List of Algorithms

3.1. Basic block detection 25
3.2. Loop detection 27
3.3. Control flow graph generation 30
3.4. Memory reconstruction 32
3.5. Recursive memory block search 33

4.1. Comparison of (mnemonic, constant)-tuple signatures 43
4.2. Wang et al. cumulative bitwise percentage method 45

Acknowledgements

First, I would like to thank Jörg Schwenk and Felix Freiling for giving me the opportunity to write this thesis. I also would like to thank Carsten Willems and Thorsten Holz for their endless support during endless conference calls. Without their knowledge this work would not been possible.

Special thanks go to my parents and my girlfriend, who supported my studies and always believed in me. Finally, I would like to thank my fellow students for our fruitful discussions.

Abstract / Zusammenfassung

In this thesis we research and implement methods to detect cryptographic algorithms and their parameters in software. Based on our observations on cryptographic code, we point out several inherent characteristics to design signature-based and generic identification methods. Using dynamic binary instrumentation, we record instructions of a program during runtime and create a fine-grained trace. We implement a trace analysis tool, which also provides methods to reconstruct high-level information from a trace, for example control flow graphs or loops, to detect cryptographic algorithms and their parameters. Using the results of this work, encrypted data, sent by a program for example, may be decrypted and used by an analyst to gain further insight on the behavior of the analyzed binary executable.

Diese Arbeit erforscht und implementiert Methoden zur Erkennung kryptographischer Algorithmen und deren Parameter in Software. Die bei der Untersuchung kryptographischer Programme festgestellten inhärenten Charakteristika dienen als Grundlage für die Entwicklung verschiedener signaturbasierter und generischer Identifikationsmethoden. Mittels dynamic binary instrumentation werden zuerst die Instruktionen eines Programms während der Laufzeit aufgezeichnet und ein fein-granuliertes Protokoll erstellt. Die Identifikationsmethoden werden dann durch ein Analysetool, das auch Strukturen wie Kontrollflussgraphen oder Schleifen rekonstruiert, auf das Protokoll angewandt, um die ausgeführten kryptographischen Algorithmen zu erkennen. Die Ergebnisse dieser Arbeit erleichtern es einem Softwareanalyst verschlüsselten Daten, welche beispielsweise durch ein Programm verschickt werden, automatisch zu entschlüsseln, um weitere Einblicke in das Verhalten des Programms zu erlangen.

Keywords: Code Analysis, Dynamic Binary Analysis, Instrumentation, Code Heuristics, Code Signatures, Applied Cryptography

Bibtex:

```
@mastersthesis{groebert2010,
Author = {Felix Gr{\"o}bert},
Title = {{Automatic Identification of Cryptographic Primitives in Software}},
School = {Ruhr-University Bochum, Germany},
Type = {Diplomarbeit},
Year = {2010}}
```

1 Chapter 1. Introduction

In this chapter we first motivate the thesis' objective from a historical perspective. We formulate a thesis and assumptions to further define the primary objective and conclude with the core contributions of the work. In the last section we describe the structure of the document.

1.1. Motivation

In 1883 the dutch cryptographer Auguste Kerckhoffs wrote the article "La cryptographie militaire" [39] and proposed a core principle of modern cryptography:

> *A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.*

For an encryption scheme, it means that the algorithm should be known to the adversary without yielding an advantage. This threat scenario is the common base of civilian cryptanalysis. Using peer-review, researchers evaluate the security of algorithms. Following Kerckhoffs' principle, researchers are not only able to determine weaknesses of algorithms, but also can rule out that an algorithm contains backdoors.

Since the 1970s, we have seen a constant development of cryptographic primitives like symmetric block and stream ciphers, asymmetric encryption schemes, and hash algorithms. While cryptographic primitives are used to fulfill the need for confidentiality, integrity, and authentication, they are also composed to more complex structures, i.e., cryptographic protocols, implemented in nearly all modern software prod-

Figure 1.1.: Auguste Kerckhoffs ⋆ 1835 † 1903

ucts. To verify the correct composition of primitives, one has to determine the protocol design and its implementation.

In the 1940s Claude Shannon rephrased Kerckhoffs' principle to:

> *The enemy knows the system.*

As the system encompasses the design and the implementation, we can see that the security relies on both parts: a correct design and a correct implementation. This applies to primitives and protocols, but is also analogous to the security of software: When security requirements of software in a networked world grew, Eric S. Raymond extended [64] Kerckhoffs' principle in 2004:

> *Any security software design that doesn't assume the enemy possesses the source code is already untrustworthy.*

With the improvements in reverse code engineering, i.e., the technique to uncover software functionality in binaries, we can broaden Raymond's principle to:

> *Any security software design that doesn't assume the enemy is able to reverse engineer the source code is already untrustworthy.*

As stated above, most security design goals heavily rely on cryptographic protocols, which are built from cryptographic primitives. Thus, for a successful assessment of the design goals, the analyst must determine, **which** cryptographic primitives are used, **where** they are implemented in code, **how** they are concatenated, **what** their parameters are, and **when** they are used. If a software can be subjected to dynamic analysis, the primitives to achieve a security goal can be executed in a monitored manner. In this case, we deduce that the automatic detection of cryptographic primitives is beneficial to uncover a security design and to verify the correctness of the implementation.

This analysis can be applied to off-the-shelf software and also to malicious software (malware). For example, modern malware has to evade detection in order to circumvent its analysis, blocking, or removal. Therefore, cryptographic algorithms have been employed in malware, which establish encrypted communication, asymmetrically encrypt and hold local files for ransom, encrypt registry keys, or authenticate command-and-control servers.

In the recent past, there have been the following examples of cryptography in real-world malware:

- Oscar Vermaas and Daan de Graaf from the High Tech Crime Unit of the Dutch Police showed that the ShadowBot malware uses its own implementation of MD5 [79]. For obfuscation XOR is used, with a static key of 8 bit.

- Tillmann Werner and Felix Leder from the University of Bonn analyzed the Conficker malware and found usage of the OpenSSL implementation of SHA1 and a reference implementation of MD6, which was later patched in a malware update to fix a buffer overflow in the MD6 reference implementation [44]. Furthermore, the malware authors use RSA with 1024 bits for signature verification [60], in newer versions with 4096 bits.

- Werner and Leder also analyzed the Waledac malware [83]. Of 4000 functions in Waledac 1000 have been from OpenSSL. AES in CBC Mode with an IV of zero is used. The self-signed RSA client certificates are used in a key exchange protocol. Unfortunately, for the malware authors, the protocol is vulnerable to a man-in-the-middle attack and the key chosen by the server is not random but static. For obfuscation of executables embedded inside a JPEG, a XOR with the key 0xED is used.

- Andreas Greulich, of Switzerland's MELANI center, analyzed Mebroot, Torpig, and Sinowal [29]. Besides BASE64 XOR for obfuscation, the malware uses its self-designed 58-round Feistel network for symmetric encryption with an effective key size of 32 bits. Furthermore, a hashing algorithm based on the encryption algorithm and using constants from SHA1 is used.

- Wang et al. [82] analyzed a sample of the Agobot malware, which uses SSL to establish an IRC connection to a pre-specified server.

- In their technical report, Caballero et al. [12] show that the MegaD malware, which communicates with a custom protocol over TCP port 443, uses encryption to evade network-level analysis.

- Stewart analyzed the algorithms used by the Storm Worm malware [76]. The P2P and Fast-Flux communicating malware uses static XOR for subnode authentication and a RSA key with 56 bits [34].

- The Nugache Malware uses a variable-length RSA key exchange to seed 256-bit AES session keys [21]. The botnet controller's commands are passed to the MD5 hashing algorithm and the resulting hash sum is signed with a 4096-bit RSA public/private key pair. Attached to each command is also a nonce to mitigate multiple executions of the same command as it traverses the custom P2P network. This also prevents any replay or modification of commands. To gain further robustness the P2P C&C protocol establishes a network topology with minimum neuralgic points.

In reaction to these cryptovirology [89] attempts, a malware analyst has to manually identify the cryptographic algorithms and their usage to decipher the malicious actions. If this task can be automated, a faster analysis of malware would be possible, thus enabling security teams to response quickly to emerging Internet threats and protect vulnerable assets.

1.2. Thesis

Our thesis is as follows:

> *Given a program executing cryptographic primitives there exists an heuristic algorithm to identify the type of cryptographic primitives. If a standardized cryptographic primitive with its input and output is present in an execution trace, an algorithm exists to identify and verify the instance of the primitive including its parameters.*

The purpose of this work is to investigate the above thesis and attempt to discover and implement a satisfying technique to solve the problem. Due to the sheer number of ways a cryptographic primitive can be executed, it is necessary to limit our research to several assumptions and a subset of algorithms.

1.3. Assumptions and Limitations

Although a working implementation can be very well suited to analyze malware, the technique is first developed for regular software. This regular software is a selection of small applications, which each use a single cryptographic primitive, also called cryptographic algorithm or just algorithm, from an open-source implementation. For an overview of the testing applications refer to Table 6.1.

Since the modes of operation[1] do not directly modify the cipher, but rather its in- and output, we do not identify the mode because it is part of the algorithmic composition. Nevertheless, given an analyst has a set of used cipher instances with their parameters, it is possible to determine the mode of operation by comparing input, keys, and output of each instance manually. Similar to the modes of operation, we also do not identify plaintext encodings, padding, or compression, because they are dependent on the higher cryptographic composition.

In Table 6.1, we also list the XOR algorithm as a cryptographic primitive and as a testing application. Hereby we mean the encryption using bitwise exclusive disjunction. XOR is often used to obfuscate a binary or to encrypt data before it is send over the network and is not a real protection mechanism. Given a XOR-key is only used once, the algorithm is also known as an One-Time-Pad.

The system's implementation is developed on Microsoft Windows XP SP3 32bit and is thus focused on the Intel Architecture. Although an intermediate language can be well suited for heuristical identification, this is out of the scope of this work. Also, we do not analyze cryptographic code executed by code interpreters, e.g., Python or Perl. We rather focus on compiled C/C++ code, because we analyze on a fine-grained instruction-level scope, where the actual instructions would be distorted by code interpreter instructions.

We do not consider code-protection, obfuscation, and encryption, although the techniques are widely deployed among malware. The focus of this work is general software and coping with anti-analysis techniques is left to the tracing framework. Nevertheless, we evaluate our implementation against a packer in Chapter 6.

[1]The modes of operation, e.g., ECB, CBC, etc., define how multiple cipher inputs and outputs are chained together, because large files require multiple cipher operations. The mode of operation circumvents the deterministic characteristic of block ciphers: otherwise it may enable an attacker to identify identical encrypted blocks inside a file.

Furthermore, we assume that a software, which is subjected to the analysis of our system, does not utilize evasion methods specifically targeted against our identification techniques. Thus, we assume that the traced executable actually runs the cryptographic code. We also assume that the cryptographic code needs a finite amount of instructions, because we buffer instructions and the FIFO queue is bound to the amount of system memory. The default FIFO size and other computational requirements are shown in Chapter 6. We consider runtime speed of the implementation as a minor design objective.

1.4. Contributions of this Work

The thesis makes the following primary contributions:

- We summarize existing, and describe novel identification techniques for cryptographic primitives in software that help reducing the time for a software analyst to determine the underlying security design. We also point out the inherent characteristics of cryptographic code.

- We have implemented a system that allows the automatic application of our technique by utilizing a dynamic binary instrumentation framework to generate an execution trace. The system then conducts the cryptographic identification on the trace and summarizes the results of the different identification methods.

- We present an evaluation where our system was used to uncover cryptographic primitives and their usage in testing, off-the-shelf, and packed applications.

1.5. Outline

In Chapter 2, we give an overview of related work in the field of cryptographic code detection and software tracing frameworks. In Chapter 3 we describe a code analysis terminology and the utilized software analysis algorithms. The identification methods and the observation on which they are based are presented in Chapter 4 and their implementation in Chapter 5. We evaluate our implementation in Chapter 6 and draw the conclusions from the implementation and its evaluation in the last chapter, Chapter 7.

Chapter 2.

2 Related Work

There have been approaches to ease the finding of cryptographic primitives inside software binaries. They can be distinguished between static and dynamic methods. Static tools analyze the binary stream of the given software without executing it. Dynamic tools run the software and analyze the stream of executed instructions and data accesses.

2.1. Static Approaches

All static tools in Table 2.1 use signatures to determine whether a particular, compiled implementation of a cryptographic primitive is present in a software binary. A signature can match a x86 assembly code snippet, magic constants of the algorithm, s-box structures, or the string for an import of a cryptographic function call. If a signature is found, the tools print the name of the primitive, e.g., DES, and optionally the implementation, e.g., OpenSSL. Refer to Appendix B for download URLs and SHA1 sums of the tools.

Name	Author(s)	Platform	Version
Krypto Analyzer (KANAL)	Several	PEiD	2.92
Findcrypt plugin	Ilfak Guilfanov	IDA Pro	2
SnD Crypto Scanner	Loki	OllyDBG	0.5b
Crypto Searcher	x3chun	standalone	2004.05.19
Hash & Crypto Detector (HCD)	Mr Paradox, AT4RE	standalone	1.1
DRACA	Ilya O. Levin	standalone	0.5.7b

Table 2.1.: Publicly available static tools

The tools mainly differ in the amount and quality of signatures and the support platform, e.g., PEiD, OllyDBG, IDA Pro, or standalone. Some tools offer the functionality to display the compiler of the binary [30], e.g., Visual C compiler version 8, and contain signatures for compression functions, e.g., zlib, and encoding functions, e.g., base64. One tool contains a specific signature for cryptographic libraries and thereby can differentiate between the use of OpenSSL DES or Delphi's DES implementation. Since the tools statically analyze the binary stream, the binary must be first unpacked, deobfuscated and unprotected if necessary. Furthermore, if the cryptographic code is dynamically loaded, the analysis will only be successful if the memory of the process containing the loaded cryptographic code is dumped. In contrast, our approach of dynamically analyzing program execution enables us to skip deobfuscation and handle dynamically created code.

We evaluated the six publicly available tools using the set of target testing applications presented in Section 6.2. Our aim was to discover whether the actually used primitive is displayed by the analysis tool. Neither tool was able to identify the Beecrypt testing applications, because they were dynamically linked, or the XOR testing application, because no XOR signatures exist.

In Table 2.2 we summarize the performance of the tools on the x-axis and the test applications on the y-axis. A + sign denotes that the tool has found the applications's algorithm. A - sign denotes that the tool has not found the specific algorithm. The numbers in Table 2.3 denote the count of false-positives. For example if a tool detects DES, SHA1, SHA256, and SHA512 for a DES application, the notation would be 3, otherwise if a tool detects SHA1 and SHA512 for a DES application, the notation would be 2. We can make the following observations:

- Every tool is able to detect MD5 with false-positives ≤ 1.
- None of the tools is able to detect RSA or XOR.
- RC4 is only detected once.

To enumerate the amount of signatures and algorithms we applied the static tools for whole cryptographic libraries. The results in Table 2.4 show that KANAL contains a large set of signatures optimized for OpenSSL. Note that DRACA summarizes and displays several signatures for one algorithm as one single match.

This short review shows the effectiveness and limitations of signature-based static search. Since static analysis is based on the static disassembly of the binary executable, we do not know whether any, or which, of the available algorithms are actually used during runtime.

	KANAL	Findcrypt	SnD	x3chun	HCD	DRACA
Gladman AES	+	-	+	-	+	-
Cryptopp AES	+	-	+	+	+	-
OpenSSL AES	+	+	+	+	-	-
Cryptopp DES	+	+	+	+	-	+
OpenSSL DES	+	-	+	+	-	-
Cryptopp RC4	-	-	+	-	-	-
OpenSSL RC4	-	-	-	-	-	-
Cryptopp MD5	+	+	+	+	+	+
OpenSSL MD5	+	+	+	+	+	+
OpenSSL RSA	-	-	-	-	-	-
Cryptopp RSA	-	-	-	-	-	-
XOR	-	-	-	-	-	-

Table 2.2.: Detection performance among the static tools

	KANAL	Findcrypt	SnD	x3chun	HCD	DRACA
Gladman AES						
Cryptopp AES	2	2	2	1		
OpenSSL AES	6	3	1	3	6	1
Cryptopp DES	3	2	3	2	1	
OpenSSL DES						
Cryptopp RC4			3			
OpenSSL RC4						
Cryptopp MD5		1	1			1
OpenSSL MD5		1				1
OpenSSL RSA						
Cryptopp RSA	4	3		3	4	1
XOR						

Table 2.3.: False-positives among the static tools

	KANAL	Findcrypt	SnD	x3chun	HCD	DRACA
beecrypt.dll	11	18	7	5	7	4
libeay.dll	126	14	17	13	20	7

Table 2.4.: Matches of the static tools for whole cryptographic libraries

2.2. Finding Keys in Binary Data

Shamir and Van Someren [73] propose a method to efficiently locate RSA and arbitrary keys inside a bit string. Their algebraic method to find RSA private keys requires that the attacker posses the corresponding public key and a ciphertext. This scenario does not apply to our preconditions. To find arbitrary keys the authors exploit the observation that keys have a higher entropy density than patterned data. The proposed methods were refined and implemented by Janssens [35] and Janssens et al. [36]. Similar work based on searching for ASN and DER encodings has been published by Klein [40].

For forensic investigation Halderman et al. [32] propose a strategic search for relations between key scheduling stages, because the key representation changes during the algorithm in a predefined manner. Maartmann-Moe et al. [50] summarize the research in this field and provide an open-source implementation to find RSA, AES, Serpent, and Twofish keys. In [33], the authors show that only 27 % of random private key bits are sufficient to recover the private exponent. The approach goes along the lines of our method to search for parameters in memory. Although, we need to note that this approach is only possible if we search for public known algorithms, which are verifiable with a reference implementation.

A somewhat related approach by Stevens [75] targets any file, not only executable files. His open-source software can be used to find encoded shellcode in a PDF file for example. It finds XOR, ROL, and ROT keys in the file and searches whether a possible payload has been encoded using this key. A similar approach is OfficeMalScanner by Boldewin [9].

2.3. Dynamic Approaches

The first paper, to our knowledge, which addresses the problem of revealing the cryptographic algorithms in a program during runtime is by Wang et al. [82]. The authors refer to the task of automatic protocol reverse engineering [47, 11, 86] and motivate the need for automatic decryption of network messages, because the methods for protocol reversing are only suitable for plaintext messages. In order to receive the plaintext analogue for an encrypted protocol message, they use two main observations:

1. A received message will go through two processing phases: decryption and normal protocol processing.

2. The instructions used for message decryption are significantly different from

the normal protocol processing phase, as shown by Table 2.5. During personal correspondence with the authors, we learnt that the table does only contain instructions, which *modify* tainted data. Therefore the table's fields do not consider *mov* or *call* instructions.

Routine	Total Instructions	Bitwise Arithmetic Instructions	Message Bytes
DES	69112	99.72%	2K
CAST	21225	89.13%	2K
RC4	3042	89.05%	2K
AES	8475	81.32%	2K
HTTP request	3227	13.29%	107
FTP port	5898	7.14%	28
DNS response	1687	13.22%	46
RPC bind	2342	7.94%	164
JPEG	12898	8.62%	3224
BMP	956	23.95%	3126

Table 2.5.: Distinguishing between decryption and other routines by Wang et al.

The authors utilize data lifetime analysis, including data tainting, and dynamic binary instrumentation to determine the turning point between ciphertext and plaintext, i.e., message decryption and message processing phase. Then, their Valgrind-based [54] tool is able to pinpoint the memory locations that contain the decrypted message. Thus, previous methods for protocol reversing can be naturally applied to the plaintext message.

Wang et al. underline their work with an evaluation of their implementation against four standard protocols, HTTPS, IRC, MIME, and an unknown one used by the Agobot [1] malware. In their tests, they are able to decipher all encrypted messages using their implementation.

We can draw a core observation from the paper and Table 2.5:

The percentages of arithmetic and bitwise instructions on tainted data in typical implementations of decryption algorithms differ vastly to normal instructions.

As a followup paper, Caballero et al. [12], [13] refined the methods of Wang et al. [82]. For the protocol reverse engineering of the MegaD malware, the authors first tried to adapt the methods from Wang et al., but ran into the problem that the MegaD malware does not use a single turning point between decryption and message processing.

MegaD rather decrypts a block from a message, processes it, and continues with the next block.

Wang et al. use a cumulative percentage of bitwise arithmetic instructions and observe leaf routines[1] bitwiseness to determine the turning point between encrypted and decrypted data. However, Caballero et al. use a different method to identify instances of encryption routines and parameters. They still rely on the intuition that the encryption routines use a high percentage of bitwise arithmetic instructions, but remove the cumulative metric, the tainting, and the concept of leaf routines.

For each instance of a function executed by their Dispatcher tool, they compute the ratio of bitwise arithmetic instructions. If the functions is executed for at least 20 times and the ratio is higher that 55%, the function is flagged as an encryption/decryption function. For their implementation and the MegaD malware, this method reveals all relevant cryptographic routines. To identify the parameters of the routine, for example the unencrypted data before it gets encrypted, the authors evaluate the read set of the flagged function. To distinguish the plaintext from the key and other data used by the encryption function, they compare the read set to the read sets of other instances of the same function. As only the plaintext varies, the authors are able to identify the plaintext data.

Caballero et al. also cite Lutz [49] on the intuition, that cryptographic routines use a high ratio of bitwise arithmetic instructions. For his master thesis Noé Lutz developed a tool to automatically reveal encrypted messages and demonstrated its effectiveness against the Kraken malware. Lutz bases his Wine/Valgrind-based tool on three observations: first, loops are a core component of cryptographic algorithms. Second, cryptographic algorithms heavily use integer arithmetic, and third, the decryption process decreases information entropy of tainted memory. A core method of the tool is to use taint-tracking and determine whether a buffer has been decrypted by measuring its entropy.

The main problem of relying on the entropy as a measure for the decryption process, is the possibility of false-positives depending on the mode of operation. If we consider for example the cipher-block chaining mode in Figure 2.1, we can note that the input to the encryption algorithm is the latest ciphertext xored with the current plaintext. Thus, the input to the algorithm will have a similar entropy as its output, because the xor operation composing the input will incorporate pseudo random data from the latest output of the cipher. Therefore, the output's entropy propagates to the next input and a difference in entropy is not measurable.

[1] A leaf routine contains contiguous instructions belonging to the same routine. Thus, if a parent routine calls a child routine and there is no routine called in the child routine, we will have three leaf routines: the instructions of the parent routine before the call, the child routine, and the instructions of the parent routine, after the the child routine returned.

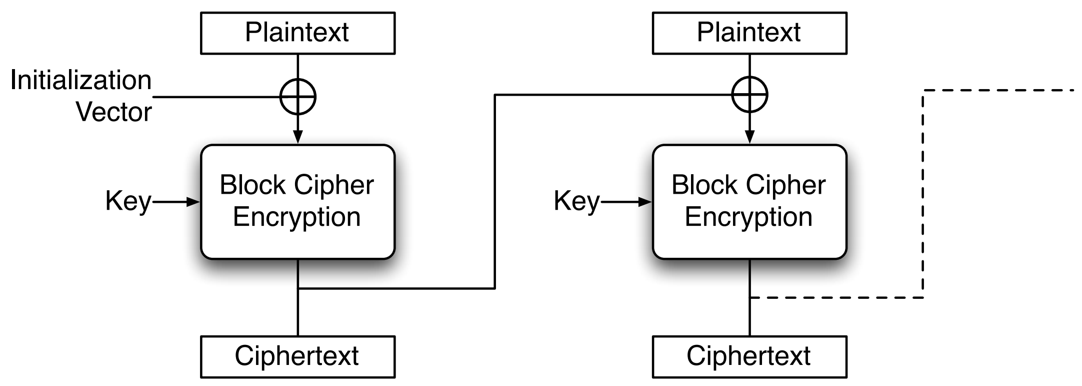


Figure 2.1.: Cipher-block chaining mode

In comparison to Lutz's work, we also research the identification of a larger set of algorithms (hash algorithms and asymmetric cryptography) and the identification and verification of input, output, and key material.

2.4. Frameworks for Execution Tracing

When discussion execution frameworks, we have to keep in mind, that the design goal of unobtrusiveness is hardly satisfiable. A system which perfectly emulates the original system is a reimplementation of the original system. Thus, choosing an execution framework can only be based on a tradeoff between usability and detectability. In the following sections we give a short overview of available dynamic execution frameworks.

2.4.1. Overview

Concerning dynamic software tracing frameworks two authors give an overview, Reynaud [65] with focus on malware and Röck [70] with focus on operating systems. Röck lists the follow dynamic operating system instrumentation frameworks: Pin [48], PinOS [10], DTrace, KProbes, SystemTap, and JIFL. Besides the listed dynamic binary instrumentation frameworks there also exists secuBT by Payer [59], Valgrind by Nethercote and Seward [54], and DynamoRIO used by Zhao et al. [90].

Reynaud distinguishes between virtualization (VMWare, Xen, Parallels, VirtualPC, etc.) and emulation (Bochs, QEMU [6], etc.). Furthermore, he addresses current difficulties in the arms-race between the malware authors and analysts. To our knowledge recent papers in this field are on Anti-Virtualization [27, 85, 24, 66], Anti-Emulation [63, 57, 58] Anti-Taint-Tracking [15], and Anti-Unpacker tricks [25].

To generate a trace of a program the framework should have the following features [78]:

- Process identification and handling
- Syscall / API call registration
- Memory read and write recording
- Instruction tracing

Further useful techniques are general taint-tracking [72, 41, 81, 51] and memory tracing [3]. The execution of adjacent code paths is a difficult challenge in dynamic analysis and is addressed by Moser et al. [52]. The difficulties of creating and maintaing a large trace are investigated by Bhansali et al. [7].

2.4.2. Virtualization and Emulation

Frameworks leveraging the virtualization extensions of AMD and Intel are the hypervisor-based approaches by Murakami [53], the MAVMM platform, based on TVMM, by Godiyal et al. [28], the unpacker Azure by Royal and Damballa [71], the Xen-based Ether by Dinaburg et al. [20], and VMware ReTrace [74]. QEMU-based tools are the K-Tracer tool for behavioral analysis by Lanzi et al. [43], the BHO Spyware analyzer by Egele et al. [22], TTAnalyze by Bayer et al. [4], Argos by Portokalidis et al. [61], and Panorama by Yin et al. [88]. An unpacker based on BitBlaze's TEMU is Renovo by Kang et al. [37]. An API-hooking-based approach is CWSandbox by Willems et al. [84].

An abstraction-layer to virtualization applications is the libvirt framework, which provides VM-independent access to Xen, KVM, QEMU, VirtualBox, VMWare and functions to modify running guest virtual machines. It is utilized for example by eKimono by Nguyen Anh Quynh [55] to scan memory for malware.

The Bochs-based frameworks, Pandoras Bochs by Böhne [8] and Zynamics' Bochs by Carrera [14], are used for unpacking. Both extend Bochs with the Python scripting language. Another Bochs-based framework, TaintBochs by Chow et al. [16], uses data tainting to analyze critical data lifetime.

2.4.3. Dynamic Binary Instrumentation

Two security research fields heavily utilize dynamic binary instrumentation (DBI): unpackers and protocol reversing. Protocol reversing has been shown in Autoformat, a Valgrind-based tool by Lin et al. [47], as well as on the iDNA-based [7] Tupni by Cui et al. [18]. Polyglot by Caballero et al. [11] also uses DBI to extract protocol message formats.

Unpackers which use DBI are, for example, Uncover by Wu et al. [87], which uses DLL injection, and MmmBop by Bania [2], implementing its own DBI engine. Furthermore, binary instrumentation is used by Dytan, a generic dynamic taint analysis framework by Clause et al. [17], and SPiKE, a malware analysis tool using unobtrusive binary-instrumentation, by Vasudevan and Yerraballi [78] based on (VAMPiRE).

2.4.4. Pin-focused Work

The Saffron unpacker by Quist and Valsmith [62] successfully uses Pin to unpack various executables. In September 2009 Daniel Reynaud used Pin on a malware corpus of approximately 60000 samples [67]. His tests showed that about 10000 errors were generated by the samples. In Guizani et al. [31] the authors report a 81.28 % success rate using Pin on binaries collected from a honeypot. In Bania [2, Section 3.2] the author states that Pin and Saffron are unable to instrument the loaders tElock and PESpin.

2.5. Summary

We can summarize that there are some research efforts which search for cryptographic primitives. Some tools utilize static analysis, others inspect the target during runtime. We can identify limitations in both categories of tools, and thus, we can conclude that further research is needed in this area.

Chapter 3. Prerequisites

In this chapter we clarify the cryptographic and software analysis terminology. We also introduce different representations of cryptographic algorithms, software tracing, and dynamic software analysis methods.

3.1. Cryptographic Algorithms

An algorithm can be examined in different representations: the formal algorithmic definition from the paper, the implemented source code, the compiled representation, and mixed cases of the later two. In the next sections, we describe the different characteristics for each representation.

3.1.1. Algorithmic Definition

Cryptographic algorithm definitions can either be known to the public or kept secret. Although, certain methods described in this work can be used to identify the existence of unknown cryptographic algorithms, we only consider the following publicly-known algorithms in this thesis and our analysis implementation.

RSA

RSA [69] is an algorithm for public-key cryptography. Based on the problem of efficiently factoring large numbers, the RSA algorithm enables the encrypting and signing of messages using public and private key pairs. The private key and the public modulus are commonly integers with a length of 1024, 2048, or 4096 bits depending on the security level.

MD5

MD5 [68] is a message digest, or hash algorithm, to generate a static length hash sum for an arbitrary length input. The hash sum has 128 bits and is computed within four rounds, each with 16 operations of either exclusive or, logical conjunction, logical disjunction, and negation.

XOR

When we refer to the XOR algorithm, we thereby mean the bitwise exclusive disjunction of the plaintext i and the key k to create the ciphertext o using the xor operation: $o = i \oplus k$. The parameters i, k, and o must be of equal, but can be of arbitrary length. Common lengths are one or multiple of eight bits. Given a XOR-key is only used once, the algorithm is also known as an One-Time-Pad, although in our evaluation we reuse key material.

DES

The Data Encryption Standard (DES) [56] is a block cipher with a fixed key size of 56 bits and a block size of 64 bits. The structure of the cipher is a Feistel network[1] with 16 rounds. The core Feistel function consists of a 32 to 48 bit expansion box, eight substitution boxes[2], and a permutation box[3]. The round key is integrated using a XOR operation.

[1] A Feistel network is a symmetric structure for the construction of block ciphers. The network separates data into two blocks L_i, R_i and applies a round function f, using a round key K_i, to one block in each round: $R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$. Then, the blocks are exchanged afterwards: $L_i = R_{i-1}$.

[2] A substitution box, or s-box, takes a number of bits and transforms them into a number of output bits.

[3] A permutation box, or p-box, shuffles input bits to create diffusion.

AES

The Advanced Encryption Standard (AES) [19] is a block cipher, originally known as Rijndael. The successor to DES supports key sizes of 128, 192, or 256 bits and uses a block size of 128 bits. The design of the cipher is based on a substitution-permutation network. The cipher operates on a 4×4 array of bytes. Depending on the key size, 10, 12, or 14 rounds are used. In the majority of rounds substitution and permutation steps are committed to the internal state. In implementations these steps are commonly performed using lookup tables, which perform a trade-off between memory space and computational time.

RC4

RC4 [38] is a stream cipher consisting of a key scheduling algorithm (KSA) and a pseudo-random generation algorithm (PRGA) to generate a stream of bits, which is commonly xored on the plaintext to generate the ciphertext. The KSA prepares an internal state array using a variable length key, typically between 64 and 256 bits. The PRGA uses and modifies the internal state to generate the stream of bits. Both algorithms, i.e., KSA and PRGA, only use bitwise arithmetic addition modulus the key size and substitution or lookup operations inside the internal state.

3.1.2. Implemented Code

The next representation of a cryptographic algorithm is its implementation. It conforms to the definition, but can be realized in many different ways. Several modifications to the internal functions of cryptographic algorithms can be performed, mostly to gain a space or time advantage. A very common form is a lookup table, which can be employed instead of bitwise addition and shifting. Another common programming technique is loop unrolling to avoid the flushing of the CPU's instruction pipeline and to save the loop's control instructions, e.g., `JMP`, `INC`.

Since the correct and efficient implementation of cryptographic algorithms is a nontrivial task, many public code libraries exist to support application developers. A majority of cryptographic code is reused from cryptographic libraries, e.g., OpenSSL, Cryptopp, or interfaces, e.g., MSCrypto API.

3.1.3. Compiled Code

The third representation of an algorithm is the compiled form of the high-level description language. Here we consider mainly C/C++ compiled code. Our testing applications are created using two different compilers, because each compiler has a different approach towards optimizing the assembly code and thus produces different results. The results also heavily depend on the used compiler settings.

Independent of the code generation is the code linkage. If the cryptographic code has been compiled as a library, and thus is dynamically linked to executable, we also need to inspect the library. Otherwise, cryptographic code can also be statically embedded into the executable and no further precautions need to be taken.

A special case is introduced by interpreted code and just-in-time generated code. In order to analyze such code, we need to mask the interpreter code sections. As such a scenario needs a specialized approach, we do not consider it here. Analogously, we do not consider code obfuscation and protection, because we also would need to normalize the code and remove the obfuscation.

3.1.4. Algorithmic Composition

Usually, cryptographic primitives are composed to more complex cryptographic systems and thus are slightly modified concerning their parameters. For example, padding, to finish a block to be enciphered, may be added to a plaintext. Depending on the mode of operation, the plaintext may never be introduced to the cryptographic algorithm, but rather xored into the ciphertext, e.g., in cipher feedback mode (CFB), as illustrated in Figure 2.1.

Since the composition varies depending on the higher protocol, we do not consider the modifications to the inputs before, or to the outputs after, the cryptographic algorithm has been invoked. Therefore, we define the plaintext as the input to the algorithm and the plaintext may already include a padding, compression, or encoding. We rather focus on the identification of atomic cryptographic primitives, because they constitute the least common denominator of cryptographic software. If we have knowledge of the used cryptographic primitives, we could reconstruct or identify the higher cryptographic protocol from their composition, as described in Section 7.2.

3.2. Execution Tracing

Under execution tracing, or generally tracing, we understand the process of analyzing a binary executable during runtime to generate a protocol, describing the instructions executed and the data accessed by the executable.

This process has several requirements and constraints. A major requirement is the correctness of the trace, meaning that the trace should contain all information to reconstruct each system state the executable encountered during its execution. An executable should not be able to escape the tracing or to render it incorrect. Secondary requirements are the efficiency and the unobtrusiveness of the tracing system. As reviewed in Section 2.4, several frameworks exist and each has different advantages and drawbacks. We use the Pin dynamic binary instrumentation (DBI) framework to conduct the trace. We choose Pin because of the available API, which enables us to create fine-grained traces. Also, the previous applications of Pin for malware analysis, referenced in Chapter 2, and the good support and ongoing development are positive criteria for the choice of the Pin dynamic binary instrumentation framework.

Instrumentation is the technique that inserts extra code into an arbitrary program to collect runtime information. Since Pin uses dynamic instrumentation, there is no need to recompile or relink the program. Pin discovers code at runtime, including dynamically-generated code, and instruments it as specified by the Pintool.

Using the Pin API, a Pintool has access to context information such as register contents or debug symbols. The Pin framework deals with the dynamic code injection, code integrity, and restoring of registers which were modified by the Pintool. Pin differentiates between two modifications to program code: instrumentation routines and analysis routines. Instrumentation routines detect where instrumentation should be inserted, e.g., before each instruction, and then modify the code accordingly. The instrumentation routines occur the first time a new instruction is executed. On the other hand, analysis routines define what actions are performed when the instrumentation is activated, e.g., writing to the trace file. They occur every time an instrumented instruction is executed. A core Pin paradigm is to filter using instrumentation routines and to keep the count of invoked analysis routines small. The instrumentation can be performed on three different granularity levels: instruction, basic block (single entry, single exit), and trace (single entry, multiple unconditional control flow changes/exits). Trace instrumentation is an instrumentation technique specified by Pin and should not be confound with our method of runtime tracing. Because we need to record on a fine-grained level, as explained in Section 3.2.2, we instrument on instruction-level.

Naturally, dynamic analysis has the general constraint that if code is not executed, it cannot be analyzed. Thus, we rely on the fact that the binary executable unconditionally executes the code we want to analyze. Otherwise, the code would not be incorporated in the trace and cannot be used in later identification methods.

3.2.1. Data Reduction

In order to minimize the size of the logfile, we utilize two filter methods. On the one hand, we filter libraries of which we have a-priori knowledge that they do not contain cryptographic code. Using a DLL whitelist, we are able to circumvent large code portions, for example of pseudo random number generation. This is especially useful to reduce the trace time and file size. On the other hand, we can filter by thread ID and are also able to start the trace after a certain number of instruction already occurred, for example to skip an unpacker.

3.2.2. Required Data

For the analysis we need to record the following information on an instruction-level granularity:

- Current Instruction pointer
- Optional debug information, e.g., DLL module, function symbol, offset to function symbol
- Current Thread ID
- Instruction disassembly
- Involved registers and their data
- Accessed memory values, before and after the instruction, including mode (read or write), size, and address

Using this information, we are able to conduct the next step: the analysis of the trace. The analysis, which is performed after or in parallel to the trace, is divided into two kinds of procedures. At first, high-level information, e.g., the control flow graph, is generated from the trace. Next, the cryptographic code identification methods are executed upon the high-level representation. The high-level reconstructing procedures are described in the following sections, the cryptographic code identification methods are described in the next chapter.

3.3. Basic Block Detection

A basic block is defined as a sequence of instructions, which always will be executed in the given order. Thus, each instruction in a basic block will always be executed before its next instruction. Each basic block has a single entry and single exit point.

Since the basic blocks are generated dynamically from a trace, the result of the basic block detection algorithm may differ from a static detection algorithm [80]. The basic blocks are generated from the dynamic trace, thus non-executed code will not be considered by the detection algorithm, because it is not incorporated in the trace. Nevertheless, an advantage of dynamic tracing is the ability to monitor indirect branches and thus we are able to incorporate their result into the basic block detection algorithm.

Algorithm 3.1 Basic block detection

```
Require: ordered list of executed instructions $I$
 1. for all instructions $i$ in $I$ do
 2.   if $i$ is a call, return, or jump then
 3.     save, that $i$ ends a basic block
 4.   else if $i$ is the next instruction after call, return, or jump then
 5.     save, that $i$ starts a basic block
 6.     save, that at the address of $i$ a basic block is started
 7.   end if
 8. end for
 9. initialize list of basic blocks $B$
10. initialize temporary basic block $b$
11. for all instructions $i$ in $I$ do
12.   if $i$ does not start a basic block and at the address of $i$ a basic block is started and
      $b$ is not empty then
13.     append $b$ to $B$
14.     empty $b$
15.   end if
16.   append $i$ to $b$
17.   if $i$ ends a basic block then
18.     append $b$ to $B$
19.     empty $b$
20.   end if
21. end for
22. return $B$
```

If a basic block is changed by self-modifying code, the change is noticed when the new code is first executed. A modified basic block is therefore registered as a new basic block, because the new block's instructions are different from the old block.

Given a trace, the basic blocks are generated as follows. A start of a basic block is denoted by the target of a jump or a call. An end of a basic block is marked by either a return, a call, or a branch (unconditional or conditional, direct or indirect branch). Therefore, a jump into a previously registered basic block divides the basic block into two blocks. Dividing jumps are detected in Algorithm 3.1 (line 12) by also saving addresses that constitute a new basic block (line 6).

Our method is shown in Algorithm 3.1. The algorithm generates the list of executed basic blocks by iterating two times over the traced instructions. In the first iteration, the algorithm saves which instruction starts or ends a basic block. Also, the algorithm saves which address starts a basic block in order to detect jumps in the middle of a basic block. To generate the final list of basic blocks, the second iteration is started. There, the algorithm uses a current basic block to temporary gather instructions. The current basic block is saved and emptied in two cases: either the current instruction ends a basic block, or at the current address a new basic block is started.

3.4. Loop Detection

Loops are defined as the repeated execution of the same instructions, commonly with different data. To perform the detection of loops, we follow the approach from Tubella and González [77]. Although we may use the dominant relationship in the flow graphs, for example the Lengauer and Tarjan [46] algorithm, it does not recover the same amount of information as [77], because it operates on a control flow graph, and therefore does not convey in which order control edges are taken during execution. However, using the Tubella and González algorithm, we are able to determine the hierarchy of loops and the exact amount of executions and iterations of each loop body.

The algorithm detects a loop by multiple executions of the same code addresses. A loop execution is completed if there is no jump back to the beginning of the loop body, a jump outside of the loop body, or a return instruction executed inside the loop body.

To track the current progress in the nesting of loops, the algorithm uses a current loop stack, denoted by L in Algorithm 3.2. For each instruction in the trace, the algorithm first checks whether the instruction is a jump. If the target address of the jump t is not inside the loop stack, the (target address t, current address i)-tuple is pushed on the loop stack and a new execution of this loop is recorded. If the target address is on the loop stack, we check whether the jump to the beginning of the loop body is actually taken. If it is taken, we record a new iteration for the loop body denoted by (t, i), clear the stack above the current loop body entry, because these loops have finished, and optionally update the loop body's end i in (t, i). If the jump is not taken and the execution at i continues outside the loop body, i.e., $i \geq b$ **for** (a, b) in L **where** $t = a$ (line 12), the loop iteration and its execution has been finished and thus it is cleared from the stack, including all loops above on the stack.

Algorithm 3.2 Loop detection

```
Require: ordered list of executed instructions $I$
1. initialize loop stack $L$
2. for all instructions $i$ and their next instruction $t$ in $I$ do
3.   if $i$ is a call, return, or jump and target $t$ is not in $L$ then
4.     record a new execution of $(t, i)$
5.     push $(t, i)$ on $L$
6.   end if
7.   if $i$ is a call, return, or jump and target $t$ is in $L$ then
8.     if the jump from $i$ to $t$ is taken then
9.       record a finished iteration for $(t, i)$
10.      pop $L$ above $(t, i)$
11.      update $i$ for $(a, b)$ in $L$ where $i > b$ and $t = a$
12.    else if $i \geq b$ for $(a, b)$ in $L$ where $t = a$ then
13.      record a finished iteration and a finished execution for $(t, i)$
14.      pop $L$ above and including $(t, i)$
15.    end if
16.  end if
17.  for all $(t', i')$ in $L$ do
18.    if $t' \leq i \leq i'$ and ($t < t'$ or $t > i'$) then
19.      record a finished iteration and a finished execution for $(t', i')$
20.      delete $(t', i')$ from $L$
21.    end if
22.    if $t' \leq i \leq i'$ and $i$ is a return then
23.      record a finished execution for $(t', i')$
24.      delete $(t', i')$ from $L$
25.    end if
26.  end for
27. end for
```

At every instruction the algorithm also checks the integrity of the loop stack in line 17-26. Therefore, it iterates over the loop stack and checks whether the current instruction address is inside a currently running loop. If it is and the current instruction is a return instruction, the particular loop's execution is finished and the loop is removed from the stack. Also, if the target of a current jump points outside the body of the current loop, then the execution of the loop is finished and the loop is removed from the stack.

For the example in Listing 3.1, we generated the following pseudo trace in Table 3.1. The loop stack illustrates the number of currently executing loops using squares. The number of iterations per loop is demonstrated using the corresponding number in the square of the loop.

The results of the loop detection algorithm for the given example would be the following:

- The outer loop is executed one time and has two iterations. The loop body begins with `c=c+1` and ends with `i<2?`.
- From the outer loop, the inner loop is executed two times: first with one, then with two iterations. The inner loop begins with `j%5<3?` and ends with `j<i?`.
- During each inner loop iteration a call to `dosomething()` is issued and therefore the call is recorded as an own loop execution, with one iteration each.

If we search for the loops of the XOR testing application from Section 6.2, illustrated in Figure 3.1, the loop detection algorithm generates the results in Table 3.2. We can note that both inner and outer loops of the algorithm are detected. The first row in the table shows the inner loop, which is executed 32 times with 128 iterations each. The second row illustrates the outer loop, which is executed once and launches the inner loop 32 times.

Loop detection, with the fine granularity presented here, is a clear advantage of dynamic analysis. Otherwise, with static analysis, no information would be available concerning how many iterations or executions a particular loop is executed. The identification methods using the loop detection are discussed in detail in the next chapter.

A common optimization technique for cryptographic code is the unrolling of loops to save the instructions needed for the loop control, e.g., counters, compare, and jump

```
for(i = 1; i < 2; i++)
{
  c = c + i;
  for(j = 0; j < i; j++)
  {
    if(j % 5 < 3)
    {
      dosomething(c,j);
    }
  }
}
```

Listing 3.1: Sample loop code

Operation	Loop Stack	Comment
i=1	$\boxed{1}$	
c=c+1	$\boxed{1}$	outer loop started
j=0	$\boxed{1}$	
j%5<3?	$\boxed{1}\boxed{1}$	inner loop started
dosomething()	$\boxed{1}\boxed{1}\boxed{1}$	temporary loop for dosomething()
j++	$\boxed{1}\boxed{1}$	dosomething() returned
j<i?	$\boxed{1}\boxed{1}$	
i++	$\boxed{1}$	inner loop finished
i<2?	$\boxed{1}$	
c=c+i	$\boxed{2}$	second iteration for outer loop
j=0	$\boxed{2}$	
j%5<3?	$\boxed{2}\boxed{1}$	
dosomething()	$\boxed{2}\boxed{1}\boxed{1}$	
j++	$\boxed{2}\boxed{1}$	
j<i?	$\boxed{2}\boxed{1}$	
j%5<3?	$\boxed{2}\boxed{2}$	second iteration for inner loop
dosomething()	$\boxed{2}\boxed{2}\boxed{1}$	
j++	$\boxed{2}\boxed{2}$	
j<i?	$\boxed{2}\boxed{2}$	
i++	$\boxed{2}$	inner loop finished with two iterations
i<2?	$\boxed{2}$	outer loop finished
		finished example

Table 3.1.: Pseudo trace for example in Listing 3.1

instructions, and to mitigate the risk of clearing the instruction pipeline by a falsely-predicted jump. While many implementations discussed here partially unroll loops, no implementation unrolls every loop. Therefore, we find a lot of looped cryptographic code and can still rely on this analysis observation elaborated in the next chapter.

Address			Iterations per Execution			Total
Start	End	Executions	Minimum	Average	Maximum	Iterations
0x401130	0x401145	32	128	128	128	4096
0x401123	0x401169	1	32	32	32	32

Table 3.2.: Detected loops for the XOR test application

3.5. Control Flow Graph Generation

We build our control flow graph generation algorithm upon the the basic block detection algorithm. Given the list of executed basic blocks, we need to find the control flow changes, i.e., which basic block jumps to which block. In order to generate the control flow graph [26] from the list of basic blocks we use a key-value data structure (dictionary). The algorithm in Listing 3.3 iterates over the executed basic blocks. It generates an unique set of basic blocks using the unique keys of the dictionary. For each basic block the next executed basic block is recorded to denote the directed edge between the two blocks. The final dictionary will hold a unique set of basic blocks, representing the vertices, as keys. The dictionary's values will hold the directed edges going away from the respective vertex in an array.

Algorithm 3.3 Control flow graph generation

```
Require: ordered list of executed basic blocks $B$
 1. initialize (key, value) dictionary $C$
 2. $last \leftarrow 0$
 3. for all basic blocks $b$ in $B$ do
 4.   insert $b$ as key into $C$
 5.   if $last \neq 0$ then
 6.     append $b$ to the value of key $last$ in $C$
 7.   end if
 8.   $last \leftarrow b$
 9. end for
10. return $C$
```

If we consider the generated control flow graph for the XOR test application, depicted in Figure 3.1, we can clearly identify the loops shown in Table 3.2. The core XOR operation takes place in the third basic block from the top (`BBL 0x401130`) during its second instruction.

For visual verification and usage in this thesis we generated PDF files from the control flow graphs. Therefore, we transformed the algorithm's output into the Graphviz dot description language [23] and converted it to a PDF file using the dot tools.

3.6. Memory Reconstruction

To further analyze the data incorporated in a trace, we need to reassemble the memory contents, i.e., generate memory dumps from the trace at different points in time. This is especially important because cryptographic keys are larger, e.g., 128 or 256 bit, than

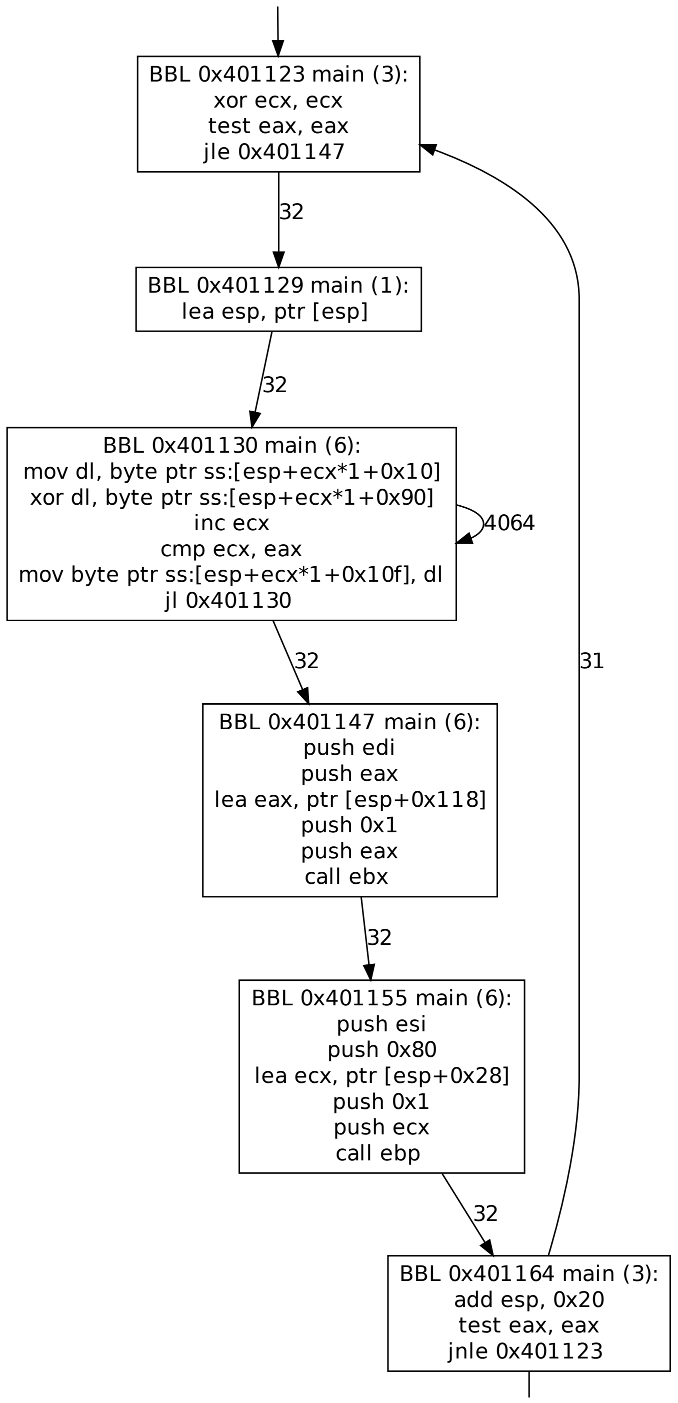


Figure 3.1.: Control flow graph generated for the XOR application

the word size of the architecture, e.g., 32 bit. Thus, a cryptographic primitive can extend over several words in memory and has to be accessed by multiple operations. To reconstruct such a primitive, we need to consider and combine multiple operations. As we do not conduct taint tracking [42, 5], we need to reassemble the memory based on its addresses.

If an instruction involves a memory access, we record the following information to the trace:

- Memory Address
- Size of access, e.g., 8 bit
- Actual data read or written
- Mode of operation, i.e., read or write

From this information, which is attached to several instructions in the trace, we need to reconstruct the memory content. Since data at an address can change during the trace, we may have several values for the same address. Thus, instead of dumping the memory for a particular point in time, we instead reconstruct blocks of memory that have a semantic relationship. For example, a read of 128 bit cryptographic key material may occur in four 32 bit reads. Then, later a 8 bit write operations to the same memory region may destroy the key in a memory reconstruction. Therefore, we try to separate the 8 bit writes from the 128 bit key block.

For this method, we rely on a few characteristics of the memory block, i.e., the interconnected memory composed of several words. At first, we distinguish between read or write blocks and thus separate the traced memory accesses based on their mode. Next, we assume that a block is accessed in a sequential order. Thus, we save the last n memory accesses, which occurred before the current memory access. In our experiments $n = 6$ showed to be a reliable threshold. As a third characteristic, we use the size of the access to distinguish between multiple accesses at the same address.

Algorithm 3.4 Memory reconstruction

```
Require: address-ordered list of memory accesses $M$
1. for all memory accesses $M'$ at address $a$ in $M$ do
2.   for all memory access $m$ in $M'$ do
3.     if memory access $m$ is not tagged processed then
4.       launch recursive memory block search with $(M, m, a)$
5.     end if
6.   end for
7. end for
```

The algorithm is shown in Listing 3.4. It calls the recursive function in Listing 3.5, which is applied to every value in memory and traverses the memory. The recursive function first generates a set of the memory accesses to the current address. Also, a precomputed set is prepared that holds all memory accesses which appeared nearby the current memory access in the trace. Then, in line 9 of Listing 3.5 the next address is computed using $n = a + \text{bytesize}(v)$. If the next address contains a memory access, which has been used nearby the current access, the block search continues in line 18-19. Also, if the next address contains the same structured memory values, e.g., a 8 bit read and two 32 bit reads, the search continues in line 21-23. A special case is handled in line 12: if multiple nearby memory accesses are found the recursive search is split and continued for each of the multiple accesses.

Algorithm 3.5 Recursive memory block search

```
Require: address-ordered list of memory accesses $M$, current memory access $m$ and
    the address $a$
1.  $M'$ = set of memory accesses to $a$ in $M$
2.  $A = \pm 6$ nearby memory accesses from the trace, where the access to $a$ occurred
3.  if memory access $m$ is not tagged processed then
4.    append data value $v$ of $m$ to current block
5.  else
6.    return current block
7.  end if
8.  tag current memory access $m$ as processed
9.  next address $n = a + \text{bytesize}(v)$
10. $N$ = the set of memory accesses at address $n$ in $M$
11. if $N$ is not empty then
12.   if $|N \cap A| > 1$ then
13.     for all memory accesses $m'$ in $N \cap A$ do
14.       duplicate current block
15.       launch recursive memory block search with $(M, m', n)$
16.     end for
17.   else if $|N \cap A| = 1$ then
18.     $m' = N \cap A$
19.     launch recursive memory block search with $(M, m', n)$
20.   else if $|N \cap A| = 0$ and the structure of set $N$ is the same as $M'$ then
21.     $o$ = offset($m$) in $M'$
22.     $m'$ = value in set $N$ with offset $o$
23.     launch recursive memory block search with $(M, m', n)$
24.   end if
25. end if
26. return current block
```

If we search for a 128 bit key for example, and we traced the memory layout shown in Table 3.3, then the algorithm starts at address `001a0000` and concatenates `31313131` to

32323232 because of a sequential layout in the memory. At address 001a0010 the algorithm encounters two different values during the trace. Then, the algorithm relates 35353535 to the current block due the similar memory layout (32 bit read) and also due to the nearby usage of 34343434 in the trace.

Address	Values read		
001a0000	31313131		
001a0004	32323232		
001a0008	33333333		
001a000c	34343434		
001a0010	35353535	01	
001a0011	02		
001a0012	03		
001a0013	04	05	60
001a0014	36363636	05	08
001a0015	06	0d	
001a0016	07	15	

Table 3.3.: Shows an example for the memory block search algorithm

When we acquired the blocks and search for a 128 bit key for example, we first discard blocks without sufficient size. Larger blocks are then divided by a sliding window algorithm to contain exactly 128 bit. For the example Figure 3.2, the block of 192 bits is divided into three blocks of 128 bit each.

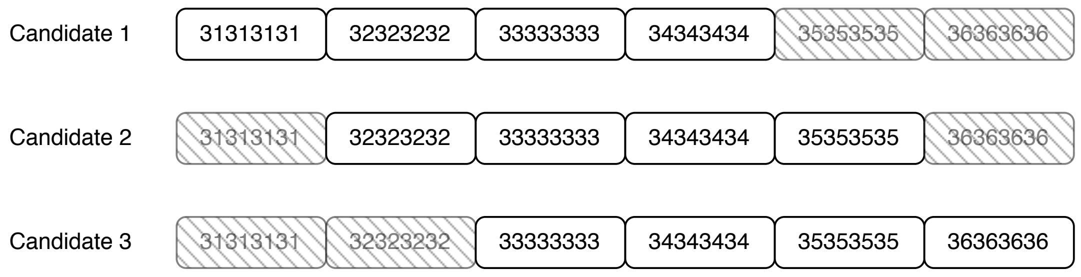


Figure 3.2.: Sliding window over candidate keys

3.7. Summary

In this chapter we presented several methods for reconstructing high-level structures from a trace. The structures are used in the following chapter to build the identification methods. We also clarified software analysis terminology and methods. For completeness, we described the cryptographic algorithms and their different representations.

4 Chapter 4. Observations and Methods

In this chapter we discuss the different properties of cryptographic code and elaborate on the implemented methods to detect the cryptographic code and its primitives. First we give an overview of the identification methodology and then, based on code observations we make, we explain the developed identification methods. In order to successfully identify the cryptographic primitives we have to algorithmically solve the following questions: **which** cryptographic primitives are used, **where** they are implemented in code, **what** their parameters are, and **when** they are used.

4.1. Reducing of Code Search Space

In order to reduce the code search space we heavily rely on the algorithms described in Chapter 3. Using the generated high-level representation of the trace, we are able to generically build the identification methods upon the representation. For example, if we search for specific code properties, we do not need to analyze the complete trace, possibly with duplicate basic blocks, but rather we are able to only search the unique set of basic blocks.

Furthermore, if a-priori knowledge of the analysis target is available, we can filter out certain code blocks. We can filter based on the name of the module, in which a particular instruction resides, to exclude system libraries. Also, we can only inspect a specific thread of the process. In order to trace certain malware samples, we could skip a number of instruction before we begin the trace, for example to circumvent code deobfuscation.

4.2. Type of Identification

We distinguish between two classes of identification algorithms: signature-based and generic. The main differentiation is the knowledge needed for the identification algorithm. For signature-based identification, we need a-priori knowledge about the specific cryptographic algorithm or implementation. On the other hand, for generic identification we use characteristics common to all cryptographic algorithms and therefore do not need specific knowledge.

4.2.1. Signature-based Identification

A signature-based identification algorithm uses a-priori knowledge of the cryptographic code in order to determine its application. For example using patterns, we are able to identify implementations of cryptographic code based on the sequence of mnemonics. A drawback of signature-based identification algorithms is that the signature or the signature matching algorithm may be evaded by modifications to the cryptographic code. The modifications could be either intentional, e.g., code obfuscation, or unintentional, e.g., compiler optimizations. An advantage of signature-based identification is the fast generation of new signatures and the simplicity of the matching algorithms.

4.2.2. Generic Identification

A generic identification algorithm exploits characteristics of the cryptographic code, for example the number of certain instructions. It counts and weights their existence in the given trace and compares the result to a threshold. The threshold is commonly determined by empiric means. If the result outweighs the threshold, the algorithm has determined the existence of the cryptographic code. Using the results of a threshold decision, we can also drop false-positives by an excluding characteristic. For example, if a threshold indicates a cryptographic algorithm in a slice of the trace, then we can check whether the data referenced in that slice contains cryptographic parameters and verify them with a reference implementation.

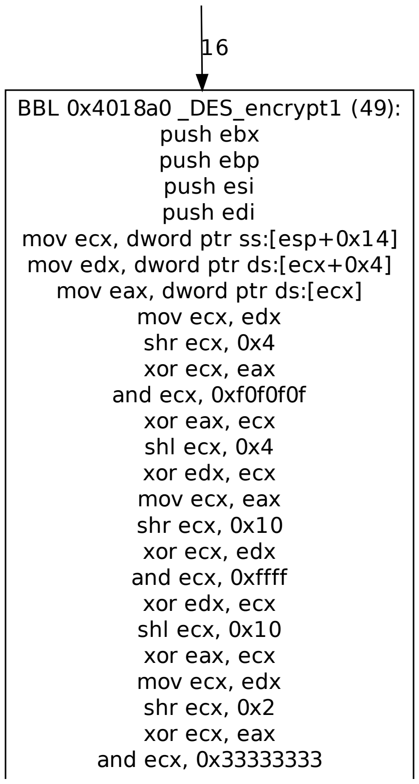
In general, generic identification algorithms have a higher probability of false-positives compared to signature algorithms. Nevertheless, they offer the possibility of detecting cryptographic code without a-priori knowledge about the code, but rather with a-priori knowledge about the characteristics of the code.

4.3. Observations

In this section, we point out four important features of cryptographic code, which we found and confirmed during the course of this work. Some observations were already mentioned in Chapter 2.

Observation 1
Cryptographic code makes excessive use of bitwise arithmetic instructions.

Due to the computations inherent in cryptographic algorithms many arithmetic instructions occur. Especially for substitutions and permutations, the compiled implementations make extensive use of bitwise arithmetic instructions. Also, many cryptographic algorithms are optimized for modern computing architectures: for example, contemporary algorithms like AES are speed-optimized for the Intel 32 bit architecture and use the available bitwise instructions.



```
BBL 0x4018a0 _DES_encrypt1 (49):
push ebx
push ebp
push esi
push edi
mov ecx, dword ptr ss:[esp+0x14]
mov edx, dword ptr ds:[ecx+0x4]
mov eax, dword ptr ds:[ecx]
mov ecx, edx
shr ecx, 0x4
xor ecx, eax
and ecx, 0xf0f0f0f
xor eax, ecx
shl ecx, 0x4
xor edx, ecx
mov ecx, eax
shr ecx, 0x10
xor ecx, edx
and ecx, 0xffff
xor edx, ecx
shl ecx, 0x10
xor eax, ecx
mov ecx, edx
shr ecx, 0x2
xor ecx, eax
and ecx, 0x33333333
```

Figure 4.1.: OpenSSL implementation of DES

In Figure 4.1 we show Observation 1 for a part of the OpenSSL DES implementation and in Figure 4.2 we show this observation for three different MD5 implementations.

We can recognize 16 bitwise of 25 total instructions for the DES example. This reproduces the results from Wang et al. [82] and the percentages in Table 2.5. Spot checks by us and Wang et al. show that non-cryptographic code has a lower bitwise arithmetic percentage than cryptographic code. For example, the BMP image processing routine has a percentage of 23.95%. Therefore, we can conclude that the opposite of the observation, i.e., non-cryptographic code contains a low percentage of bitwise arithmetic instructions, is verified empirically.

```
rol ecx, 0x7
add ecx, edi
mov ebp, esi
xor ebp, edi
and ebp, ecx
xor ebp, esi
add ebp, ebx
lea edx, ptr [edx+ebp*1-0x173848aa]
mov ebx, dword ptr ds:[eax+0x18]
rol edx, 0xc
add edx, ecx
mov ebp, edi
xor ebp, ecx
and ebp, edx
xor ebp, edi
add ebp, ebx
```

```
rol ecx, 0x7
add ecx, edi
mov ebp, esi
xor ebp, edi
and ebp, ecx
xor ebp, esi
mov dword ptr ss:[esp+0x54], ebx
mov ebx, dword ptr ds:[eax+0x4]
add ebp, ebx
lea edx, ptr [edx+ebp*1-0x173848aa]
rol edx, 0xc
add edx, ecx
mov ebp, edi
xor ebp, ecx
and ebp, edx
xor ebp, edi
```

```
rol edx, 0x7
add edx, ebp
mov edi, ebx
xor edi, ebp
and edi, edx
xor edi, ebx
add edi, esi
mov esi, dword ptr ds:[ecx+0xc]
lea esi, ptr [edi+esi*1-0x173848aa]
mov edi, ebp
xor edi, edx
rol esi, 0xc
add esi, edx
and edi, esi
xor edi, ebp
add edi, dword ptr ds:[eax-0x30]
```

Figure 4.2.: Beecrypt, Cryptopp, and OpenSSL implementations of MD5

Observation 2
Constants and sequences of mnemonics indicate the type of cryptographic algorithm.

As shown by Figures 4.1 and 4.2 the implementation and the algorithm has a characteristic set of constants and mnemonics. The OpenSSL DES figure has very distinctive constants, e.g., `0x33333333` and `0xf0f0f0f0`, which are also found in the Cryptopp implementations. If we compare the sequence of mnemonics in the MD5 implements, we clearly see an analogy for the first six instructions: `ROL`, `ADD`, `MOV`, `XOR`, `AND`, `XOR`. Also, the sequence of mnemonics in OpenSSL DES can be partially found in Cryptopp DES. In order to verify that the above sequence is inherent to MD5, we searched for it in all testing applications, including cryptographic and system libraries. The only occurrences of the sequence can be found in the traces of the MD5 testing applications. Although this finding does not verify the non-existence of false-positives, but it shows that mnemonic sequence signatures may constitute a sound identification method.

Observation 3
Cryptographic code contains loops.

While substitutions and permutations modify the internal data representation, they are applied multiple times commonly with modifications to the data, e.g., the round key. We can recognize, even in the unrolled code of Figure 4.1, that the basic blocks of cryptographic code are executed multiple times. Another example is the XOR testing application in Figure 3.1 which shows the loops around the `XOR` operation.

Solely for an identification method the Observation 3 is insufficient. The observation rather has to be combined with other methods to provide a sound identification, because loops are inherent in all modern software. Although the number of encryption rounds is unique to each algorithm and may be used for an identification, this is not the case for unrolled algorithms, where the original number of rounds cannot be found in the majority of unrolled testing applications which we investigated.

Observation 4
Input and output to cryptographic code have a predefined, verifiable relation.

The cryptographic algorithms which we consider in this thesis are deterministic. Therefore, for any input the corresponding output will be constant over multiple executions. Given a cryptographic primitive was executed during the trace, the input and output parameters contained in the trace will conform to the deterministic relation of the cryptographic algorithm. Thus, if we can extract possible input and output candidates for a cryptographic algorithm, we can verify whether a reference algorithm generates the same output for the given input. Thereby, we cannot only verify which cryptographic algorithm has been traced, but we can also determine what cryptographic parameters have been used. Of course, this observation can only utilized with a reference implementation: if the software program contains a proprietary algorithm, we cannot verify it.

4.4. Identification Methods

Based on our observations detailed before, we developed and implemented several identification methods. The effectiveness of the methods is evaluated in Chapter 6.

4.4.1. Signature-based

We implemented several signature-based identification methods. One is based on the sequence of instructions. The sequence of instructions is defined as the ordered concatenation of all mnemonics in a basic block. For identification, an unknown sample's sequence is created and compared to the set of existing sequences in the pattern database. If the sequence can be found, a cryptographic implementation has been detected.

We prepared the pattern database with different open-source cryptographic implementations. To differentiate between sequences defining an algorithm and sequences defining an implementation, we generated multiple datasets for each algorithm and each implementation. Thereby, we can identify implementations and algorithms in different levels of granularity and compare the effectiveness of the different patterns.

We created the following sequence patterns:

- sequences for an implementation, a set defined by a single implementation of a single algorithm
- unique sequences for an implementation, as above, but subtracted with all other implementation sets
- sequences for an algorithm, the union of all implementation sets composing the specific algorithm
- unique sequences for an algorithm, as above, but with all other algorithm sets subtracted

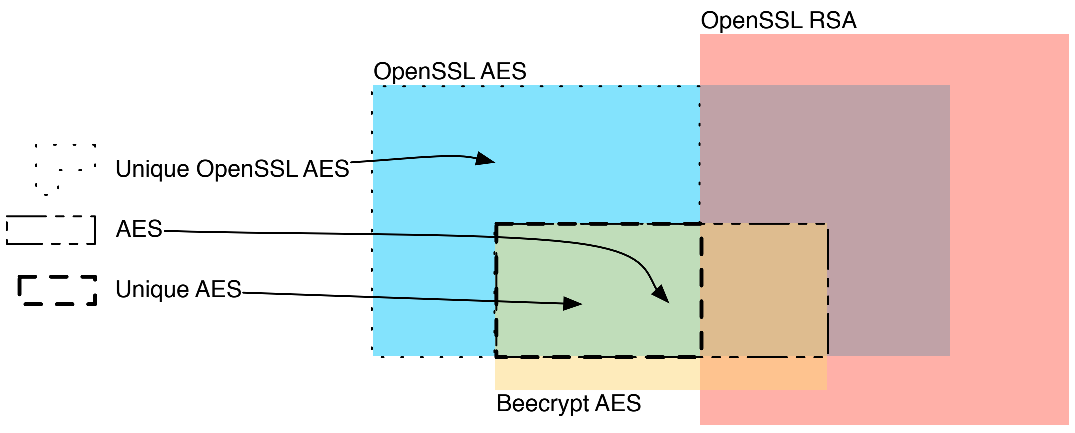


Figure 4.3.: Composition of sequence sets

An example for the composition of the different sequences is given in Figure 4.3. Each set consists of multiple sequences, defining the algorithm or implementation. For the example in Figure 4.3, we generate two distinctive sets for each algorithm and implementation. At first, we generate the set describing the implementation of an algorithm, .e.g., the blue OpenSSL AES in Figure 4.3. Then we subtract all other implementations from this set and save the result as the unique set for the specific implementations, e.g., the blue, dotted unique OpenSSL AES set. A third set is the algorithm set, e.g., AES, which is composed by the intersection of all implementations of the specific algorithm. If we subtract all other algorithm sets from this set, we get the unique set, e.g., unique AES. When we inspect a software sample for the signatures, we first derive all mnemonic sequences for the given trace. Then, we compare the given sequences with the signature sequences using intersection. The resulting match value is computed by the division of the length of the signature and the length of the intersection. Thereby, we get a predication to what percentage the signature has been found in the trace.

A second, signature-based, identification method searches for constants in memory. As noted in Section 3.1, multiple implementations of cryptographic algorithms use lookup tables to increase the speed of substitution boxes in algorithms. The lookup tables reside in memory and can be identified by their definite value. An example for such a memory value is `0xc66363a5`, which is the first value of the unrolled lookup table for AES implementations. A quick code search[1] determined that this value is used in up to 500 software projects[2]. The constant can also be found in Beecrypt, Cryptopp, and OpenSSL. However, it is not included in Vincent Rijmen's reference implementation and Brian Gladman's optimized implementation of AES, because both implementations use different lookup tables.

We manually determined about 4000 constants for MD5, AES, and DES. To collect the constants, we used the scripting interface of our implementation's interactive console, as described in Section 5.3. Based on these signatures we search the trace for memory containing such constants. If the number of found constants surpasses an empirically-determined threshold of 70%, we can record the existence of the cryptographic code.

A third identification method is based on the combination of instructions and constants. As shown in Figure 4.2, every MD5 implementation contains a `ROL 0x7` and a `ROL 0xC` instruction. Based on this observation we developed a third identification method, which employs a dataset based on (mnemonic, constant)-tuples. For every implementation we generate a set of bitwise instructions and their corresponding constants, e.g., `ROL 0x7`.

[1]See under `http://www.google.com/codesearch?as_q=0xc66363a5`.

[2]For example, Cryptopp, OpenSSL, Filezilla, WPA Supplicant, QEmu, GNUTLS, Plan 9, MySQL, Berkeley-DB, Tor, SVN, Beecrypt, Mozilla libnss.

Then, as in the first signature-based method, we form different datasets using union, intersection, and subtraction. We create the datasets:

- For each implementation of an algorithm
- For each algorithm, based on the intersection of all implementations of the particular algorithm
- An unique dataset for each algorithm, based on the subtraction with other algorithms

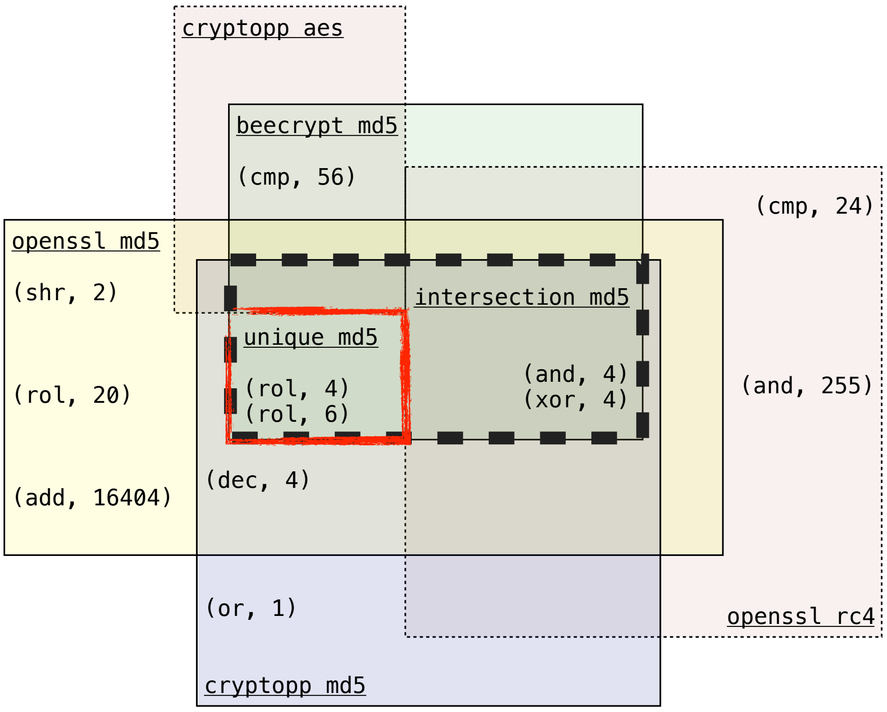


Figure 4.4.: Composition of (mnemonic, constant)-tuple datasets

An example for the datasets is given in Figure 4.4. For a given set of (mnemonic, constant)-tuples from a trace, we can therefore measure to which percentage the tuples from a signature dataset are included in the trace. We observed that the unique and intersection datasets have a stronger relation to the algorithm. Implementation datasets have a looser connection to the traced implementation and pose a higher risk of generating false-positives.

The number of tuples per testing application varies between 40 and 454 and the mean value is 165 tuples. Without detailing the results described in Chapter 6, we can note the following characteristics of the generated signature database:

- AES has no unique set, because other implementations also use its (mnemonic, constant)-tuples.

- The RC4 unique set only has two entries and therefore does not describe RC4 very well. Accordingly, we cannot use this dataset because it generates false-positives for RC4 with a higher probability.

- RSA, MD5, and DES have very well defined unique sets, and also the intersection and implementation sets show a good performance in Chapter 6.

We could further modify the signatures, based on our findings in Chapter 6, to eliminate the risk of generating false-positives, e.g., with a set of only two elements. We may also weight the unique datasets to represent their expressiveness, compared to the intersection and implementation datasets.

The comparison is implemented as noted in Algorithm 4.1. At first we generate the set of (mnemonic, constant)-tuples found in the trace. Using this trace-set we check for each of the a-priori-known pattern datasets to which percentage the trace-set intersects the signature-set. If the percentage is above the threshold of 70%, we report a positive identification. The threshold was empirically determined during the development process by testing.

Algorithm 4.1 Comparison of (mnemonic, constant)-tuple signatures

```
Require: ordered list of executed instructions $I$
1. $T \leftarrow$ empty array
2. for all instructions $i$ in $I$ do
3.   if $i$ is a bitwise arithmetic instruction and $i$ contains a constant in its disassembly then
4.     $m \leftarrow$ mnemonic of $i$
5.     $c \leftarrow$ constant in the disassembly of $i$
6.     $T \leftarrow$ append $(m, c)$
7.   end if
8. end for
9. for all tuples-signatures $S$ in (unique, intersect, implementations) do
10.   $I \leftarrow S \cap T$
11.   $p \leftarrow \frac{|I|}{|S|}$
12.   print match for signature $S$ is $p$ %
13. end for
```

A fourth signature-based identification method is the use of debug symbols to determine a cryptographic function. We manually created a small database of common cryptographic functions. Two sample entries of 19 total entries are shown in Listing 4.1.

```
# function symbol   : description
'FGIntRSA'          : 'Delphi RSA (FGIntRSA)',
'CryptDecrypt'      : 'MS Crypto API Cipher (CryptDecrypt RC2,RC4)',
```

Listing 4.1: API signatures

Using the debug strings incorporated in the trace, we match whether an entry in the signature database is included in the function symbol of executed code. Although this simple symbol based approach is ineffective against stripped-symbol software and malware, common software uses debug symbols extensively.

All signature-based methods have in common that they do not detect cryptographic parameters. Many perform well for the detection of the cryptographic algorithm and sometimes also detect the type of implementation. To detect cryptographic parameters we developed a further method which is described in the next subsection. For the performance of all identification methods please refer to Chapter 6.

4.4.2. Generic Bitwise Arithmetic-based

A simple identification method is built upon Observation 1 and Caballero et al. [12]. We evaluate basic blocks and determine whether the percentage of bitwise instructions is above a certain threshold. If the percentage is above the empirically-determined threshold of 55%, then we have identified cryptographic code. The calculation of the bitwise percentage skips operations which do not directly modify data, i.e., move instructions like `LEA` and `MOV`. We determine the type of instructions using an x86 instruction reference [45]. We summarize the categories logical, shftrot, bit, binary, arith in [45] as bitwise arithmetic instructions. Thus, these instructions include, but are not limited to, `AND`, `DEC`, `IMUL`, `ROL`, `SHL`, `XOR`, and also `PADDQ` (SIMD), or `FMUL` (FPU).

To eradicate false-positives, we use a minimum instructions per basic block threshold[3] of 20. Therefore, common, small basic blocks with three bitwise instructions out of five do not yield a false-positive result.

Following the work from Wang et al. [82], we also implemented a cumulative measurement of the bitwise arithmetic instructions. Instead of measuring the bitwise percentage for basic blocks or function names, we update the percentage of bitwise instructions as we traverse the trace.

[3]The threshold of a minimum basic block instructions of 20 is determined by Caballero et al. and proved to be successful in our experiments.

For better understanding we describe the approach from Wang et al. here. The algorithm's goal is to find the transition function, i.e., the point where the plaintext is encrypted to ciphertext and vice versa. Therefore, the algorithm follows the steps shown in Algorithm 4.2.

Algorithm 4.2 Wang et al. cumulative bitwise percentage method

```
Require: ordered list of executed instructions $I$
1.  $b \leftarrow 0$ // Step 1
2.  $t \leftarrow 0$
3.  $C \leftarrow$ empty array
4.  for all instructions $i$ in $I$ do
5.    if $i$ is a bitwise arithmetic instruction then
6.      $b \leftarrow b + 1$
7.    end if
8.    if $i$ is not a mov instruction then
9.      $t \leftarrow t + 1$
10.     save the current cumulative bitwise arithmetic percentage $\frac{b}{t}$ to $C$
11.   end if
12. end for
13. $x \leftarrow \max(C)$ // Step 2
14. $y \leftarrow \min(C \text{starting from} x)$
15. $l \leftarrow$ length of longest leaf routine // Step 3
16. $X \leftarrow$ function symbol at $x$
17. $Y \leftarrow$ function symbol at $y$
18. for $i = 1$ to $l$ do
19.   if $x - i$ has a different function symbol than $X$ then
20.     break
21.   end if
22. end for
23. $x \leftarrow x - i + 1$
24. for $i = 1$ to $l$ do
25.   if $y + i$ has a different function symbol than $Y$ then
26.     break
27.   end if
28. end for
29. $y \leftarrow y + i - 1$
30. for all leaf routines $l$ in the interval $[x, y]$ of the trace $I$ do
31.   if the bitwise percentage of $l > 50\%$ then
32.     return the transition point is $l$ // Step 4
33.   end if
34. end for
```

At first the cumulative percentage is calculated for every instruction in the trace. In a second step we determine the minimum and maximum percentage, because Wang et al. suppose that the decryption process occurs in between them. Although, this assumption is insufficient for larger protocols, as shown by Caballero et al., because larger protocols contain several different cryptographic algorithms and execute them in an arbitrary order. In these cases, the cumulative percentage over the complete trace may identify the first algorithm, but miss later executed cryptographic code.

Next, we determine the longest leaf routine[4]. Beginning from the maximum and minimum percentage, we traverse the trace backwards and forwards for at least the length of the longest leaf routine. Thereby we adjust the x and y indices to the beginning and end of a leaf routine. Thus, we extend the interval created in step two to the beginning and end of the respective leaf routines.

At last we iterate over the trace interval which is defined by $[x, y]$. If the bitwise arithmetic percentage of the leafs in the interval is above the threshold of 50%[5], we record the leaf routine as the transition point.

4.4.3. Generic Loop-based

We instrumented our loop detection routine to conduct further checks based on the loop information deduced from a trace file. On an experimental basis we partially reproduced the approach from Lutz [49], who tries to detect cryptographic code by measuring entropy changes of data in loops. Since we did not implement a taint-tracking algorithm, the results of the entropy-based approach are not evaluated. Furthermore, we see the problem that input/output-entropy is constant in certain modes of operation, as described in Section 2.3. Nevertheless, we encountered similar results as described by Lutz [49].

Our loop-based identification methods are based on a technique which we call loop differ. The loop differ differentiates between data values in between loop executions, iterations, and bodies. A schematic view is presented in Figure 4.5. Each cube in the three-dimensional space represents a memory access and is placed according to its position in the loop body (green in Figure 4.5), iteration (blue), and execution axis (yellow).

The loop differ utilizes the cubic structure and processes through the executions, iterations, and loop bodies. For each memory access, the loop differ determines the axis for the current execution (yellow), iteration (blue) and body (green).

[4]A leaf routine is defined as the uninterrupted sequence of instructions with the same function symbol. If a function calls an inner function, it will therefore be broken in two leaf routines, the one before the call and the one after the called function returned. Also, the callee will be a leaf routine in between the caller.

[5]Empirically determined threshold by Wang et al.

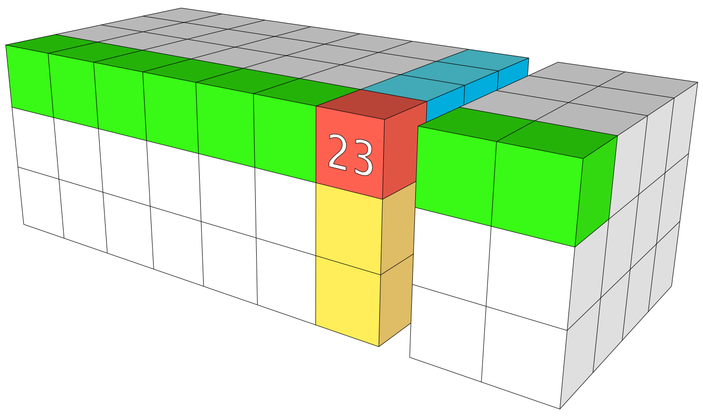


Figure 4.5.: Three-dimensional view of the loop differ

Using this method we are able to compare the values of a specific instruction, e.g., `INC` increment, over time, in each axis. We execute different analysis methods on the ordered set of data available for each axis:

1. We search for counters by observing decrements or increments in an axis. If the axis contains at least five data values and at least 90% of the deltas between each value are one, we test whether the subtraction of the first and last value of the axis is equals the length of axis. If it is, then we detected a counter value. If the first value in the axis is greater than the last value, it is a downward, otherwise an upward counter. Although a deterministic identification of a counter may be possible, we use a threshold of 90% to mask a counter, which may be later, in the remaining 10%, used for another non-counter calculation.

2. On an experimental basis, we search for relations in algorithms. We implemented a search for the permutation box in DES, which takes a 32 bit input and outputs a 32 bit permuted value.

   Given an axis, we check for all combinations in this axis whether the hamming weight percentage is in between 20% and 80%. The hamming weight of an integer is defined as the number of non-zero symbols in the (bit) integer. The hamming weight percentage for an integer i is thus the division of the hamming weight by the bit length $\log_2(i)$. If the percentage is in the given range, we can be certain that the permutation will have a significant, measurable effect. Otherwise, if $axis_{input}$ would contain only zeros in the bit representation, the p_{box} permutation result would also always be zero, and thus yield false-positives with a higher probability.

We evaluate the DES permutation box for all memory values in the given axis. If $axis_{output} = p_{box}(axis_{input})$ holds, we have found an instance of a DES encryption or decryption round. This approach can be applied to every relation contained in an algorithm. Although the hamming weight check is necessary to evade false-positives, especially for permutation boxes.

3. Using the values of an axis, we also check whether the XOR relation holds for combinations of any three values. Therefore, we use the Python `itertools` package to create every possible combination of three values in the given axis. For each of the combinations, we verify whether an XOR relation is valid for the three values, as shown in Listing 4.2.

```python
for combo in itertools.combinations(data, 3):
    if combo[0] ^ combo[1] == combo[2]:
        logging.debug('found xor relation %d ^ %d == %d' % combo)
```

Listing 4.2: Loop differ check for XOR relations

4. A fourth check, which we implemented for the loop differ, searches for changes in entropy. Given an axis, we first separate read from write values. We follow the approach from Lutz and calculate the number of unique bytes, the number of different bytes, and the entropy for reads and writes.

The entropy calculation is normalized, so that the result is in the range $[0, 1]$ for up to 256 input bytes. The exact formula of the entropy $H()$ for the byte array a of length n is

$$H(a) = \frac{-\sum_{256}^{i=1} \frac{a_i}{n} \cdot \log_2 \frac{a_i}{n}}{\log_2(\min(n, 256))}$$

and due to the normalization gives better comparable results for arbitrary length arrays. Furthermore, we implemented the functions to determine the number of unique and different bytes, as shown in Listing 4.3.

```python
def calcEntropy(buf):
    s = 0.0
    n = float(len(buf))

    assert(n > 1)
    for i in buf:
        assert(i <= 255 and i >= 0)

    for i in range(256):
        div = buf.count(i)/n
        if div != 0:
            s += div*math.log(div,2)
    return -s/math.log(min([n, 256]), 2)

def numUniqueBytes(buf):
    d = {}
    for c in buf:
        if d.has_key(c):
            d[c] += 1
        else:
            d[c] = 1
    return d.values().count(1)

def numDifferentBytes(buf):
    return len(set(buf))
```

Listing 4.3: Entropy calculation for the loop differ

When the three functions above are applied to the reads and writes, as shown in Listing 4.4, we check whether the change between the functions outputs of the reads and writes differs. Our experiments have shown that a `changeThreshold = 0.15` detects several entropy changes, but for the exact performance we refer to Chapter 6. The threshold is taken from the paper [49].

```
uniqueIn = numUniqueBytes(reads)/float(len(reads))
differentIn = numDifferentBytes(reads)/float(len(reads))
entropyIn = calcEntropy(reads)

uniqueOut = numUniqueBytes(writes)/float(len(writes))
differentOut = numDifferentBytes(writes)/float(len(writes))
entropyOut = calcEntropy(writes)

if min(abs(uniqueIn-uniqueOut), \
      abs(entropyIn-entropyOut), \
      abs(differentIn-differentOut)) > changeThreshold:
    # success ...
```

Listing 4.4: Entropy check of the loop differ

Besides the loop differ's XOR check, we also developed a special method for XOR encryption. The identification method is based on three facts. First if a `XOR` instruction occurs, it is often used to zero a register, for example `xor eax, eax`. In contrast to that, XOR used in encryption writes a zero to the target register in the minority of cases. Thus, we search for `XOR` instructions in the trace, which do write out zeros in at most 60% of the executions. The threshold of 60% is a heuristic value, which we determined during development. The probability, that during an XOR encryption the resulting value is zero, is at most $1 : 256 \approx 0.39\%$ for a 8 bit value and assuming a randomly chosen XOR encryption key. On the other hand, a `xor eax, eax` returns zero in 100% of the executions. A threshold of at most 60% zero results thus is about in the middle of both extreme values. Instead of 50% we use 60% to shift the probability between false-negative and false-positive towards false-negatives, because a non-random XOR key may easily create 50% zero XOR outputs. Although, a higher threshold of up to 95% may still work well, given that `xor eax, eax` returns zeros in 100% of all cases.

Furthermore, the same `XOR` instruction will be executed multiple times in an encryption or decryption. Thus, we use a threshold of a minimum number of 16 executions of the `XOR` instruction. The malware protocols in Chapter 1 and the related work in Chapter 2 describe several XOR encryption protocols, which at minimum use encryption blocks of 16 bytes. Therefore, we chose a threshold of minimum 16 executions.

The second observation which we use, is the fact that usually a `MOV` instruction occurs nearby the `XOR` instruction. Although, the `MOV` instruction should share the same register as the `XOR` instruction, we do not verify it due to the lack of taint-tracking. Nevertheless, we exclude further candidates based on missing `MOV` instruction near the `XOR` instruction. *Near* is defined here as an interval of instructions around the `XOR` instruction, by default ± 5 instructions.

As a third rule, we use the fact that we can verify the XOR relation using a reference XOR test in the `XOR` instruction's encompassing loop. Therefore, we check whether the XOR relation holds for all combinations of data values used in the loop body for each iteration. A sample output of the detection is shown in Listing 4.5.

```
[DEBUG] assumption holds for 100 % of values
[DEBUG] xor key = 'u \xa6\xd1^\x1b\x19\xaa\x' ...
[DEBUG] xor plaintext = 'DDDD3333DDDD3333DDDD33' ...
[DEBUG] xor ciphertext = '1\xe2\x95\x1a(*\x99\xf5' ...
$ xxd io/rand.0128 # the random XOR key file used by the testing application
0000000: 75a6 d15e 1b19 aac6 49be 6b2c 80d4 f49e u..^....I.k,....
```

Listing 4.5: Sample output of the XOR check

4.4.4. Generic Memory-based

The final identification method, which we developed, is focused on memory data. As mentioned above, we are using verifiers to confirm an XOR encryption or a relationship between the input and output of a permutation box. Using the memory reconstruction method described in Section 3.6, we are able to verify complete instances of a cryptographic algorithm using plaintext, key, and ciphertext residing in memory.

As the memory reconstruction method reassembles cryptographic data of any length, we are able to reconstruct a set of possible key, plaintext, and ciphertext candidates. These candidates are then passed to a reference implementation of the particular algorithm. If the output of the algorithm matches the output in memory, we have successfully identified an instance of the algorithm including its parameters. The process is illustrated in Figure 4.6, where dotted lines indicate further candidates to be checked. The main limitation of the method is the premise that the algorithm is public and our system contains a reference implementation to verify the input-output relation.

Optionally, we can reduce the set of candidates using previous identification methods. For example, if a signature has detected AES code, we can reduce the memory reconstruction to this code section, instead of the complete trace. Furthermore, we only need to check for 128, 192, and 256 bit keys and 128 bit input/output blocks, based on the previous identification of AES.

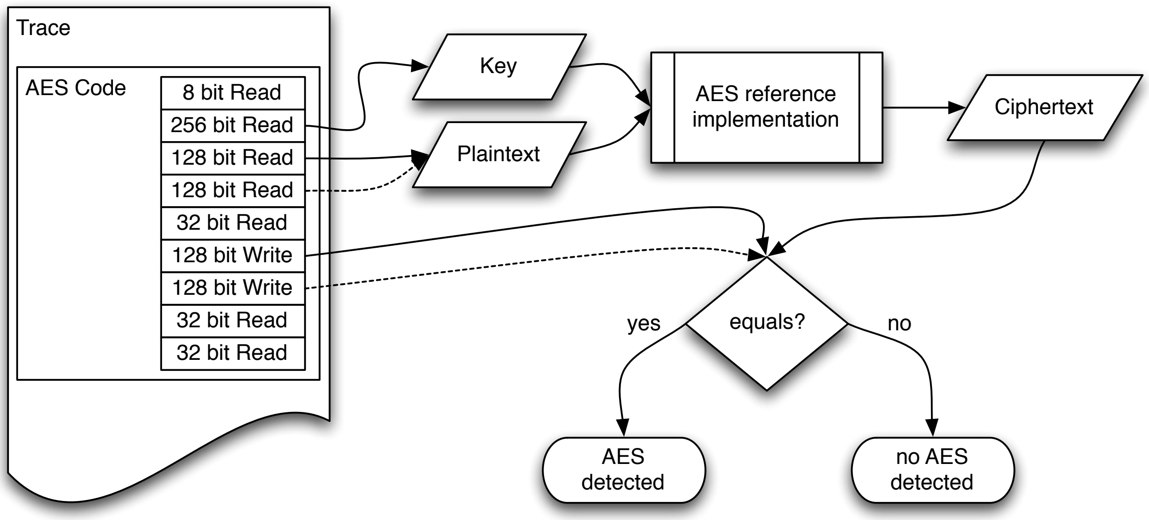


Figure 4.6.: Verification of the algorithmic relation

We do not specifically have to consider and distinguish between encryption or decryption, because the encryption and decryption are commonly the same algorithms for stream and block ciphers. The efficiency of this approach is bound to the amount of candidates. If we can identify specific cryptographic code using other identification methods before, the efficiency is highly increased, since less candidates need to be checked. Of course, the reference implementation has to be optimized to be fast.

Interestingly, this method isolates the cryptographic values from further modifications. Since we only verify and test using the reference implementation, further modifications, i.e., padding, encoding, or compression, can be separated and we detect the exact parameters to the cryptographic algorithm. Because of this soundness of the method, we already can note that we do not encounter false-positives using this method, as shown in the evaluation.

4.5. Summary

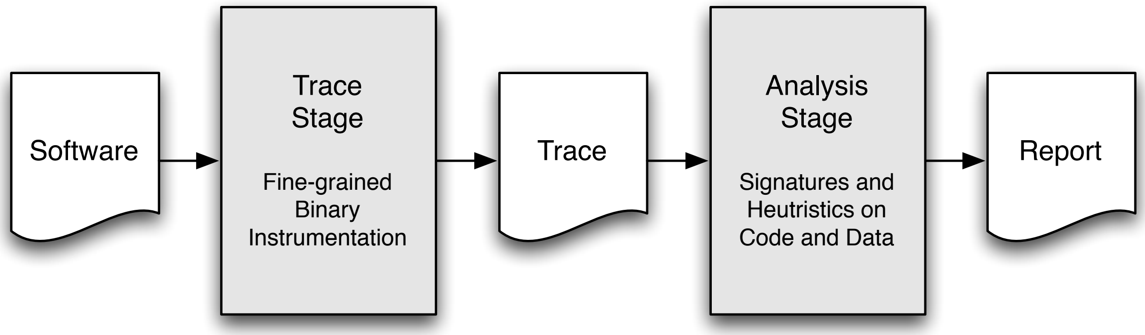
In this chapter we presented several observations upon which our identification methods are built. The identification can be categorized by the type of identification method, signature or generic, but also by the used high-level structure from Chapter 3, for example loop-based or memory-based. The different methods and their parameters, especially their thresholds, are further evaluated in Chapter 6.

5 Chapter 5. System Implementation

In this chapter we present the implementation of the system. First we give an overview and then detail the trace and analysis implementations separately.

5.1. Overview

The system implementation is divided in two stages, which are performed for each analysis of a software sample. A schematic overview of the stages is shown in Figure 5.1.



Software
Trace Stage
Fine-grained Binary Instrumentation
Trace
Analysis Stage
Signatures and Heutristics on Code and Data
Report

Figure 5.1.: Overview of the stages

In the first stage of Figure 5.2, during controlled execution of the target binary program, dynamic binary instrumentation traces the execution to gain insight on the program flow and also monitor the accessed memory.

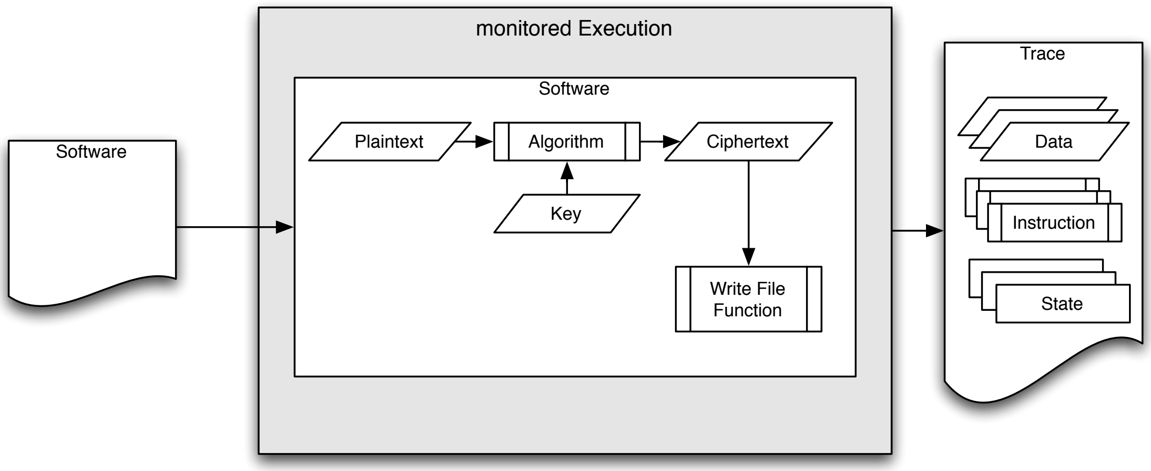


Figure 5.2.: Displaying an overview of implementation stage 1

For the dynamic analysis, we build upon the existing software framework Pin[1]. The framework supports fine-grained instruction-level tracing of a single process. Our implemented Pintool creates a run trace of a software sample to gather the relevant data for the second stage.

In the second stage, the instruction and data trace is used to detect employed cryptographic algorithms, e.g., RC4, MD5, or XOR, and their parameters, e.g., keys or plaintext. An overview of the second stage is shown in Figure 5.3.

To detect the algorithms and their parameters, the analysis methods described in Chapter 3 elevate the trace to high-level structured representations, i.e., loops, graphs, and basic blocks. Then, the identification methods described in Chapter 4 are executed and utilize the high-level representation of the trace to inspect the execution for cryptographic primitives. Based on the findings of the cryptographic algorithms, a report is generated, which displays the results, especially the identified algorithms and their data.

A main design goal is to split the tracing from the analysis tool. Thereby we are able to independently choose different tracing frameworks and exchange them, if necessary, for the Pin based approach described below. But this design decision also poses disadvantages. The main problem is the increased space and time overhead for creating a complete trace file, because the tracing cannot filter and process the trace in real-time and has to save the trace to disk. A core paradigm when creating a Pintool is to filter using the Pin framework and keep the analysis small and efficient. We ignore this paradigm, trace everything and migrate the analysis to another software, in order to flexibly choose between tracing frameworks.

[1]See http://pintool.org/.

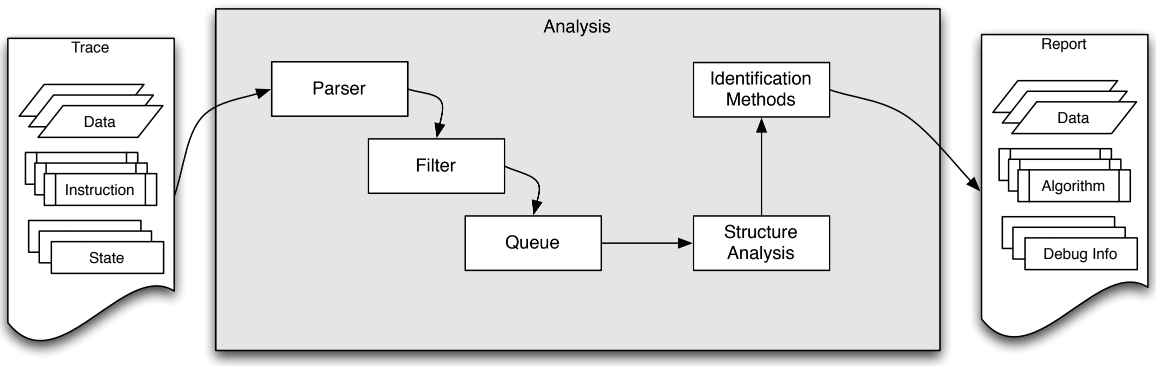


Figure 5.3.: Displaying an overview of implementation stage 2

We chose this approach to be independent of the characteristics of the tracing framework, because the successful runtime tracing depends on the features of the framework. If the program cannot be run under Pin, because it contains anti-debugging code, we can still use another framework with available tracing functionality.

5.2. Tracing

As mentioned before, the software sample is traced using the Pin dynamic binary instrumentation framework [48]. Pin is distributed by Intel free of charge and provides a rich API to develop instrumentation tools, called Pintools. We developed a Pintool to conduct the trace. It is written in 632 lines / 464 sloc[2] of C++ code.

Pin supports tracing spawned processes by using the `-follow_execv` command line switch. Although we did not fully evaluate the performance of this switch, we recognized the problem that it is not able to follow started services, malware-manipulated processes, or out-of-process COM objects. Therefore, we focus on a single process. This limitation could be removed in future work, because a different tracing solution may follow the in-system-propagation of malware.

To conduct a runtime trace of a software sample, we launch the target executable using Pin and our custom Pintool `kerckhoffr.dll`:

```
pin.exe -t kerckhoffr.dll -- target.exe
```

Listing 5.1: Running the Pintool

[2]According to SLOCCount 2.26 (http://www.dwheeler.com/sloccount/).

Pin itself supports multiple command line options. For example, `-follow_execv`, to follow spawned processes, and `-smc_strict`, to handle self-modifying code. The double hyphen `--` in Listing 5.1 separates Pin command line parsing from the `target.exe` command line. Additionally, our Pintool is adjustable using the following options:

- `-mw` defines the module whitelist to only include the listed modules. The switch can be specified multiple times. If the switch has been specified, modules, which are not in the whitelist, are not traced. Thereby, we reduce the needed disk/memory space.

- `-o` specifies the output file name of the trace.

- `-istart` sets the number of instructions, that should be skipped, before the instrumentation starts. Furthermore, `-istop` specifies after which number of instructions the instrumentation should stop. Both switches are useful to select a specific time frame of the target executable. Thereby, we can skip the process starting phase, for example unpacking, and terminate the process after the necessary trace data has been recorded.

- `-watch_thread` determines whether a specific thread should be traced. If it is not specified, all threads are traced.

After launching the command in Listing 5.1 Pin loads the target program and the Pintool starts. It then generates a trace file without further user interaction and quits.

In the following pages we provide an overview of the Pintool's implementation. For code details, please refer to Appendix A. In the main function the Pintool first initializes Pin, then the whitelist, and at last the `-istart` and `-istop` alarms. Then it specifies the main instrumentation function, the `PIN_AddFiniFunction()` handler and starts the target program using `PIN_StartProgram()`.

The instrumentation function handles the newly discovered code, which is encountered by Pin for the first time. Therefore, it iterates over a `TRACE` instance using `TRACE_BblHead()` and `BBL_Next()` to check whether the module is contained in the whitelist. The module is determined by `IMG_Name(IMG_FindByAddress(BBL_Address(bbl)))`. The whitelist is implemented using a `std::set<std::string>`. If the module is in the whitelist, or the whitelist has not been specified, we traverse through the `BBL` using `BBL_InsHead()` and `INS_Next()` to call two functions for each instruction: `MemoryTrace()` and `InstructionTrace()`. Each of them prepares the analysis code, which is executed every time this particular instruction is executed. Both functions prepare the trace string which is written to the trace file, as shown in Listing 5.2.

The first analysis function, `MemoryTrace()`, formats the trace string for a memory access.

Depending on whether an instruction triggers one or two reads before it is executed, and whether it does a memory write after it is executed, the `MemoryTrace()` is called. The function uses `INS_InsertPredicatedCall()` to prepend and append analysis code to the instruction in order to emit the trace string for the memory access. The memory values are copied using `PIN_SafeCopy()`.

The second analysis function, `InstructionTrace()`, formats the trace string for an instruction. It determines the current `EIP` using `INS_Address()` and debug symbols with `IMG_Name(IMG_FindByAddress())` and `RTN_Name()`. The debug symbols, i.e., module and function name, are compressed as detailed below. Thereby, we can reduce the disk space of the trace file. In `InstructionTrace()`, we also emit the current thread ID with `PIN_ThreadId()` and a disassembly of the current instruction using `INS_Disassemble()`. We also print the registers modified by the instruction.

The trace syntax is shown in Listing 5.2. Each line beginning with `R` or `W` denotes a memory access. Lines beginning with `0x` describe instructions. The pipe character, `|`, is a delimiter.

```
R|32|0022F948=0x22f9a4
0x7c9111f3|@1|@2|0x0016|0|mov esi, dword ptr [ebp+0x8]|esi=0x22f9a4
```

Listing 5.2: Example of the trace file

The memory access in line one is a read access (`R` instead of `W` for write) and reads `32` bits starting for the address `0022F948`. The literal, which was read, is `0x22f9a4`.

The memory access belongs to the instruction in line two of Listing 5.2. In this line, the first field separated by the delimiter denotes the current instruction pointer, `0x7c9111f3`. The second and third field contain compressed debug information. The compressed fields describe the current module `@1` and the current function `@2`. Module and function information is compressed using the syntax `@1@C:\WINDOWS\system32\ntdll.dll` or `@2@RtlDeactivateActivationContextUnsafeFast`. The compression occurs when an unknown symbol is encountered for the first time. Afterwards, if we parse a compressed module denoted by `@1` in the trace, we can decompress it by looking up `@1` in a dictionary which then resolves to the value `ntdll.dll`. The compression is helpful to decrease the trace size, because an uncompressed trace would list the full debug information for every instruction. For a trace with 100 instructions with the above exemplary debug information, the compressed size would be composed by the length of the compression ($3 + 28 + 3 + 40 = 74$) and 99 compressed module and function fields ($99 \cdot 2 \cdot 2 = 396$). An uncompressed trace would contain the full string in both fields for 100 times ($40 \cdot 100 + 28 \cdot 100 = 6800$). In this example we can decrease the file size by the factor $14.46 \approx \frac{6800}{396+74}$.

The fourth field, `0x0016`, describes the offset to the start of the current function. The fifth field is the current thread ID, `0`, then follows the disassembly of the current instruction, `mov esi, dword ptr [ebp+0x8]`. In the next fields we store an arbitrary length delimited list of modified registers. For example, if `XOR` modifies `eax` and `eflags`, we parse a trace string like `xor eax, eax|eax=0|eflags=0x246`.

5.3. Analysis

In this section we describe the general architecture of the analysis. Please refer to Chapter 3 and 4, and Appendix A for the exact implementation.

We developed the analysis as a Python tool. Python was chosen because the author is most familiar with it and it allowed us to flexibly implement the analysis, i.e., easily adjust class definitions during development and use dynamic typing on instance attributes. Using the interactive console, it also proved to be a flexible experimentation environment, and we were able to try new analysis methods interactively. The tool consists of 3598 lines or 2204 sloc of Python code. It is based on Python 2.6.4 and utilizes two external Python packages[3] [4] and the following standard Python packages: `logging`, `os`, `sys`, `re`, `itertools`, `struct`, `hashlib`, `math`, `struct`, and `time`.

The analysis tool is started from the command line and shows the usage in Listing 5.3:

```
# python kerckhoffr/
Usage: python kerckhoffr/ <single|full|debug|cfg|parse> <inputfile1> <inputfile2>..
```

Listing 5.3: Running the analysis tool

The tool accepts multiple modes of operation:

- `single`, to first parse the trace and then analyze it in sequence,
- `full`, to parse the trace file and analyze it in parallel, threaded mode,
- `debug`, to parse a trace file and launch an interactive console to manually inspect the trace,
- `cfg`, to generate a control flow graph PDF from the trace, and
- `parse`, to test the syntax parsing of trace files.

[3] Crypto.Cipher - *PyCrypto The Python Cryptography Toolkit* http://www.pycrypto.org/.
[4] IPython.Shell - *IPython: an interactive computing environment* http://ipython.scipy.org.

When launched in `single` or `full` mode, the analysis tool will parse, filter and process the trace using the code analysis and cryptographic identification methods. The output will contain logging information on several levels, including debug information on the analysis algorithms and results from the identification methods. An example is shown in Listing 5.4, although we shorten the output of the analysis tool in further listings and remove the date and file information.

```
2010-01-31 20:19:05,578 __main__.py:<module>@58 [INFO] running single (non-threaded)↩
     mode on file log-small/kerck-cryptopp_aes.out...
2010-01-31 20:19:05,893 Parser.py:run@64 [INFO] 6 % <Instructions with 6832 elements↩
    , 6832 total, maxsize 750000>
[...]
2010-01-31 20:19:12,368 Parser.py:run@64 [INFO] 95 % <Instructions with 102347 ↩
    elements, 102347 total, maxsize 750000>
2010-01-31 20:19:12,968 Parser.py:run@90 [DEBUG] parsing finished (EOF).
2010-01-31 20:19:12,968 Analysis.py:analyze@44 [DEBUG] running analysis on <↩
    Instructions with 106764 elements, 106764 total, maxsize 750000>
2010-01-31 20:19:14,129 Analysis.py:wang@279 [INFO] wang reformat transistion ↩
    function (0.618059) unnamedImageEntryPoint
[...]
```

Listing 5.4: Sample output of the analysis tool

When launched in `debug` mode, we are able to manually inspect the trace. We can use the software analysis functions to interact with the parsed trace. For the example in Listing 5.5, we use the API to search for basic blocks, which have at least six instructions and a bitwise percentage of at least 60%, and print the results (user input in line 6-8). The result is one basic block, shown in line 16.

```
 1 # python kerckhoffr/ debug log/kerck-xor256.out
 2 [INFO] running debug mode with ipython on file log/kerck-xor256.out...
 3 [DEBUG] parsing finished (EOF).
 4
 5
 6 In [2]: for bbl in instructions.getUniqueBBLs():
 7    ...:     if bbl.isBitwiseArithPercentage() > 0.6 and len(bbl) > 5:
 8    ...:         print repr(bbl)
 9    ...:
10 [DEBUG] creating BBLs step 1
11 [DEBUG] creating BBLs step 2
12 [DEBUG] creating BBLs finished <353 BBLs (executed) first starting with instruction ↩
       <Instruction 0x401446 z:\2_dev\exe\xor256.exe:mainCRTStartup+0x0 'call 0x4018c8'↩
        <Register esp=0x12ffc0> <Write (32) 0x12ffc0=0x40144b>>>
13 [DEBUG] creating CFG step 1: unique bbls, instruction data array
14 [DEBUG] creating CFG step 2: create edges
15 <Instructions 0x401130 main (6): <mov dl, byte ptr ss:[esp+ecx*1+0x10]> <xor dl, ↩
       byte ptr ss:[esp+ecx*1+0x90]> <inc ecx> <cmp ecx, eax> <mov byte ptr ss:[esp+ecx↩
       *1+0x10f], dl> <jl 0x401130> >
```

Listing 5.5: Sample experimentation session using the analysis console

5.3.1. Architecture

In the following pages, we describe the architecture of the analysis tool. The classes are distributed among several files and the tool is arranged to a Python package with following submodules and subpackages:

- `kerckhoffr/` denotes the main package and contains all other subpackages.
- `kerckhoffr/Analysis.py` implements the cryptographic identification in methods of the `Analysis` class. The analysis methods, `caballero()`, `chains()`, `constmemory()`, `constmnemonic()`, `lutz()`, `sigAPI()`, `wang()`, `xorNotNullAndMov()`, `loopDiffer()`, and `symmetricCipherDataTester()`, implement the identification methods described in Section 4.4.
- `kerckhoffr/CryptoReport.py` implements the `CryptoReport` class for report generation. The result of an identification method can be saved as an instance of `CryptoResult`, defined in `kerckhoffr/CryptoResult.py`. The instances are then appended to the list `Instruction.cryptoResults`.
- `kerckhoffr/Instruction.py` implements the class for an `Instruction`. Each instruction contains several defining attributes, e.g., `eip`, `data`, or `disasm`. If an instruction has been executed multiple times, different `InstructionData` instances are saved to the `multiData` attribute. The class also implements several methods, for example,
  - `disConst()`, to determine constants in the disassembly,
  - `mnemonic()`, to extract the mnemonic from the disassembly,
  - `isBitwiseArith()`, `isMov()`, and `isBranch()`, to determine the type of instruction, or
  - `modulefile()`, to determine the file name of the module.
- `kerckhoffr/InstructionData.py`. Each register and memory value accessed by an instruction is modeled as an instance of `InstructionData`. Therefore, the `InstructionData` class contains the attributes: `addr`, the target address of the memory access, `data`, the memory value which was read or written, `size`, the amount of bits read or written, and `mode`, defining whether the access was a read or write. Besides helper methods like `isMemoryWrite()`, the class defines the method `data2bytes`, which converts the hex-string based memory value from the trace file to a byte-string. For example, `'0xdead'` will be converted to `'\xde\xad'`.
- `kerckhoffr/InstructionTypes.py` implements the mapping of mnemonics to instruc-

tion types using a table generated from [45] as described in Section 4.4.2. To query the mapping, the methods `ins2type()` and `type2desc()` are implemented.

- `kerckhoffr/Instructions.py` contains the classes `Instructions`, for generic encapsulation of a sequence of instructions, `BBLs`, for basic block generation, `Loops`, for loop detection, `RTNs`, for extracting instructions belonging to a single function, `leafRTNs`, for detection of leaf routines, and `CFG`, for control flow graph generation.

  The `Instructions` class implements a sequence of instructions. It can be either used as a first-in-first-out (FIFO) queue or as a static sequence of instructions. The class is used by several other classes in this file, e.g., to describe basic blocks. An object is initialized with either `insertInstruction()` or `importInstructionList()`. It contains 37 methods to inspect, modify, and evaluate the instructions. For example, the method `instructionTypes()` determines the distribution of different types of instructions among the `Instructions`. The method `getNearbyBlocks()` implements the memory reconstruction method described in Section 3.6. The mnemonic signature generation is implemented in `mnemonicChain()`. Furthermore, the `Instructions` class also contains interfaces to generate the high-level structures and caches them, so that they only need to be generated once.

  The analysis methods from Section 3.3, 3.4, and 3.5 are implemented in the classes `BBLs`, `Loops`, and `CFG` respectively. The classes' initialization resolves the instructions, which builds the base for the high-level structures, from the calling `Instructions` instance. Furthermore, we provide access to executed functions using `RTNs` and leaf routines using `leafRTNs`, as described in Section 4.4.2.

- `kerckhoffr/Math.py` contains shared utilities for the analysis methods. The methods are: `hammingOnePercentage()`, for determining the hamming weight, `calcEntropyLutz()` and `calcEntropy()`, for calculating the normalized and common entropy, `numUniqueBytes()` and `numDifferentBytes()`, for counting the number of unique respectively different bytes in a list.

- `kerckhoffr/Parser.py` implements the parsing of the trace file. The process is described in detail below.

- `kerckhoffr/__init__.py` is the obligatory file defining a Python package.

- `kerckhoffr/__main__.py` implements the command line parsing and initializes common objects, as described below.

- `kerckhoffr/signatures/` is a subpackage containing several signatures for the identification methods described in Section 4.4.1: `api.py`, `chainsForImplementation.pkl`, `memory.py`, `memoryreadnowrites.py`, `mnemonicchains.py`, and `mnemonicconst.py`.

- kerckhoffr/verifier/ is a second subpackage providing interfaces to the reference implementations of the listed algorithms: aes.py, des.py, md5.py, rc4.py, rsa.py, and xor.py. The modules partially depend on reference implementations from Crypto.Cipher and hashlib.

5.3.2. Process

When the analysis tool is started, the __main__ module parses the command line, configures the logging module, initializes an instance of Instructions as the main queue (step one in Figure 5.4) and passes the instance to the instantiation of Parser and Analysis (step two and three). Depending on the command line mode, whether the parsing and analysis should be threaded or not, the __main__ module starts two threads in step four and five, or iteratively calls the respective functions, i.e., run(). Since the parsing and the analysis are concurrent methods, which may be run in parallel, the option to run the tool in threaded mode is given to the user. A complete parallel processing is not yet feasible, because both concurrent methods share the same data, i.e., the Instructions instance. Thus, the analysis method locks the Instructions instance, so that the parsing method does not modify the instance during analysis.

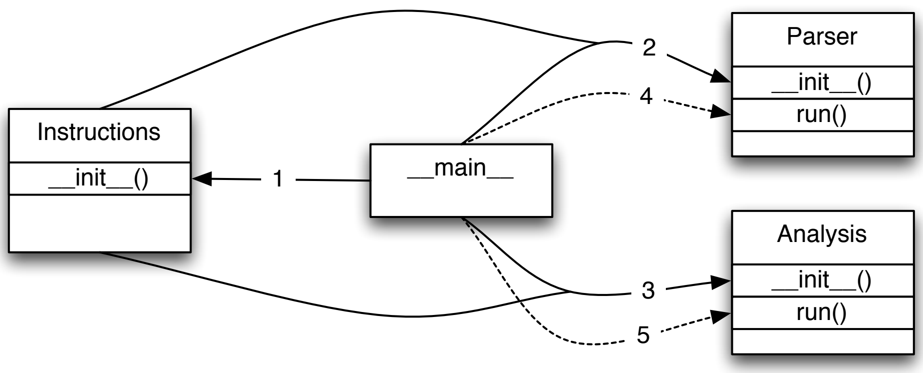


Figure 5.4.: Initialization of objects

Implemented in Parser.py, the method Parser.run() opens the trace file and associates memory and instruction entries for each contained line. It then calls Parser.generateInstruction() to create an instance of Instruction and appends it to the target queue: Instructions.insertInstruction(). Parser.generateInstruction() uses the method Parser.lookup() to expand compressed symbol information. If the module name does not match the specified filter, the instruction can be discarded. Also, Parser.generateInstruction() calls Parser.checkBufferSize() to verify whether it can still fill the queue, or otherwise has to call pauseParsing() to pause the parsing of new lines.

The restriction of the queue is due to the memory constraints and the pauseParsing()

functionality assures that the queue is first analyzed, before it will be refilled again. To thwart a cropping of the trace queue at an unfavorable position, i.e., inside a cryptographic function, the FIFO queue is not fully flushed before a new analysis session is started. The analysis is already called when 75% of the queue is refilled. This leads to an overlapping of 125,000 instructions for the default FIFO queue size of 500,000 instructions[5], which is sufficient for most cryptographic functions, as shown by the instructions column in Table 6.3.

When the parsing is finished or the FIFO queue triggered `pauseParsing()`, the analysis starts. The method `Analysis.run()` then calls the identification methods listed above, which have access to the `Instructions` queue using the attribute `Analysis.target`. When the analysis is done and the results have been recorded, the tool either quits or continues parsing and analyzing the next block.

5.4. Summary

In this chapter, we described a system implementation composed by a Pintool and a Python analysis tool. We also discussed the design decisions encountered during the development. The system could be extended and exchanged in several parts, but we show in the next chapter, that it performs well for a variety of software.

[5]The default size is suitable for the development and evaluation system described in Section 6.1.

6 Chapter 6. Experimental Evaluation

In this chapter we evaluate the performance of the previously described methods and their implementation. First, we give an overview of the testing environment and then describe the system's performance for the testing applications, an off-the-shelf application, and a packed testing application.

6.1. Evaluation Environment

The tracing is performed in a Sun VirtualBox 3.1.2 running Windows XP SP3 which is hosted on Mac OS X 10.6.2. The Pin version is 2.7-31933. The virtual machine is configured to have 1024 MB of RAM and operates with a single core of the host computer. The trace is written to the disk of the host computer through a VirtualBox shared folder.

The host computer, on which the analysis runs, provides a 2.4 GHz Intel Core 2 Duo with 4 GB of RAM. The FIFO queue size of the analysis is by default 500,000 instructions. With a fully loaded queue the analysis process uses about 1.9 GB of RAM.

6.2. Testing Applications

For the evaluation, we developed 14 testing applications. The testing applications consist of 1102 lines or 835 sloc of C code in total. The applications take input parameters, e.g., two files holding plaintext and key, on the command line. The mode of operation, i.e., encryption or decryption, and the output file are also specified on the command line. Then, the testing applications process the input files, initialize the cryptographic library including the algorithm, and encrypt or decrypt the plaintext file. Finally, the result is written to the output file. An overview of the cryptographic libraries' versions, used compilers and mode of operation is given in Table 6.1. We illustrate the sizes of the parameters in Table 6.2.

The compilers used were the Microsoft C/C++ Compiler version 15.00.21022.08 and the MinGW port of GCC version 3.4.2. Some cryptographic libraries were linked statically, others dynamically, to test the Pintool's handling of dynamically loaded libraries.

We created a batch job launching the Pintool to generate the trace files of the testing applications. The duration of the Pintool instrumenting the different testing applications depends on the command line configuration. Our Pintool's module whitelist contained only the executable and, if dynamically linked, also the DLLs of the cryptographic library, e.g., `beecrypt.dll` and `LIBEAY32.dll`. Otherwise, if we would also trace Windows libraries like `ntdll.dll`, the tracing time and size would increase.

For example, if we trace the OpenSSL RSA encryption and do not enable the module whitelisting, we would generate trace of about 20 GB in file size with 97% of traced code in `rsaenh.dll`. This is due to the random number generation of the OpenSSL PRNG implementation in `crypto/rand/rand_win.c`. There, OpenSSL calls `CryptAcquireContextW` and `CryptGenRandom` of `rsaenh.dll` and thereby increases the trace size and time. For the XOR testing application with a 256 byte input/output, the difference between an unfiltered and filtered trace is an increase of factor 29 in size (137 KB versus 4 MB), factor 23 in time (2 versus 46 seconds), and factor 28 in instructions (1935 versus 54585 instructions).

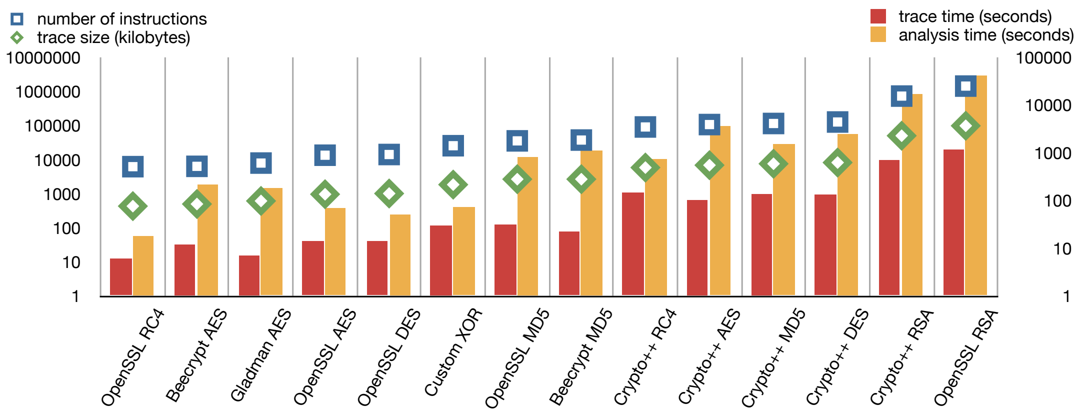


Figure 6.1.: Analysis and trace time/size costs, logarithmic scaled

The duration and trace file sizes of the filtered tracing are shown in Table 6.3. Hours are abbreviated by h, minutes by m, and seconds by s in the duration fields. We plot

Implementation	Algorithm	Version	Compiler	Mode
Beecrypt	AES	4.1.2	VC dynamic	ECB encryption
Brian Gladman	AES	07-10-08	VC static	CBC encryption
Cryptopp	AES	5.6.0	VC static	CFB encryption
OpenSSL	AES	0.9.8g	MinGW static	CFB encryption
Cryptopp	DES	5.6.0	VC static	CFB encryption
OpenSSL	DES	0.9.8g	MinGW static	ECB encryption
Cryptopp	RC4	5.6.0	VC static	encryption
OpenSSL	RC4	0.9.8g	MinGW static	encryption
Beecrypt	MD5	4.1.2	VC dynamic	
Cryptopp	MD5	5.6.0	VC static	
OpenSSL	MD5	0.9.8g	MinGW static	
Cryptopp	RSA	5.6.0	VC static	OAEP SHA1
OpenSSL	RSA	1.0.0-beta3	VC dynamic	PKCS1.5
Custom	XOR	1.0	VC static	

Table 6.1.: Overview of testing applications

Implementation	Algorithm	Keysize	Inputsize	Outputsize
Beecrypt	AES	128 bit	128 bit	128 bit
Brian Gladman	AES	128 bit	128 bit	128 bit
Cryptopp	AES	128 bit	128 bit	128 bit
OpenSSL	AES	256 bit	128 bit	128 bit
Cryptopp	DES	128 bit	128 bit	128 bit
OpenSSL	DES	128 bit	128 bit	128 bit
Cryptopp	RC4	128 byte	128 byte	128 byte
OpenSSL	RC4	128 byte	128 byte	128 byte
Beecrypt	MD5	-	4096 byte	128 bit
Cryptopp	MD5	-	4096 byte	128 bit
OpenSSL	MD5	-	4096 byte	128 bit
Cryptopp	RSA	1024 bit	128 byte	128 byte
OpenSSL	RSA	512 bit	128 byte	144 byte
Custom	XOR	128 byte	4096 byte	4096 byte

Table 6.2.: Parameter size of testing applications

Implementation	Algorithm	Trace size	Trace duration	Number of Instructions	Analysis duration
Beecrypt	AES	497 KB	12s	6342	3m36s
Brian Gladman	AES	611 KB	7s	7913	3m
Cryptopp	AES	6.7 MB	1m42s	106764	46m4s
OpenSSL	AES	962 KB	14s	13449	1m9s
Cryptopp	DES	8.0 MB	2m12s	126929	41m3s
OpenSSL	DES	1011 KB	14s	14104	51s
Cryptopp	RC4	5.7 MB	1m25s	90695	12m19s
OpenSSL	RC4	431 KB	6s	6201	18s
Beecrypt	MD5	2.6 MB	22s	37460	18m39s
Cryptopp	MD5	7.3 MB	1m17s	114759	25m15s
OpenSSL	MD5	2.6 MB	31s	35455	13m35s
Cryptopp	RSA	49 MB	11m41s	731430	4h43m49s
OpenSSL	RSA	96 MB	19m18s	1437457	11h38m52s
Custom	XOR	1.8 MB	30s	25545	1m12s

Table 6.3.: Analysis and trace time/size costs

the results in Figure 6.1, where we convert the size in kilobytes and the analysis duration in seconds. Also, we illustrate the duration of the analysis with almost all[1] identification methods enabled. The large analysis duration is partially due to the fully enabled analysis methods. Especially the `loopDiffer()` is very time intensive and is responsible for about 90% of the analysis duration in the RSA cases.

The performance of the analysis is rated by the successful identification of the cryptographic algorithm and parameters. Therefore, we analyze each trace of a testing application and review which identification method has identified the correct cryptographic algorithm.

6.2.1. Published Methods

At first, we evaluate the identification methods from the related work: `caballero()`, `lutz()`, and `wang()`. Although, one has to keep in mind that we did not exactly implement Lutz's identification method due to the lack of a taint-tracking functionality. Table 6.4 shows the results of the identification methods. False-positives are abbreviated as FP and basic blocks as BBL. The results of the identification were compared with the source code and control flow graphs of the testing application in order to rate the performance of the methods.

[1] Excluding the `symmetricCipherDataTester()` because the high time duration of the memory reconstruction.

Implementation	Algorithm	caballero	lutz	wang
Beecrypt	AES	success	found BBL	no result
Brian Gladman	AES	success	only FP	no result
Cryptopp	AES	partial	found BBL	error
OpenSSL	AES	success	found BBL	success `OPENSSL_cleanse`
Cryptopp	DES	success	found BBL	error
OpenSSL	DES	success	key schedule	success `DES_ecb_encrypt`
Cryptopp	RC4	partial	only FP	error
OpenSSL	RC4	success	no results	no result
Beecrypt	MD5	success	found BBL	success `md5Process`
Cryptopp	MD5	success	found BBL	error
OpenSSL	MD5	success	partial	success `MD5_Final`
Cryptopp	RSA	success & FP	only FP	error
OpenSSL	RSA	no success & FP	only FP	no result
Custom	XOR	n/a	n/a	n/a

Table 6.4.: Analysis performance for the published methods

Generally, the method of Caballero et al. has a good success rate despite its simplicity. It always identifies the cryptographic basic blocks of the cipher and the hash implementations. It also identifies the key scheduling basic blocks and we rate this as a successful identification, because key scheduling is a core part of cryptographic algorithms. For example, for the OpenSSL DES testing application, the `caballero()` method enumerates 13 basic blocks from the functions `_DES_ecb_encrypt`, `_DES_encrypt1`, and `_DES_set_key_unchecked`. For two Cryptopp applications, the method only partially identifies the set of basic blocks: it misses parts of the key scheduling and the encryption phase. In case of the Cryptopp RSA testing application, the method successfully identifies the asymmetric encryption, but also lists several false-positive basic blocks. For the OpenSSL RSA implementations, the method only identifies false-positive basic blocks from the functions `SHA1_Final`, `SHA512`, and `EVP_DecodeBlock`. The XOR encryption basic block is not found, because it only contains one bitwise arithmetic instructions amount six total instructions, as shown in Figure 3.1.

The method of Lutz [49] cannot be completely evaluated, because we did not implement the taint-tracking needed for it. However, we can note, that using data comparison without taint-tracking, the method is still able to identify cryptographic code. For the AES and DES testing applications, it identifies encryption basic blocks or key schedule blocks, due to entropy changes in the data. Also for the MD5 applications, it identifies the core MD5 functions. Although, with each successful identification, there is also a high rate of false-positives. For testing applications with few loops, i.e., XOR and OpenSSL RC4, the method shows no results, because the loop bodies or iterations are to small. In all results, the identification of plaintext or ciphertext is not successful.

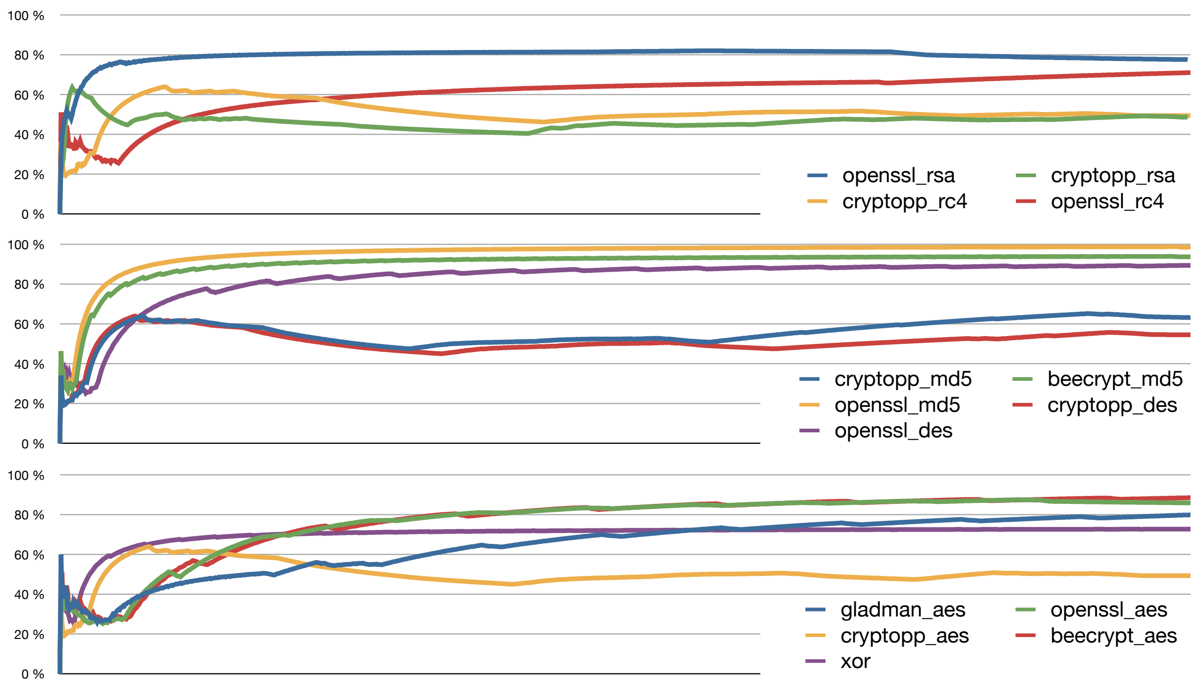


Figure 6.2.: Cumulative entropy for all testing applications

The cumulative bitwise percentage method by Wang et al. shows a good success rate for the testing applications with debug symbols. The method is based on the identification of functions by their debug symbols, therefore it yields false-positive or no results for the testing applications without debug symbols: the Cryptopp applications and Gladman's AES implementation do not contain debug symbols. Nevertheless, for the Beecrypt and OpenSSL applications the success rate is 57%.

The transition functions were verified using the source code and the control flow graphs and are shown in the result table. The cumulative bitwise arithmetic percentages for all testing applications are shown in Figure 6.2. In most cases one can visually identify the point where the encryption starts. Some implementations share a similar progression, e.g., Cryptopp DES and MD5. Over the progression, we can also identify cryptographic rounds by small edges, e.g., for the OpenSSL DES or Beecrypt MD5 applications.

6.2.2. Signature-based Methods

Next, we evaluate the signature-based identification methods: `chains()`, `constmemory()`, `constmnemonic()`, and `sigAPI()`. The results of the evaluation are shown in Table 6.5. Empty cells indicate no results. We would like to note that the XOR testing application was not detected by a signature, because we did not create signatures for it. The XOR encryption does not contain algorithm-specific constants or mnemonic sequences and also most implementations do not provide a dedicated function for it. Therefore, a signature-based identification is not feasible for XOR encryption.

Implementation	Algorithm	sigAPI	constmemory	constmnemonic	chains
Beecrypt	AES	success		success	success
Brian Gladman	AES			success	success
Cryptopp	AES		success	success	success
OpenSSL	AES	success	success	success	success
Cryptopp	DES		success	success	success
OpenSSL	DES	success		success	success
Cryptopp	RC4			success	success
OpenSSL	RC4	success		success	success
Beecrypt	MD5	success		success	success
Cryptopp	MD5			success	success
OpenSSL	MD5	success		success	success
Cryptopp	RSA		FP	success	success
OpenSSL	RSA	success		success	success & FP
Custom	XOR	n/a	n/a	n/a	n/a

Table 6.5.: Analysis performance for the signature-based identification methods

The naïve `sigAPI()` method, to check for cryptographic function symbols, is only efficient when symbol information is included in the code. If we presume this fact, we can also argue that an manual identification by a `strings` command would yield the same information.

The performance of the `constmemory()` identification method is only moderate, because of the loose connection of the constants to the algorithm. To verify this deduction, we inspected the constants in the traces:

```
AES = '10', '20', '1', '0', '5', '65', '1d'
DES = '10', '20', '22', '7a', '33', '1', '0', '2d', '5', '4', '8', '7c817074', '65'
MD5 = '10', '0', '10b', '1', '7c8099c0', '4', '7c817074'
```

Listing 6.1: Intersecting constants of the implementations

The results from Listing 6.1 suggest that the set of intersecting constants of the AES, DES, and MD5 algorithm is (a) very small, (b) not verify descriptive due to common numbers, e.g., `10`, and (c) overlapping between the algorithms. Thus, we conclude that the method is not sufficient for an identification. This conclusion is underlined by Figure 6.3, which shows the performance of the method using signatures derived from the static tools from Section 2.1. In the figure all values are percentages. We can note that the AES constants have a moderate identification rate, but also yield false-negatives. The MD5 and DES constants have a high false-positive rate and are not usable.

	beecrypt	beecrypt	cryptopp	cryptopp	cryptopp	cryptopp	cryptopp	gladman	openssl	openssl	openssl	openssl	openssl	custom
	aes	md5	aes	des	md5	rc4	rsa	aes	aes	des	md5	rc4	rsa	xor
aes t1			30 %				11 %		80 %					
aes t2			60 %				26 %		69 %					
aes t3			57 %				27 %		33 %					
aes t4			58 %				24 %		80 %					
aes t5	1 %	1 %			1 %						1 %			
aes t6														
aes t7														
aes t8														
aes t9														
aes t10	1 %	1 %			1 %						1 %			
md5 t1	46 %	13 %					60 %	53 %	53 %	33 %	13 %	20 %		33 %
md5 t2														
md5 t3														
des all	18 %	7 %						44 %	46 %	13 %	9 %	12 %		50 %
des spbox	7 %	4 %	17 %	77 %	13 %	12 %		8 %	9 %	17 %	4 %	4 %	11 %	5 %

Figure 6.3.: Memory constant identification method for all testing applications

The mnemonic sequences used by the `chains()` method have a similar problem. Since the signatures are partially generated from the testing applications, their matching performance seems successful in the evaluation. But if we evaluate against slightly different code, we assume that the detection rate decreases. Therefore, a fuzzy matching algorithm for the mnemonic sequence comparison could mitigate the problem.

	beecrypt	beecrypt	cryptopp	cryptopp	cryptopp	cryptopp	cryptopp	gladman	openssl	openssl	openssl	openssl	openssl		
	aes	md5	aes	des	md5	rc4	rsa	aes	aes	des	md5	rc4	rsa	xor256	xor4096
rc4 unique	0 %	0 %	100 %	100 %	100 %	100 %	100 %	0 %	50 %	0 %	0 %	100 %	0 %	0 %	0 %
des unique	0 %	0 %	44 %	100 %	44 %	44 %	44 %	22 %	33 %	100 %	11 %	0 %	0 %	0 %	0 %
rsa unique	22 %	8 %	58 %	61 %	50 %	46 %	89 %	34 %	18 %	1 %	7 %	1 %	89 %	0 %	0 %
md5 unique	0 %	100 %	6 %	29 %	100 %	6 %	12 %	0 %	0 %	0 %	100 %	0 %	0 %	0 %	0 %
rc4 intersect	68 %	68 %	100 %	100 %	100 %	100 %	95 %	64 %	77 %	77 %	68 %	100 %	68 %	59 %	59 %
aes intersect	100 %	82 %	100 %	100 %	82 %	82 %	94 %	100 %	100 %	88 %	88 %	71 %	88 %	59 %	59 %
des intersect	56 %	51 %	87 %	100 %	77 %	77 %	82 %	51 %	74 %	100 %	64 %	46 %	64 %	38 %	38 %
rsa intersect	34 %	28 %	71 %	71 %	63 %	57 %	93 %	41 %	35 %	24 %	29 %	16 %	92 %	12 %	12 %
md5 intersect	40 %	100 %	60 %	67 %	100 %	52 %	62 %	26 %	45 %	43 %	100 %	38 %	52 %	36 %	36 %
rc4 cryptopp	13 %	14 %	83 %	82 %	82 %	100 %	57 %	16 %	17 %	16 %	15 %	11 %	31 %	8 %	8 %
rc4 openssl	60 %	58 %	68 %	63 %	58 %	55 %	65 %	38 %	55 %	53 %	50 %	100 %	45 %	53 %	53 %
aes beecrypt	100 %	33 %	35 %	34 %	27 %	27 %	58 %	62 %	41 %	29 %	27 %	26 %	40 %	24 %	24 %
aes gladman	41 %	12 %	27 %	28 %	23 %	22 %	45 %	100 %	21 %	17 %	13 %	11 %	32 %	8 %	8 %
aes cryptopp	12 %	13 %	100 %	73 %	64 %	62 %	59 %	15 %	16 %	14 %	14 %	10 %	29 %	6 %	6 %
aes openssl	52 %	34 %	56 %	55 %	47 %	47 %	62 %	40 %	100 %	45 %	37 %	30 %	52 %	26 %	26 %
des cryptopp	12 %	14 %	74 %	100 %	65 %	62 %	53 %	15 %	15 %	15 %	15 %	10 %	29 %	6 %	6 %
des openssl	26 %	22 %	36 %	38 %	29 %	30 %	36 %	22 %	32 %	100 %	29 %	20 %	27 %	17 %	17 %
rsa cryptopp	12 %	9 %	48 %	43 %	39 %	36 %	72 %	14 %	11 %	9 %	9 %	6 %	23 %	4 %	4 %
rsa openssl	22 %	19 %	47 %	47 %	42 %	38 %	62 %	28 %	23 %	17 %	20 %	11 %	91 %	8 %	8 %
md5 beecrypt	45 %	100 %	50 %	56 %	74 %	41 %	58 %	26 %	38 %	35 %	73 %	35 %	47 %	33 %	33 %
md5 cryptopp	11 %	22 %	74 %	76 %	100 %	72 %	57 %	14 %	15 %	13 %	23 %	10 %	30 %	7 %	7 %
md5 openssl	34 %	66 %	49 %	53 %	71 %	41 %	55 %	25 %	37 %	41 %	100 %	27 %	45 %	26 %	26 %

Figure 6.4.: Results of the signature matching using (mnemonic, constant)-tuples

However, for the `chains()` method the performance is overall good, if we consider the unique-signatures. However, the chains-for-implementation and chains-for-algorithm signatures produce a higher rate of false-positives, for example the chains-for-algorithm matches for Beecrypt AES are 9%AES, 2% RSA, 8% DES, 10% RC4, and 10% MD5. Nevertheless, the chains-for-algorithm-unique signature only identifies AES for the Beecrypt AES application.

If we take a look at the performance of the (mnemonic, constant)-tuple matching method implemented in `constmnemonic()`, we can see it is the most successful of the signature identification methods. The details of the results are presented in Figure 6.4. There, the signatures are displayed on the y-axis and the testing applications are shown on the x-axis. Each highlighted field links the testing application to the respective signature. We can note that if we apply the threshold of 70%, most of the implementations are correctly identified. The XOR testing application naturally does not have (mnemonic, constant)-tuple signatures. Although, we can also note that they do not yield a false-positive, because their percentage is always below 70%.

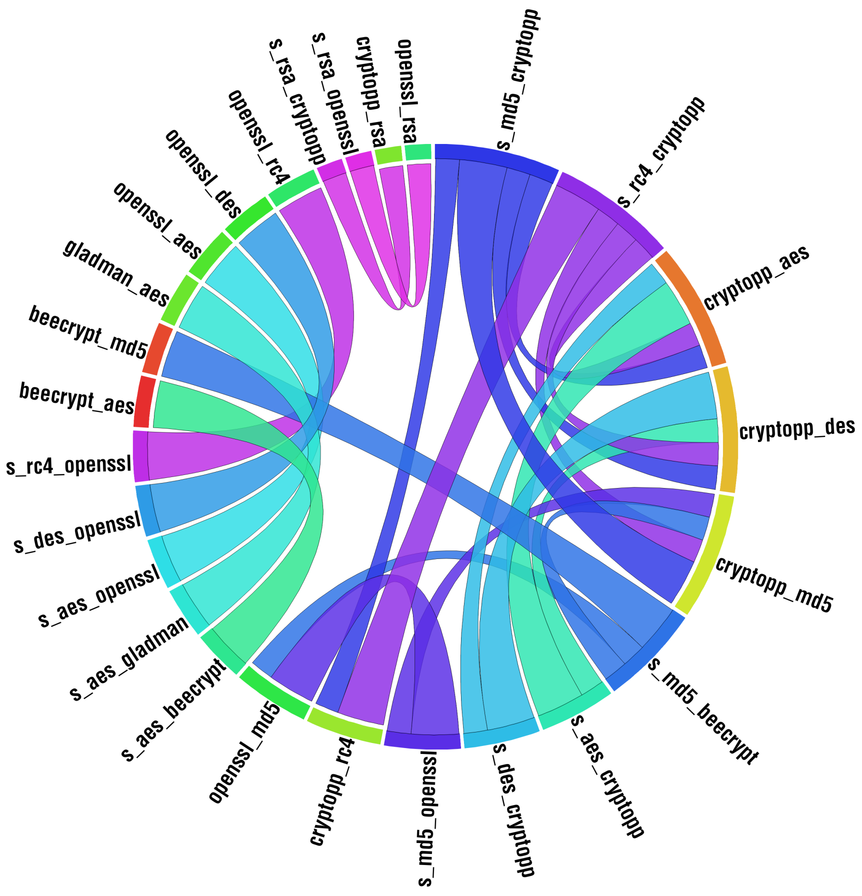


Figure 6.5.: Results of the implementation signatures

If we plot the x- and y-axis of Figure 6.4 around a circle and interconnect the axes if the threshold is above 70%, we get the image shown in Figure 6.5. Each part of the circle is annotated by a testing application, e.g., beecrypt_md5, or a signature, e.g., s_aes_-beecrypt. The signatures displayed in this image are the implementation signatures as described in Section 4.4.1 and Figure 4.4.

We can point out two aspects of Figure 6.5: the turquoise Beecrypt AES signature at 8 o'clock matches only the Beecrypt AES testing application, which is a positive result without false-positives. The s_md5_cryptopp signature is representing the implementation of MD5 by Cryptopp and is shown in blue at 12 o'clock. The signature matches the Cryptopp testing application of MD5, which is positive result, but also with a smaller connection line RC4, AES, and DES, which represent false-positive results, although the thickness of the lines indicate a matching value of less than 100%. For the intersect signatures, we can note that while some signatures, e.g., s_md5_intersect, have a good performance, others have a high rate of false-positives, e.g., s_aes_intersect. The intersect signatures and their performance are enhanced by the the unique signatures.

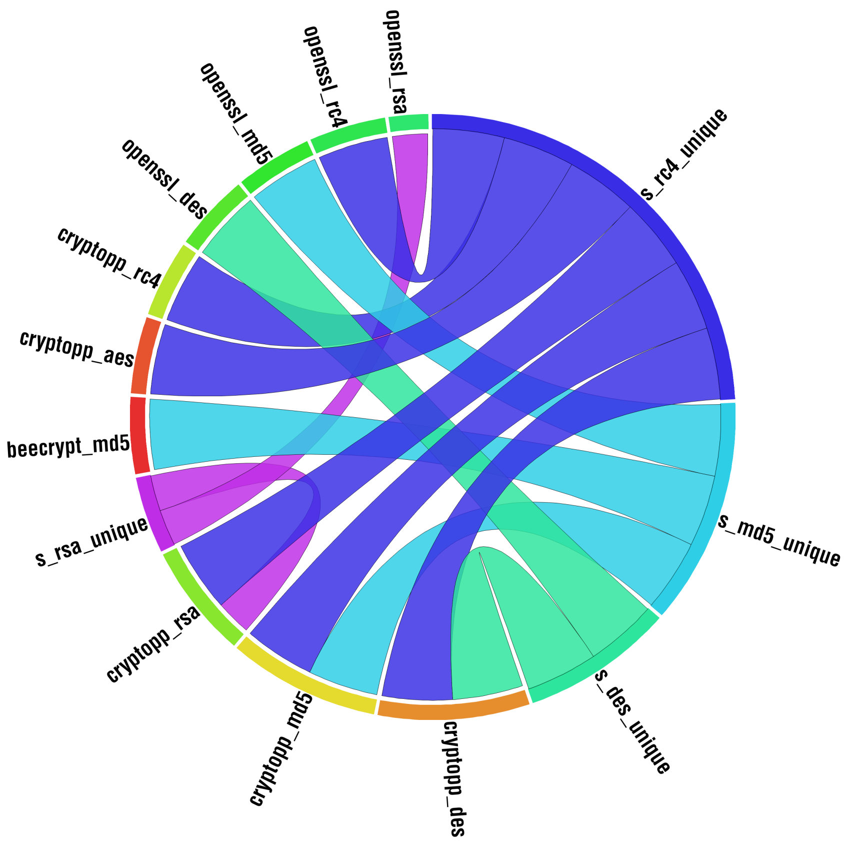


Figure 6.6.: Results of the unique signatures

The unique signatures are shown in Figure 6.6. At first we can recognize that the RC4 signature at 2 o'clock yields three false-positives. This is due to the small size of the RC4 unique signature with only two values, as described in Section 4.4.1. Nevertheless, the RC4 signature can be used to successfully identify the RC4 implementations of OpenSSL and Cryptopp. Furthermore, we can note that the RSA, DES, and MD5 signatures correctly identify their respective implementations without false-positives.

In summary, the signature identification methods work most successful if multiple characteristics are combined and compiled to a signature. A combination of all characteristics to a (mnemonic, disassembly constant, adjacent memory value)-tuple-sequences signature with a fuzzy matching algorithm could constitute a good method.

6.2.3. Generic Methods

At last, we evaluate the identification methods which also target the cryptographic parameters: `xorNotNullAndMov()`, `loopDiffer()`, and `symmetricCipherDataTester()`. In terms of computational requirements, the three methods are the most expensive.

Table 6.6 shows the performance of the generic identification methods. Empty cells indicate no results. At first, we would like to note that the `xorNotNullAndMov()` method successfully found the XOR encryption. Although it is only a single example of a XOR encryption, the method also identified the XOR operations in the cipher-block chaining (CBC) and cipher feedback mode (CFB) encryptions. There, the XOR operation is used to merge the initialization vector or encryption output with the plaintext. Therefore, we conclude that the `xorNotNullAndMov()` method is well suited to find XOR encryptions.

Secondly, we investigate the performance of the `loopDiffer()`. The method performs several searches: for counters, for xor relations, for entropy changes, and for the DES p-box. Because some of the searches are still experimental, we only evaluate the search for counters and the DES p-box. As shown in Table 6.6, the counter search successfully determined several counters in most of the implementations. Although some false-positive counters existed, for example when incrementing memory addresses in Listing 6.2, most found counters seem plausible.

```
[DEBUG]   upward counter from 27672868 to 27672894, distance 26
[DEBUG]   upward counter from 1 to 128, distance 127
```

Listing 6.2: Excerpt of found counters

The search for the DES p-box in the `loopDiffer()` is not as successful. It neither finds a p-box relation for the DES testing applications, nor does it find a false-positive result in the other testing applications. A reason for the missing p-box relation may be that the p-box is merged with the preliminary s-box in the DES implementation and therefore we do not encounter the intermediate value in the trace. If the p-box input value is not

Implementation	Algorithm	xorNotNull...	loopDiffer	symmetric...
Beecrypt	AES			success
Brian Gladman	AES	CBC XOR	only counters	no success
Cryptopp	AES	partial CFB XOR	only counters	success
OpenSSL	AES	partial CFB XOR	only counters	success
Cryptopp	DES	partial CFB XOR	only counters	success
OpenSSL	DES			success
Cryptopp	RC4		only counters	success
OpenSSL	RC4		only counters	success
Beecrypt	MD5		only counters	
Cryptopp	MD5		only counters	
OpenSSL	MD5		only counters	
Cryptopp	RSA		only counters	
OpenSSL	RSA		only counters	
Custom	XOR	success	only counters	

Table 6.6.: Analysis performance for the generic identification methods

calculated in the cryptographic implementation, we cannot verify the p-box relation due to the missing value.

The third generic identification method is the `symmetricCipherDataTester()`. This method is the most successful identification method, which also verifies the existence, and the parameters of a symmetric encryption. Table 6.6 shows that the method is able to detect nearly every instance of the symmetric encryption algorithms. The only undetected trace is Gladman's AES implementation. Naturally, the method does not yield false-positive results. The success of the `symmetricCipherDataTester()` method is closely bound to the memory reconstruction method described in Section 3.6. In case of the Gladman AES implementation, the memory reconstruction method is unable to reconstruct the cryptographic parameters. Thus, the `symmetricCipherDataTester()` method has no success. Although the memory reconstruction often leads up to 2000 candidates for encryption key, plaintext, and ciphertext each, the time for the candidate check is feasible. For AES, our non-optimized AES candidate check function is able to conduct 400,000 checks per second. In the extreme case of 2000 candidates for each parameter, the verification of all the candidates would only need $\frac{2000^2}{400000} = 10$ seconds.

6.3. Off-the-Shelf Application

To show the generic usage of our approaches, we tested our system implementation against off-the-shelf software. We traced and analyzed a SSL session of the Curl HTTP client. Curl is available on many systems and embedded as a HTTP client library in PHP for example. Curl itself utilizes the OpenSSL library for establishing a SSL connection.

In the testing environment we setup Curl version 7.19.7 with OpenSSL version 0.9.8l. Then, we generated the trace using the following command line options:

```
pin.exe -t kerckhoffr.dll -watch_thread 0 -mw libeay32.dll -istart 300000000  --  ↩
    curl.exe -k -s -o out.html https://www.ssllabs.com
```

Listing 6.3: Command line options for tracing Curl

Since we knew that the encryption is performed by OpenSSL, we only traced this DLL using `-mw libeay32.dll`. Also, we configured our Pintool to only watch the main thread with the switch `-watch_thread 0` and to start after 300 million instructions with `-istart 300000000`. We determined the thread and the instruction count using tests, which we made with our Pintool. Precisely, we first generated a complete trace, starting from instruction zero, with all threads and libraries included. In the complete trace of 13 GB, we found that the OpenSSL library is used in the last 2% of the trace. Thus, we could configure a smaller trace, as shown in Listing 6.3, which took 7 minutes and 38 seconds to finish and generated a trace file with 583,417 instructions (45 MB).

The Curl options instructed it to skip the X509 certificate verification (`-k`), to suppress console output (`-s`), and to save the retrieved HTML to a file (`-o out.html`). The retrieved HTML file was the starting page of `https://www.ssllabs.com`, although any other HTTPS site could have been used as well. The retrieved file contained 5136 bytes. We also recorded the runtime of the Curl executable using `procmon.exe` and analyzed its network communication with `tcpdump` and Wireshark. We observed that the remote SSL server and the Curl client negotiated the SLL cipher suite setting `TLS_DHE_RSA_WITH_AES_256_CBC_SHA`[2]. Thus, we knew that the analysis should at least detect the RSA and AES invocation. After the secure channel had been established, the selected cipher was used to encrypt three packets of SSL application data. Obviously, the first packet was the client HTTP request of 160 encrypted bytes, and then followed the server response with 272 bytes for the HTTP header and 5168 bytes of content.

The analysis tool took 2 hours 34 minutes to finish, although the `xorNotNullAndMov()` identification method was the most time consuming method with 2 hours 25 minutes runtime. The complete results of the analysis are summarized in Table 6.7. The `xorNotNullAndMov()` method was also the method with the highest rate of possible false-positives: of 90674 verified XOR relations found, we were not able to manually confirm a relation as part of the cipher-block chaining mode XOR. Therefore, we guess that the found XOR relations are part of the inner computation of the AES or Curl.

[2]The cipher suite specifies Diffie-Hellman Key Exchange, with RSA certificates, symmetric encrypted by AES in CBC mode with 256 bit keys, and integrity checked by SHA1.

Method	Results
`xorNotNullAndMov()`	only false-positives / unknown results
`symmetricCipherDataTester()`	detected 94% of AES instances including parameters
`loopDiffer()`	detected AES counters, some false-positives
`sigAPI()`	detected cryptographic functions
`constmemory()`	detected AES, one false-positive
`chains()`	detected AES and RSA, including implementation
`constmnemonic()`	detected AES implementation, one false-positive
`wang()`	no results
`caballero()`	detected core AES basic blocks
`lutz()`	detected core AES loops

Table 6.7.: Analysis performance for the Curl trace

The `symmetricCipherDataTester()` identification method performed very good. Of the 350 blocks of encrypted AES data, which we recorded using `tcpdump`, the identification method was able to find and verify the plaintext, key, and corresponding ciphertext of 331 blocks (success rate of 94.57%). Using the AES reference implementation the method checked whether 3395 candidate keys and 4205 candidate plaintexts correspond to one of 8037 candidate ciphertexts. The method took 46 minutes to finish, although the majority of time consumption is due to the memory reconstruction algorithm. The missed 5.43% of AES primitives were also caused by the memory reconstruction method, because the identification method only uses data from the reconstruction and verifies it using the reference implementation. Thus, the missing data has not been reconstructed and therefore could not be verified.

We manually compared the results of the identification method with the recorded `tcpdump` file. We confirmed that the memory-generated ciphertexts correspond to the network-generated `tcpdump` file. In order to decrypt the CBC mode[3], we wrote a script, which xors adjacent input/output blocks and thereby reveals the plaintext of the SSL application data. In Listing 6.4 we reveal the plaintext of the client request. We can clearly recognize the encoded end of the request by `\x03\x03\x03\x03` and the 20 byte SHA1 sum before that.

```
position 48:144 = '7.19.7 OpenSSL/0.9.8l zlib/1.2.3\r\n'+\
'Host: www.ssllabs.com\r\n'+\
'Accept: */*\r\n'+\
'\r\n'+\
'\x9f\xda*\x1d\xca\xc1\x0eEh!\n\x1a\x1d\xed_K\xc3&#\xef'+\
'\x03\x03\x03\x03'
```

Listing 6.4: Decrypted HTTPS client request

The `loopDiffer()` method detected 427 counters of which 345 were an upward counter

[3]For an illustration of CBC mode refer to Figure 2.1.

from 1 to 15. Since AES-256 is executed with 14 rounds, the counters may relate to the 350 AES computations. The permutation box test of the `loopDiffer()` showed no results because DES was not used by Curl.

The signature-based identification methods revealed similar results as in the previous section. Since the OpenSSL library was build with debug symbols `sigAPI()` detected the following functions in Listing 6.5:

```
AES_set_encrypt_key
RSA_PKCS1_SSLeay
AES_cbc_encrypt
AES_decrypt
RSA_free
AES_encrypt
EVP_aes_192_ecb
MD5_Update
EVP_aes_256_cfb8
AES_set_decrypt_key
```

Listing 6.5: Detected function symbols in Curl

The `constmemory()` method, which searches for cryptographic constants in memory, detected several AES constants and concluded with a 56% match for AES (2560 constants in signature). Yet, it also found MD5 constants with 33% (91 constants in signature) and DES with 57% (186 constants in signature). This result shows that either the constants have to be further extended or the length of the signature should be incorporated into the match.

The `chains()` method, which compares mnemonic sequences, determined many false-positives for the non-unique signatures (for implementation and for algorithm). However, the unique signatures detected both AES and RSA without false-positives. Also, the correct implementation was detected by the implementation-unique signatures without a false-positive match.

An interesting result was revealed by the signature-based identification method `constmnemonic()`. Since we were not able to generate an unique or intersecting set for the AES algorithm, as described in Section 4.4.1, we only had the implementation signature for OpenSSL AES to match the trace. Among the implementation signatures, the OpenSSL AES signature had a relatively low match of 49%, compared to the previous section. Nevertheless, other implementation signatures followed at about 20-30% and OpenSSL AES still stood out among them. The intersect and unique signatures (available only for DES, RSA, MD5) detected one high false-positive (intersecting DES with 56%) and some lower false-positives around 35%.

The previously published identification methods by Lutz, Caballero et al., and Wang et al. also performed similar to Section 6.2. Wang's method generated no results, probably due to the fact that the trace did not start at the beginning of the application. Caballero's method successfully detected 19 basic blocks in the encryption and key scheduling functions `AES_decrypt`, `AES_encrypt`, `AES_set_decrypt_key`, and `AES_set_encrypt_key`. The method of Lutz, without taint-tracking, revealed 2121 entropy changes in 26 loop bodies corresponding to 22 functions, for example in the AES encryption and decryption functions, in the `SHA1_Update` function, but also in false-positive functions like `OBJ_NAME_do_all_sorted`, `ASN1_OBJECT_it`, or `OPENSSL_cleanse`.

6.4. Modified Application

In order to test the identification performance against binary modification, e.g., anti-analysis and obfuscation, we packed a testing application and analyzed it using our system. The used packer was ASPack in version 2.12 and the testing application was the XOR application with an input/output of 4096 bytes. Although the trace size increased by factor 17 and the analysis took longer, but the analysis tool in Listing 6.6 was still able to identify all blocks of XOR encrypted text:

```
[DEBUG] assumption holds for 100 % of values
[DEBUG] xor key = 'u \xa6\xd1^\x1b\x19\xaa\x' ...
[DEBUG] xor plaintext = 'DDDD3333DDDD3333DDDD33' ...
[DEBUG] xor ciphertext = '1\xe2\x95\x1a(*\x99\xf5' ...
$ xxd io/rand.0128
0000000: 75a6 d15e 1b19 aac6 49be 6b2c 80d4 f49e u..^....I.k,....
```

Listing 6.6: Sample output for the XOR check

Interestingly, the packer introduced 24 new loops, but the loop analysis was still able to point out the original XOR encryption loop, which was also found in the original testing application. The packed loop still had 32 executions, with 128 iterations each, to encrypt the totaling 4096 bytes. The control flow graph for the loop is shown in Figure 6.7 for comparison with Figure 3.1.

6.5. Summary

In the evaluation we tested our methods and their implementation against several applications. Where applicable, we detailed advantages and drawbacks of methods or their implementation. We elaborate on the evaluation's conclusion in the next chapter.

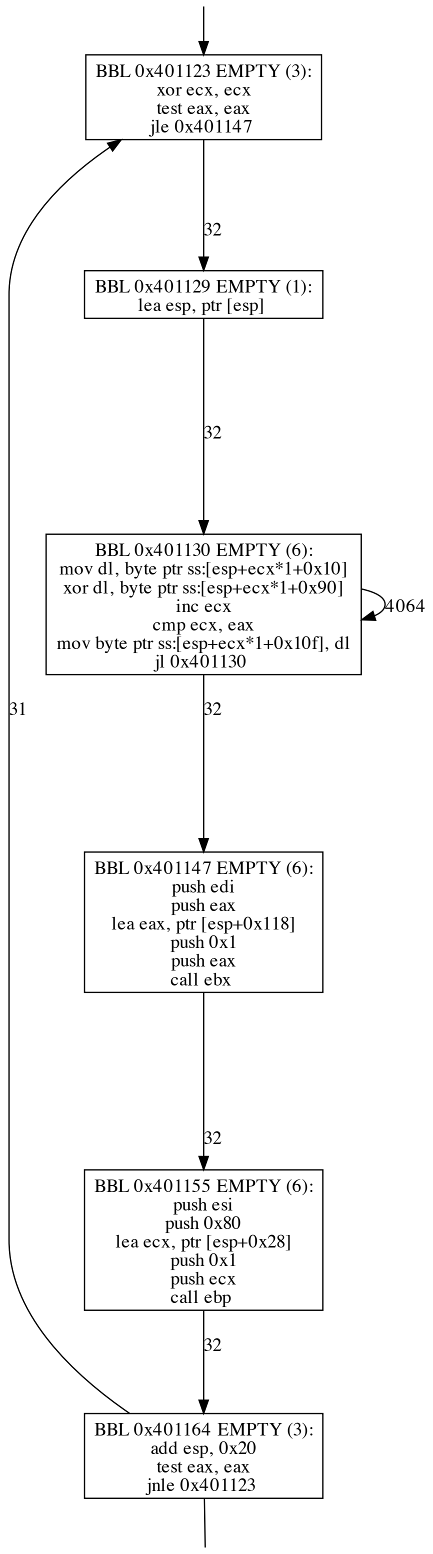


Figure 6.7.: The core XOR encryption packed with ASPack

Chapter 7. Conclusions

In this chapter we draw the conclusions from the evaluation and the previous chapters. We also point out shortcomings of the current solution and describe means to improve it.

7.1. Evaluation Performance

In the previous chapter we evaluated our identification methods and their implementation against multiple programs. The used testing environment is based on consumer computing hardware and our system implementation only depends on free and open-source software.

The 14 testing applications were successfully traced using our Pintool in a short time frame. Also, the analysis tool was able to perform the identification methods in an acceptable time frame. The identification method by Caballero et al. successfully found cryptographic basic blocks in 92% of the testing applications. This shows that a simple heuristic based on bitwise arithmetic instructions can yield a good identification method. The method by Wang et al. provides an interesting measurement for finding the turning point of a single encryption in a complete trace. Yet, it was only successful in 28% of the testing applications. Although we did not completely implement the taint-tracking of Lutz's method, we evaluated his entropy-based method. We found that for 57% of the testing applications the method was fully or partially successful. In 28% of the tests it generated false-positives and in 14% it returned no results.

Our signature-based methods provided good results for the testing applications. If given, the debug symbols were identified by the `sigAPI()` method in 100% of the cases. The (constant, mnemonic)-tuple method generated successful results for 100% of the cryptographic applications. Also the mnemonic-sequence method found 100% of the cryptographic algorithms and implementations, although it returned false-positives in one case. The most ineffective signature-based identification method was the memory-constant method. It only had success in 21% of the testing applications and generated one false-positive, because not all algorithms can uniquely identified by only constants, and therefore a combination of characteristics yields the best results.

The generic loop differ method provided only moderate results. Although it detected counters for most of the testing applications, its other tests for entropy, XOR, and permutation boxes are still too experimental. The `xorNotNullAndMov()` method was successful for the XOR testing application and also provided satisfactory results for the CBC and CFB mode block ciphers. However, the best performing generic identification method was the `symmetricCipherDataTester()`, which was able to identify the cryptographic algorithm, plaintext, key, and corresponding ciphertext in 87% of the symmetric cipher testing applications.

The Curl HTTPS evaluation was also successful, because we were able to discover the actually used algorithms and their implementation including plaintexts and ciphertexts of the secure channel. Furthermore, we uncovered the SSL session keys used for the encryption and decryption of application data, using the `symmetricCipherDataTester()` method.

On an experimental basis, the packing of the XOR application with ASPack showed that the analysis algorithms presented in Chapter 3 are robust and stable. Also the tracing and analysis implementation generated successful results for the packed testing application.

The evaluation shows that the most successful identification methods are the `constmnemonic()` and the `symmetricCipherDataTester()` methods. For special algorithms, like XOR, other methods are more suitable, but generally a combination of a signature-based and a generic approach works well. We were able to identify the majority of cryptographic algorithms, including their parameters, and thereby prove the thesis which is stated in Section 1.2.

7.2. Further Work

With the results of the evaluation, we see two areas of further work: (a) the research for other identification methods and (b) the improvement of the existing system, enhancing the most successful identification methods from above.

The most promising method for the identification of cryptographic keys is the `symme-`

`tricCipherDataTester()` method based on Observation 4. Further research is necessary to optimize the generation and filtering of candidate keys from a trace. The optimization could utilize known cryptographic code and its API. Thereby we could enhance the extraction of the cryptographic parameters due to the known positioning. Also, the methods from Halderman et al. [32] should be further investigated and combined with the methods of Observation 4. Another challenging task is the development of an identification method, which is able to generically identify custom cryptographic code, for example the custom Feistel network found in malware by Greulich [29]. This generic discovery of cryptographic parameters without a reference implementation is an open problem.

The signature-based identification methods could be enhanced to allow fuzzy matching. For example, we could tolerate wrong sample mnemonics in a mnemonic sequence signature to a certain level. We could use abstract interpretation, either in the native or an intermediate language, in combination with symbolic execution to extract relations from the executed code and thereby determine parts of a cryptographic algorithm.

Depending on the application of the analysis, methods could be developed to identify the padding, encoding, and compression of the plaintext. Also, a sound analysis tool may be used to recompose the cryptographic protocol flow and graph the cryptographic composition. Therefore, an identification of the mode of operation must be done, although this seems trivial given all input/output parameters of all blocks.

To develop an efficient system implementation, we need to reconsider the design decision of separating tracing and analysis. A better interconnection between tracing and analysis would at least discard the overhead of the trace sizes. Using a solution with named pipes, i.e., writing the trace directly to the analysis tool's memory, may be a good start. On the other hand, the syntax and thereby the size of the trace files can be easily enhanced. The Nirvana/iDNA paper [7] points to several methods for reducing the trace size. Instead of an ASCII trace, we could write a binary trace, which would radically decrease the trace size. Also, the trace should only contain changing values, e.g., we should only log the current thread ID if it has changed.

Depending on the type of software to be analyzed, we also need to reconsider other tracing frameworks from Chapter 2. In general, the tracing should be more flexible and further options could be developed to filter the tracing target. For example, a trigger instruction, e.g., int3, could be used to start the trace. The attachment of the instrumentation to a running target could be also supported. A restriction of the instrumentation to certain memory address ranges may also yield better, filtered results. In the analysis tool, the reporting needs to be enhanced and abstracted to better distinguish between debugging and result output.

7.3. Conclusion

In this thesis we presented several methods to identify cryptographic code in arbitrary programs. We pointed out the relatively small amount of related work in this area and evaluated available signature-based static tools. To underpin our dynamic approach, we described several software analysis methods, including control flow graph generation and loop detection. We characterized cryptographic implementations and observed dedicated attributes, on which we developed and implemented our identification methods. The implemented system was evaluated and we concluded with possible improvements.

In 2009, the research community has reported several flaws in a variety of software solutions. The flaws include wrong cryptographic compositions, static-key protocols, and fraudulent cryptographic implementations. We show that an automated analysis for cryptographic software usage is possible and we believe that the interest in security analysis of cryptographic code will increase.

Bibliography

[1] P. Bächer, T. Holz, M. Kötter, and G. Wicherski. Know your enemy: Tracking botnets. *The Honeynet Project and Research Alliance, March*, 2005.

[2] P. Bania. Generic Unpacking of Self-modifying, Aggressive, Packed Binary Programs. 2009.

[3] Y. Bao, M. Chen, Y. Ruan, L. Liu, J. Fan, Q. Yuan, B. Song, and J. Xu. HMTT: A Platform Independent Full-System Memory Trace Monitoring System. In Z. Liu, V. Misra, and P. J. Shenoy, editors, *SIGMETRICS*, pages 229–240. ACM, 2008. ISBN 978-1-60558-005-0.

[4] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 2006.

[5] P. Beaucamps and E. Filiol. On the possibility of practically obfuscating programs Towards a unified perspective of code protection. *Journal in Computer Virology*, 3 (1):3–21, 2007.

[6] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46. USENIX, 2005.

[7] S. Bhansali, W. Chen, S. De Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments*, page 163. ACM, 2006.

[8] L. Böhne. Pandora's Bochs: Automatic Unpacking of Malware. Master's thesis, RWTH Aachen University, 2008.

[9] F. Boldewin. Analyzing MSOffice malware with OfficeMalScanner. Technical report, reconstructer.org, 2009.

[10] P. P. Bungale and C.-K. Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. In C. Krintz, S. Hand, and D. Tarditi, edi-

tors, *Proceedings of the 3rd international conference on Virtual execution environments*, pages 137–147. ACM, 2007. ISBN 978-1-59593-630-1.

[11] J. Caballero, H. Yin, Z. Liang, and D. X. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pages 317–329, 2007.

[12] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Bidirectional Protocol Reverse Engineering: Message Format Extraction and Field Semantics Inference. Technical report, University of California at Berkeley, 2009.

[13] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling Active Botnet Infiltration using Automatic Protocol Reverse-Engineering. In *ACM Conference on Computer and Communications Security*, 2009.

[14] E. Carrera. Bochs python. *LaCon Proceedings*, 2008.

[15] L. Cavallaro, P. Saxena, and R. Sekar. Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. Technical report, Stony Brook University, 2007.

[16] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security Symposium*, pages 321–336. USENIX, 2004.

[17] J. A. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In D. S. Rosenblum and S. G. Elbaum, editors, *Proceedings of the 2007 international Symposium on Software Testing and Analysis*, pages 196–206. ACM, 2007. ISBN 978-1-59593-734-6.

[18] W. Cui, M. Peinado, K. Chen, H. Wang, and L. Irun-Briz. Tupni: Automatic Reverse Engineering of Input Formats. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 391–402. ACM New York, NY, USA, 2008.

[19] J. Daemen and V. Rijmen. AES proposal: Rijndael. *Proceedings from the First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, 1998.

[20] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In P. Ning, P. F. Syverson, and S. Jha, editors, *ACM Conference on Computer and Communications Security*, pages 51–62. ACM, 2008. ISBN 978-1-59593-810-7.

[21] D. Dittrich and S. Dietrich. P2P as botnet command and control: a deeper insight. In *Proceedings of the 3rd International Conference on Malicious and Unwanted Software*, pages 46–63, 2008.

[22] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. X. Song. Dynamic Spyware Analysis. In *USENIX Annual Technical Conference*, pages 233–246. USENIX, 2007.

[23] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull. Graphviz-open source graph drawing tools. *Lecture Notes in Computer Science*, pages 483–484, 2002.

[24] P. Ferrie. Attacks on virtual machine emulators. *Symantec Security Response*, 2006.

[25] P. Ferrie. Anti-unpacker tricks. In *Proceedings of the 2nd International CARO Workshop*, 2008.

[26] H. Flake. More fun with graphs. *Proceedings of BlackHat Federal*, 2003.

[27] J. Franklin, M. Luk, and J. M. McCune. Detecting the Presence of a VMM through Side-Effect Analysis. Technical report, Carnegie Mellon University, 2005.

[28] A. Godiyal, A. Nguyen, and N. Schear. A Lightweight Hypervisor for Malware Analysis. Technical report, University of Illinois at Urbana-Champaign, 2008.

[29] A. Greulich. Mebroot / Torpig / Sinowal. *InBot*, 2009.

[30] I. Guilfanov. Fast Library Identification and Recognition Technology, 1997. URL http://www.hex-rays.com/idapro/flirt.htm.

[31] W. Guizani, J. Marion, and D. Reynaud-Plantey. Server-Side Dynamic Code Analysis. In *4th International Conference on Malicious and Unwanted Software*, pages 55–62, 2009.

[32] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *USENIX Security Symposium*, pages 45–60, 2008.

[33] N. Heninger and H. Shacham. Reconstructing RSA private keys from random key bits. In *Advances in Cryptology (CRYPTO)*, 2009.

[34] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling. Measurements and Mitigation of Peer-to-Peer-based Botnets: A Case Study on Storm Worm. In *First USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

[35] D. Janssens. *Heuristic methods for Locating Cryptographic Keys Inside Computer Systems*. PhD thesis, Katholieke Universiteit Leuven, 1999.

[36] D. Janssens, R. Bjones, and J. Claessens. KeyGrab TOO - The search for keys continues...,. *Whitepaper, Utimaco Safeware AG, KU Leuven*, 2000.

[37] M. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, page 53. ACM, 2007.

[38] K. Kaukonen and R. Thayer. A Stream Cipher Encryption Algorithm ”Arcfour”. *The Internet Society*, 1999.

[39] A. Kerckhoffs. La Cryptographie Militaire. *Journal des sciences militaires*, 1883.

[40] T. Klein. All your private keys are belong to us. Technical report, 2006.

[41] J. Kong, C. Zou, and H. Zhou. Improving Software Security via Runtime Instruction-Level Taint Checking. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 18–24. ACM New York, NY, USA, 2006.

[42] C. Kruegel, D. Balzarotti, W. K. Robertson, and G. Vigna. Improving Signature Testing through Dynamic Data Flow Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 53–63. IEEE Computer Society, 2007.

[43] A. Lanzi, M. I. Sharif, and W. Lee. K-Tracer: A System for Extracting Kernel Malware Behavior. *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[44] F. Leder and T. Werner. Know your enemy: Containing conficker - to tame a malware. *Know Your Enemy Series of the Honeynet Project*, 2009.

[45] K. Lejska. X86 Opcode and Instruction Reference. URL http://ref.x86asm.net/.

[46] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 121–141, 1979.

[47] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[48] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200. ACM New York, NY, USA, 2005.

[49] N. Lutz. Towards Revealing Attackers' Intent by Automatically Decrypting Network Traffic. Master's thesis, ETH Zürich, 2008.

[50] C. Maartmann-Moe, S. Thorkildsen, and A. Årnes. The persistence of memory: Forensic identification and extraction of cryptographic keys. *Digital Investigation*, 6:132–140, 2009.

[51] S. McCamant and M. Ernst. Quantitative Information-Flow Tracking for C and Related Languages. *Computer Science and Artificial Intelligence Laboratory Technical Report, MIT, MIT-CSAILTR-2006-076*, 2006.

[52] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[53] J. Murakami. A Hypervisor IPS based on Hardware assisted Virtualization Technology. *Black Hat USA*, 2008.

[54] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, 2007.

[55] R. A. Nguyen Anh Quynh, Kuniyasu Suzaki. eKimono: a Malware Scanner for Virtual Machines. *Hack In The Box Kuala Lumpur*, 2009.

[56] NIST. FIPS 46-3: Data Encryption Standard. 1977.

[57] T. Ormandy. An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. *CanSecWest*, 2007.

[58] R. Paleari, L. Martignoni, G. Roglia, and D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *Proceedings of 3rd USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[59] M. Payer. secuBT: Hacking the Hackers with User-Space Virtualization. In *Proceedings of the 26c3*, pages 157–163, 2009.

[60] P. Porras, H. Saidi, and V. Yegneswaran. Conficker C P2P Protocol and Implementation. Technical report, SRI International, 2009.

[61] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 15–27, 2006.

[62] D. Quist and Valsmith. Covert Debugging Circumventing Software Armoring Techniques. *Black Hat Briefings USA*, 2007.

[63] T. Raffetseder, C. Krügel, and E. Kirda. Detecting System Emulators. *Lecture Notes in Computer Science*, 4779:1–18, 2007.

[64] E. S. Raymond. If Cisco ignored Kerckhoffs's Law, users will pay the price, 2004. URL `http://lwn.net/Articles/85958/`.

[65] D. Reynaud. A Survey on Virtual Machines for Malware Analysis. *3rd International Workshop on the Theory of Computer Viruses*, 2008.

[66] D. Reynaud. A look at anti-virtualization in malware samples, 2009. URL `http://indefinitestudies.org/2009/09/21/a-look-at-anti-virtualization-in-malware-samples/`.

[67] D. Reynaud. Large bunch of pin.log files, 2009. URL `http://tech.groups.yahoo.com/group/pinheads/message/4153`.

[68] R. Rivest. RFC1321: The MD5 message-digest algorithm. *Request for Comments*, 1992.

[69] R. Rivest, A. Shamir, and L. Adleman. Cryptographic communications system and method (RSA), 1983. US Patent 4,405,829.

[70] H. Röck. Survey of Dynamic Instrumentation of Operating Systems. Technical report, University of Salzburg, 2007.

[71] P. Royal and I. Damballa. Alternative Medicine: The Malware Analyst's Blue Pill. *Black Hat USA*, 2008.

[72] P. Saxena, R. Sekar, and V. Puranik. Efficient Fine-Grained Binary Instrumentation with Applications to Taint-Tracking. *Proc. of the 2008 International Sym. on Code Generation and Optimization (CGO)*, 2008.

[73] A. Shamir and N. Van Someren. Playing Hide and Seek with Stored Keys. *Lecture Notes in Computer Science*, 1999.

[74] M. Sheldon, G. Weissman, and V. Inc. ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. In *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2007.

[75] D. Stevens. XORSearch, 2009. URL http://blog.didierstevens.com/programs/xorsearch/.

[76] J. Stewart. Inside the Storm: Protocols and Encryption of the Storm Botnet. *Black Hat USA*, 2008.

[77] J. Tubella and A. González. Control Speculation in Multithreaded Processors through Dynamic Loop Detection. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, 1998.

[78] A. Vasudevan and R. Yerraballi. SPiKE: Engineering Malware Analysis Tools using Unobtrusive Binary-Instrumentation. *Proceedings of the 29th Australasian Computer Science Conference*, 2006.

[79] O. Vermaas and D. de Graaf. Operation ShadowBot: Case study in a botnet investigation. *InBot*, 2009.

[80] G. Vigna. Static Disassembly and Code Analysis. *Malware Detection*, 2006.

[81] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2007.

[82] Z. Wang, X. Jiang, W. Cui, and X. Wang. ReFormat: Automatic Reverse Engineering of Encrypted Messages. Technical report, NC State University, 2008.

[83] T. Werner and F. Leder. Waledac Isn't Good Either! *InBot*, 2009.

[84] C. Willems, T. Holz, and F. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security & Privacy*, 5(2):32–39, 2007.

[85] R. Wojtczuk. Subverting the Xen hypervisor. *Black Hat USA*, 2008.

[86] G. Wondracek, P. Comparetti, C. Kruegel, E. Kirda, and S. Anna. Automatic Network Protocol Analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[87] Y. Wu, T. Chiueh, and C. Zhao. Efficient and Automatic Instrumentation for Packed Binaries. In *Proceedings of the 3rd International Conference and Workshops on Advances in Information Security and Assurance*, page 316. Springer, 2009.

[88] H. Yin, D. X. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. *Proceedings of the 2007 ACM Conference on Computer and Communications Security (CCS)*, pages 116–127, 2007.

[89] A. Young and M. Yung. Cryptovirology: Extortion-Based Security Threats and Countermeasures. In *IEEE Symposium on Security and Privacy*, pages 129–141, 1996.

[90] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W. Wong. How to do a million watchpoints: Efficient Debugging using Dynamic Instrumentation. *Lecture Notes in Computer Science*, 4959:147, 2008.

Appendix A. Digital Attachments

The contents of the attached DVD are the following:

- The Python analysis tool `kerckhoffr/`
- The Pintool source, and compiled its compiled form `kerckhoffr.dll`
- The current Pin release
- The compiled testing applications with source
- This thesis including graphics and bibliography
- Gathered traces of the testing applications
- Generated control flow graph PDFs of the testing applications
- Papers from the bibliography
- Slides of intermediate talks on the progress of the thesis
- The static tools from Appendix B

Appendix B. Static Tools

All URLs and SHA1 sums were last checked September 19, 2009.

Krypto Analyzer (KANAL) Version 2.92
`http://www.peid.info/files/PEiD-0.95-20081103.zip`
`f04ebf471ebcf1598949af5a1aff11279e41da70 PEiD-0.95-20081103.zip`

Findcrypt plugin Version 2
`http://www.hexblog.com/ida_pro/files/findcrypt2.zip`
`361db4090396a24599bff9a2254dd5b7c2dd2f22 findcrypt2.zip`

SnD Crypto Scanner Version 0.5b
`http://www.tuts4you.com/request.php?2222`
`5fbcbf1739f1b167c5941084e4fd3716efbc696c CryptoScanner 0.5b.rar`

Crypto Searcher by x3chun Version 2004.05.19
`http://quequero.org/uicwiki/images/Cryptosearcher_2004_05_19.zip`
`735627196a3e573134b69f75351a0bca036b8014 cryptosearcher.exe`

Hash & Crypto Detector (HCD) Version 1.1
`http://www.woodmann.com/collaborative/tools/images/Bin_Hash_%26_Crypto_Detector_2009-2-8_1.0_Hash_%26_Crypto_Detector_v1.1.rar`
`098515422fed76ccd1a5c5e85413c6ae2baf1fdc HCD.rar`

DRACA Version 0.5.7b
`http://www.literatecode.com/get/draca.zip`
`6f32e1e6207c136240a5de9ea3857e4a3c9dae99 draca.zip`