Towards Auditing of Control-Flow-Integrity

Royal Holloway

Luke Atherton (100905113)

1st March 2019

Abstract

This is my abstract

Dedication

This is my dedication

Declaration

This is my declaration

Acknowledgements

Thanks everyone!

Contents

Chapter 1

Introduction

1.1 Control Flow

Control flow is how instructions within software transition from one to another.

1.1.1 Basic Blocks

Basic blocks is a useful way of breaking down the instructions of an application into manageable chunks. A basic block is a set of instructions which sequentially run from beginning to end. For example from the following pseudo-code:

```
void GreaterThanPlusOne(int firstNum, int secondNum)
    {
        string message;
        firstNum++;
        if (firstNum > secondNum)
        {
            message = "firstNum plus one is greater than secondNum";
        }
        else
        {
            message = "firstNum plus one is nto greater than secondNum"
        }
    }
```

The instructions from `string message;` to and including `if (firstNum > secondNum)` will sequentially (barring an exception being thrown). This is a basic block. The contents within the `if` and also the `else` parenthesis are also basic blocks.

So in this example the sequence of instructions can flow from the first basic block to either the second or the third basic block. An unauthorised flow would be from BB 2 to BB 3.

Called functions can also be a separator creating basic blocks, as they can be called from multiple places and return to multiple places.

1.1.2 Control-flow graphs

Control-flow graphs (CFGs) are the graphical representation of the series of valid basic blocks within an application. It is logical that a CFG can become enourmous as the size of an application grows (need to find reference for this). So essentially a valid control flow is one which follows a CFG for a given application.

1.1.3 Types of branches

An important aspect to consider is the difference between branch instruction types: direct, indirect and unintended.

Direct

A direct branch is one where the jump to instruction is hard coded - the control can flow to a pre-defined location.

This can be violated by a hardware attack such as glitching to skip instructions or from an attack directly on memory where the destination address is swapped for another.

Indirect

An indirect branch is one where the next instruction address is set at run-time, either through registers or pointers pointing to addresses in memory containing the jump destination. This could be the result of a compiled switch-case statement or dynamic libraries (e.g. reflection). In the case of the switch-case statement it is not so indirect in the source code, however when compiled has the possibility of indirectness.

In what common ways are these violated?

Unintended

De Clerq [1] describes an indirect branch as one which arises due to variable length instruction encoding in architectures. An attacker alters control flow to the middle of an instruction. A high percentage (80 %) of gadgets exploit this, according to Kayaalp [2]. For more information reference section 4.2 of Kayaalp's work [2] which describes unintended branches in detail. Does this affect embedded systems architectures?

Function Return

When a function returns to its calling section of code it references the stack for its return address. The interesting part here is that it is not hard coded and according to a CFG a function could legally return to any locations in code which call that function. Tracking a path taken along the CFG allows the checking of calling location, to make sure the function has returned to the correct caller.

If the stack is attacked the return address can be re-written to point to existing code, or instructions placed in data (this is usually protected for by read XOR write). This is a well-explored area in computer security with various existing solutions including stack canaries.

1.1.4 Loops

Loops are problematic in terms of CFG. Modelling a loop which could run indefinitely is not feasible with CFGs. Various methods has been used to approach this, with some simply compacting loops (this presents issues where an attacker continually run a loop until they reach the desired outcome, such as PIN attempts. Although would this not be fixed by control flow checks within the loop?) while others [3] track loop metadata such as iteration count.

1.2 Introduction

1.3 Attacks on hardware - taken from Smart Card Attacks

1.3.1 Introduction

1.3.2 Invasive Attacks

Page 196. Invasive attacks: microprocessor removed, and attacked directly through physical methods. In theory any microprocessor can be attacked in this way. Requires expensive equipment and large investment in time. Examples of attack: Probing bus lines between blocks on a chip (with a hole being made in a chip's passivation layer). Secret information is derived by observing information sent from one block on to another. Extreme example: Use focused ion beam to destroy or create tracks on the chip's surface. This could be used to reconnect disconnected fuses (think fuse used to deactivate PUF derivation). Use of fuses can also be to turn off test mode which is used to read/write to memory addresses during manufacture. This vulnerabiltiy has now been removed as test circuit is axtually removed from when the chip is cut from the die. [4] [5]

1.3.3 Semi-Invasive Attacks

Semi-invasive attacks: surface of chip needs to be exposed, security is compromised without directly modifying the chip. Examples: Observing electo-magnetic eminations using a suitable probe [6], [7], injecting faults using laser [8] or white light [9]. Numerous more [10].

Fault Injection

Variations in supply voltage [4],[11]: may cause processor to misinterpret or skip instructions. Variations in external clock [4],[12],[5]: Data can be misread (data is attempted to be read before memory has time to latch-out correct value). Instruction miss. Extremes of temperature [13],[14]: unpredictable effects in microprocessor. Two effects obtained [8]: random modification of RAM cells due to overheating, read and write temperature thresholds in most NVM do not coincide. If temperature is set to level where write ops work by read do not a number of attacks can be mounted. Laser light (French!!(15), [15], Frechn!!(39)): Light arriving on metal surface induces a current, if intense enough could induce fault in a circuit. White light [4]: Proposed as alternative to laser [9], but not directional so may be a challenge to apply to particular portions of microprocessor. Electromagnetic flux [16]: change values in RAM, strong eddy currents can affect microprocceors - only observed in insecure microprocessors.

Effects: Reset data: force data to blank state Data randomisation: Change data to new random value. Modifying opcodes: Change intstructions executed on chip's cpu [4]. Often same effect as previous effects. Additionally removal of functions and breaking of loops.

Countermeasures given in [8]

1.3.4 Non-Invasive Attacks

Non-invasive attacks: Derive information without modification. Derive information through information that leaks during computation of given command, or attempt to inject faults in manner other than light. Examples: Observe power consumption [17], [18], inject faults by glitching power supply [4], [8]

1.4 Software attacks

1.4.1 Buffer overflow

Target of attack is manipulating control-flow information stored on the stack and heap of a program in order to achieve further objectives.

1.4.2 Injected code

Where control flow is deviated to existing injected code (usually as data) - usually solved by using NX bit which marks data memory as non-executable.

1.4.3 Code re-use attack

Where control flow is deviated to existing code such as system functions or unintended code sequences (refactor last phrase).

1.4.4 Return Oriented Programming (ROP)

An attacker can string together (benign) existing code sequences (gadgets) to form malicious program actions.

1.4.5 jump-to-libc

1.4.6 return-to-libc

1.4.7 Pointer subterfuge

1.4.8 Non-control data attacks

Corrupting data used to decide on control flow for example in a comparison in an if statement. These usually produce unintended yet valid program flow, however there are examples which do not induce unintended flows - as discussed by Shacham [19].

1.5 Introduction

1.6 Requirements

1. Works with compiled binaries
2. Works with external libraries
3. Can be used to bind software and hardware
4. Works with embedded systems
5. Immediate identification
6. Not reliant of TPM (TEE)

1.7 Properties

1. Hardware based
2. Software based
3. Hardware modification required
4. Source code modification required
5. Granularity
6. Provisioning method

7. How it is secured?
8. CPU Overhead
9. Storage overhead
10. Memory overhead
11. Execution time overhead
12. Compatible architectures
13. Action after identification of insecure event
14. Prevention / Detection / Attestation
15. CFG or other

1.8 Solutions

1. C-FLAT [20]
2. LO-FAT [3]
3. CEP-LEE [21]
4. CCFI-Cache [22]
5. HCFI [23]
6. Hafix [24]
7. SOFIA [25]
8. Sullivan [26]

1.9 Conclusion

Chapter 2

Control Flow Integrity

2.1 Introduction

2.2 Control Flow Integrity

2.2.1 Introduction

Control-Flow Integrity (CFI) security policy states that software execution must follow a path of a Control-Flow Graph (CFG).

2.2.2 Control Flow Graphs

Abadi et al. [27]introduced CFG as a method to protect code, using static analysis of a application binary to create Control Flow Graph (CFG). Only control flow transfers within the CFG are permitted.

Clerq and Verbauwhede [1] posed CFGs as a solution for instructions causing control flow transfers. Lee et al. [21] note that they do not focus on sequential transitions. They argue that the method used in [1] (where forward edges are described as control flow transfers caused by jumps and calls and backward edges are described as those caused by returns) is disadvantageous as by considering all jumps and calls equally there is a loss in distinction between jumps to register-determined and instruction-determined locations, and they state that when implementing a scheme instruction-dependant transitions are simpler to process than instruction-independent transitions. This is described in reference to [28]. What are sequential transitions vs control flow transfers?

2.2.3 Others?

[27] Describes jump labelling.
[1] Describes Shadow Call Stacks
[25] Describes SOFIA

2.2.4 Introduction to CFGs

[27] states that CFGs can be defined by analysis-type methods or explicit policies. Examples of analysis-type methods include source-code analysis, binary, analysis binary analysis or execution profiling. An example of an explicit security policy method is writing as security automatat [29].

CFGs have been used to provide protection against soft faults (single-event upsets) using software based methods [30] [31] [32]. [27] restricts control flow through inlined labels and checks. The CFG is embedded through a set of static, immdeidate bit patterns in program code. The problem with this is that they are

evaluated at the destimations of all branches and jumps but not at the sources. These fail to prevent jumps into middle of functions (e.g. ones which bypass seucrity checks such as access control)

The method descibed in [27] ensures that whenever a machine-code instruction instruction transfers control, it targets a valid destination as determined by the CFG (ahead of time). When the destination is determined at runtime this must go through a dynamic check.

Static checks in [27] are made by rewriting machine code using modern tools for binary instrumentation to get around the resulting new memory addresses [33] [34].

Dynamic checks are a little more complex. If this need to be understood then lets do that at a later time.

Instrumentation codes are used.If this need to be understood then look up instrumentation codes.

Standard control flow analysis techniques exist [35] [36] [37] which run at compile time. Unique IDs could be created at run-time. Could this be useful to bind hardware to software?.

Tools used: Vulcan [33], Techniquies from programming languages and intrustion detection literatures [35] [38] [39] [37]. Measured overhead on SPEC computation benchmarks http://www.spec.org/cpu2017/.

2.2.5 Hardware-based CFI

[1] includes a reference to [40] , a survey for software-based CFI.

Generating CFG from static analysis is troublesome and often an over-approximation is used which is not fully precise [41] [42], again [40] classifies the precision of computing CFG using differenct static analysis techniques.

Problems: Unintended branches [2] - the majority of gadgets in a program consists of unintended branches.

Seems similar: [43]

Branch limitation is an alternative to CFG [44] [45]. Another alternative is Code Pointer Integrity described in [46] and implemented in [47] [48] [49].

Generalised path signature analysis [50] while [51] implemented in Arm Cortex-M3.

2.2.6 Software-based CFI

[40] Performs a survey

2.2.7 Building Control Flow Graphs

[52] Talks of Control Flow Graph construction and uses Jakstab[53] for comparison.

"Jakstab translates machine code to a low level intermediate language on the fly as it performs data flow analysis on the growing control flow graph. Data flow information is used to resolve branch targets and discover new code locations. Other analyses can either be implemented in Jakstab to run together with the main control flow reconstruction to improve precision of the disassembly, or they can work on the resulting preprocessed control flow graph."

In his dissertation [54] the author of Jakstab goes into more detail about everything...

Jakstab creates a 'dot' extension file. describes DOT as a graph description language which can be viewed with such tools as

neo4j is a graphical database which could be an ideal method of storing the resultant CFGs. neo4j uses cypher as its description language - I have been unable to find a method of converting the resultant dot files to neo4j.

Chapter 3

Associated background

3.1 Introduction

This section analyses and summarises contributing and related academic work applicable to this project.

It will be broken up into several sections: the first section "Subject-matter Surveys" will discuss works which describe the problems to be addressed and existing solutions to said problems, this section also included a subsection on FPGA security which makes useful reading as many embedded systems are FPGA-based. The second section "Attacks" contains a brief look at other physical attacks not addressed in the first section. The third section "Solutions" gives a deeper analysis of a handful of existing solutions, some of which directly address binding of software and hardware and some of which focus on the related subject of secure software execution. The fourth section "Primitives" provides a brief introduction into some of the founding principles used in many of the solutions already described and which will be heavily used in this project. Finally the conclusion will sum up the literature seen so far and describe its place in relation to this project.

3.2 Subject-matter Surveys

Many surveys on solutions which increase security for firmware/software in embedded systems have been completed. One such survey [55] focusses on existing mature solutions, while others [56], [57] and [58] provide a look at broader principles. All of these surveys also paint a picture of the attacks and threats which embedded systems face. Other surveys exist on the technologies described in this project, one such survey is [59] which focuses on FPGA security.

These subject-matter surveys will be discussed in 3.2.1 and 3.2.2 and a deeper analysis of some of the solutions presented will be discussed in 3.4.1 and 3.4.2.

3.2.1 Surveys on Existing Solutions for IP Protection and Secure Execution

A survey in to technologies designed to ascertain trust for embedded systems is provided in [55]. They compare various technologies, some of which are mature and some in their infancy. Studied solutions include: Trusted Platform Module (TPM), Secure Elements(SE), hypervisors and virtualisation (e.g. Java Card and Intel's Trusted eXecution technology), Trusted Execution Environments (TEEs), Host Card Emulation (HCE) and Encryption Execution Environments (E3 - which has also been directly discussed in [28]). The paper's authors set out a series of criteria which solutions are tested against, including such criteria as "Centralised Control', where the trust technology is under the control of the issuer or the maintainer, and "Remote Attestation" where the trust technology provides assurance to remote verifiers that the system is running as expected. The paper goes on to describe each technology in a small amount of detail and populates a matrix of technologies vs. criteria.

In a survey of anti-tamper technologies [56], a series of *cracking* threats and software and hardware protection mechanisms are described, many of which apply to embedded systems. Such threats include:

- Reverse engineering, achieved through a variety of methods including gaining an understanding of software or simply *code lifting* where sections of code are re-used without understanding of their functionality;
- Violating code integrity, where code in injected into a running program to make it carry out illegal actions outside of the desired control-flow of the program.

Hardware solutions described include: using a trusted processor used to secure the boot of the system, using hardware to decrypt encrypted software from the hard-drive and RAM, using a hardware *token* which is required to be present for the software to run.

The advantages of using hardware solutions include: using a complex CPU which is difficult to defeat while not redirecting resource from the processor used for standard operation, it is more costly to repeat attacks on hardware than it is for software (physical access is required each time) and secure hardware can also control which peripherals can be connected to the system and which software (signatures) can be allowed to run. There are some disadvantages of using hardware solutions which need to be considered, including:

- Secure data traversing the secure to non-secure boundary needs to be encrypted (which creates an additional overhead for the main processor)
- Hardware solutions tend to be inflexible and less secure than commonly assumed
- Additional components can add to the cost of manufacture, which is a high priority for embedded systems design.

Software solutions described include:

- Encryption wrappers, where all or just the critical portions of software are stored in a ciphertext form and dynamically decrypted. The value of this is that the attacker will not see all of the source program at the same time, however they can piece it together through snapshots or simply learn the encryption key/s. This paper does not cite any references for the subject of encryption wrappers;
- Code obfuscation, where the look of the code is adjusted to make it not easily readable or understandable by the attacker but performs in the same manner;
- Software watermarking and fingerprinting, which can be used for proof of ownership or authorship and for finding the source of leak of the software;
- Guarding, which is the act of adding code purely to perform anti-tamper functionality. An example of guarding is comparing checksums of running code to expected value and performing certain actions if they do not match. It is recommended that guarding is implemented automatically rather than manually as providing sufficient coverage is a complex task. It is also noted that a guard should not react immediately so as to not reveal the point in the code which triggered it.

The paper also describes a series of steps to take when using anti-tamper technology as put forward by the "Defence Acquisition Guidebook" created by the Defence Acquisition University [60].

A similar survey [57] covers three types of attacks: reverse engineering, software piracy and tampering which it describes as "malicious host attacks". To defend against such attacks the paper states three corresponding defences: code obfuscation (as well as anti-disassembly and anti-debugging measures), watermarking and tamper-proofing. The authors note that they could not find a wealth of information on tamper-proofing at the time of writing (2002) but they do draw an interesting parallel with the anti-tamper mechanisms used in computer viruses.

3.2.2 Defence against fault injection

A series of high-coverage tests for security protections against fault injection attacks were run and described in [58]. It describes 17 different countermeasures, including: countermeasures protecting the data layer, combinations of data protection methods, countermeasures protecting control flow layer, combinations of control flow protection methods and combinations of data and control flow protection. To test these methods the authors produced a high number of simulated fault injections on a simulator of an ARM-Cortex-M3 processor running a benchmark application representing a bank card.

The experiments found that a combination of redundant condition checks (such as data duplication) and source and destination IDs reached the best coverage with moderate performance overhead. They also found that simple ID-based inter-block control checking were able to outperform more sophisticated (and complex) methods such as Control-Flow Checking by Software Signatures (CFCSS) as seen in [51] and Assertions for Control-Flow Checking (ACFC) seen in [61].

3.2.3 FPGA Security

An excellent high-coverage survey on FPGA security is provided in [59], its contents include the background of FPGAs, attacks associated with FPGAs, defences for protecting FPGA implementations (existing at the time and ongoing research) and many more.

3.3 Attacks

The following are some examples of analysis of threats, all of which are aimed towards disrupting the flow of software, the likes of which are the focus secure software execution solutions.

Attacks which can be used to break instruction-level countermeasures are described in [62]. This paper discusses various attack countermeasures and how these are circumvented. The only countermeasures addressed in this paper are algorithm-level and instruction-level (both of which are mostly redundancy-based). This paper suggests that a purely software-based countermeasure could be a futile defence.

Findings that physical faults can be injected in a non-random manner and in a low cost environment are presented in [63], this contradicts assumptions made in many of the examined solutions that physical attacks are too costly. It finds that instruction-skipping attacks create a vulnerability to skipped-instruction errors (which, in my opinion, drives the motivation behind control-flow monitoring right down to the intra-block level).

Further details on side-channel attacks, as well as a brief description of the security concerns associated with FPGAs are provided in [64].

3.4 Solutions

A myriad of creative technical solutions have been put forward which address the problems already discussed. They can be placed in to one of two categories - binding hardware 3.4.1 ([28], [65], [66] and [67]) and software or secure software execution 3.4.2 ([51], [68], [20] and [69]). Hybrids of the two approaches are presented in [70] and [71].

3.4.1 Binding Hardware and Software

Hardware software binding is a technique where hardware and software a co-designed in such a way that software needs to be tailored to run on an individual instance of hardware. The same principle works the other way in that a individual piece of hardware will not execute software unless is it specifically tailored to it.

The first piece of work we consider is [28]. The problem the paper aims to address is device counterfeiting. An example of the requirement for binding of hardware and software is for Graphics Processing Units (GPUs),

where GPUs are fabricated and then tested on their operating performance and subsequently graded. Once graded, the GPUs are loaded with firmware which controls their voltage and clockspeed. The paper states that firmware aimed towards the superior graded GPUs could be installed on lesser graded GPUs which would then be sold on as superior GPUs.

The attacker described in [28] is one which has several special attributes: they have physical access the the device, access to the device storage where they can read and copy the entire contents of memory, they are able to use hardware which has been built to the exact specifications as the original hardware and they can "read and copy any data which is loaded onto any of the buses which make up the embedded system". The attacker's aim is to either create a counterfeit platform which performs and functions in the same manner as the original or to install software retrieved from the legitimate product onto different (counterfeit) hardware.

The method presented in [28] uses a function applied to either previous contents of memory or a randomly generated number to produce a mask which is applied to the program instructions residing in memory. The intention is that the CPU unmasks the contents as part of the execution process prior to actually carrying out the operation. The paper discusses the options for the mask-creating function, suggesting the use of hash-functions, block ciphers or PUFs before finally selecting PUFs due to their intrinsic nature. The act of masking the software has been undecided in this paper, which suggests that either the software is masked prior to loading or is masked during the loading process. The paper's author describes the provisioning process in a further paper [72].

The second piece of work we consider is [65] where the goal is to "protect against intellectual property (IP) extraction or modification on embedded devices without dedicated security mechanisms". In this paper the attacker is aiming to extract IP (in the form of software or secrets) stored on the device. The attacker may use this information in any of the following ways: they may implement the extracted information on counterfeit devices, they may modify the software or data to remove licensing restrictions or unlock premium features, they may downgrade to earlier firmware versions in order exploit previous vulnerabilities, they may wish to alter firmware to capture valuable data such as password, change output data such as readings on smart meters or reveal secrets such as cryptographic keys.

although how does this help with this? I suppose the earlier firmware would have to have been taken from the same device.

In [65] the attacker is assumed to have physical access to the device, can read the contents of external memory and can inspect and modify on-chip memory values. The assumed limitations placed on the attacker by the paper are that the attacker cannot change the code of the boot-loader as it is stored in a masked read-only memory (ROM), they cannot replace the ROM chip with one with a boot-loader under the attacker's control as the ROM chip would be heavily integrated on a system-on-a-chip (SoC) so would require skill levels outside of those expected of the attacker and finally the attacker cannot read the start-up values of the on-chip SRAM during start-up which are protected by the boot-loader and are erased once read by the boot-loader.

How are the start-up values on the chip protected?

The method described in [65] heavily utilises PUFs created using the SRAM start-up values to derive a key used to decrypt the firmware. The firmware is decrypted by the the boot-loader before being loaded and executed. The system was implemented on a SoC platform using a two-stage bootloader (u-boot). The paper's authors provide an extensive review of SRAM PUFs for ARM Cortex-M and Pandaboard's IMAP and includes a description of Fuzzy Extractor design used by the solution.

are the where are the plaintext instructions now stored?

The third piece of work [70] has not been published in a well-established journal however the authors have been invited to present their findings in [73]. Here the problem of injection of malicious code is also addressed, as well as prevention of code reverse-engineering. This paper uses secure execution to bind hardware to software.

The attacker identified in [70] has physical access to the processor and peripheral connections and that they can read out contents of memory or registers. They are also assumed to be able to place arbitrary data into the main memory of the processor (either locally or remotely). Attacks comprising denial-of-service (DoS)achieved by, for example, injection of random invalid instructions and hardware side-channel attacks

have not been addressed.

The method described has been labelled "Secure Execution PUF-based Processor" or SEPP. The operating principle of SEPP is the encryption of basic blocks (which have exactly one entry point and one exit point) which make up programs. The blocks are encrypted using a symmetric cipher in CTR mode with the parameters set in relation to instruction location within a block and the block's location within memory. The key used for this encryption is set by the user. SEPP utilises a ring-operator (RO) PUF to create a new key used to encrypt the users key. The decryption module is included in the instruction fetch stage of the processor's pipeline and makes use of first-in, first-out (FIFO) buffer to store encryption pads before they are needed by the processor (therefore making use of spare time provided by instructions which take more than one processor cycle). This system also implements u-boot as a bootloader which has been modified to provide the functions of the security kernel. It appears that due to the nature of this method (the device tailoring the software to itself), it does not prevent malices uploading of a new program to device which the device then processes.

Where us Ku (the user key) stored? As it is used to decrypt it is surely readable by the attacker? If so the programmer could be extracted and decrypted.

The fourth method identified in [71] had identified illegitimate reproduction as a problem that requires a solution. It also identifies modification of software to bypass the need for purchasing a license for particular features as another attack scenario. Here the attacker has the ability to read and modify the content of external memory such as flash memory or RAM, they can also do the same with internal memory including software with hard-coded secrets and cryptographic keys. The method consists of four basic mechanisms: two check functions and two response functions. The first check function hashes the native program code and compares this to the current running code. The second check function uses a SRAM-derived PUF to measure the authenticity of the device. If these functions indicate that either the software is not in its intended state or is running on the incorrect device the first response function adjusts the flow of the program to move in a random manner and the second response corrupts the program's execution stack. Both response functions are designed to cause a malfunction in the program. One has to question the safety of having a program jump to a random block.

The fifth method described in [66] is developed to protect against IP theft or reverse engineering. This paper does not describe the attacker but makes some assumptions that they will not be able to access the PUF-based key used for encryption as it is internal to the FPGA. This method uses an obfuscated secure ROM to start the boot process, checking and running the integrity kernel which decrypts and runs the software using the PUF-based key. In my opinion, the problem with this solution is the reliance on an obfuscated ROM. This is because once there is an understanding of a ROM it could be possible for malicious software to be written in a manner that is accepted by the boot program in ROM.

So is is the obfuscated ROM the same on each chip and will it be able to be understood in order to create malicious integrity and security kernels or just the software? Also once decrypted where are the plain text instructions stored?

An honourable mention should be made for [67] which was published in 2006 and led the way in using PUFs to secure software and also clearly describes the enrolment process with defined message exchanges.

Actually why have I ranked this one so low? If it's just because of its age I should re-review.

3.4.2 Secure execution

Secure software execution (or control-flow security/integrity) has the goal of preventing the desired results of attacks against program control flow, which aim to use physical attacks to make the execution of running programs jump to blocks which should not be running at that particular time (e.g. an administration function).

The first solution [51] raises the point that most research on fault-attacks has been aimed towards cryptographic functions which result in gaining knowledge of the secret key. This paper takes this further by considering the modification of games consoles to make them skip the function which checks the validity of loaded software. The paper focuses on securing against fault-attacks.

The method mainly utilises control-flow integrity to solve the stated problem. The basic principle behind control-flow integrity is the understanding of the basic blocks of a making up a program. The blocks will flow into one another control-flow instructions. This flow can be described as the control-flow graph (CFG) and a correctly functioning program will abide by this graph. The signature of the program flow is created and compared against the expected value according to the CFG. The solution provides assistance to C programmers in that it automatically inserts signature updates, although programmers can also insert them manually for critical sections of the program. This functionality has been provided via the editing of LLVM compiler. If using assembly code the programmer must manually insert signature updates whenever branches, loops and function calls are encountered. The control flow signatures are calculated through a recursive disassembling approach. Important principles introduced in this paper (along the same lines as CFG) are generalized path signature analysis (GPSA) and continuous-signature monitoring (CSM). This paper does require modifications to be made to Cortex-M3 architecture.

Should I directly address GPSA and CSM?

How hard is this/ how is this done? Perhaps need more info on processor architecture

Do I include this? "In order to check the running program's integrity a "derived signature" is calculated on the running code's path, this can then be compared to the corresponding derived signature of the intended route of the CFG"?

The second paper [68] (ConFirm) states that "given the critical role of firmware, implementation of effective security controls against firmware malicious actions is essential", having read the various prior examples seen we can safely agree with this.

The attacker is assumed to have the ability to inject and execute malicious code, or call existing functions not abiding by the control-flow graph. These attacks are assumed to be possible either on-line or off-line (which would require a device reboot after uploading of malicious firmware image) and depending on the design of the device the firmware alterations could be achieved locally or remotely.

The method employed revolves around making use of hardware performance counters (HPCs) which count various types of events. The HPCs are utilized in conjunction with a bootloader (which sets checkpoints, initialises an HPC handler and contains a database of valid HPC-based signatures). While the program is run, HPC values are checked once checkpoints are reached (checkpoints are placed at the beginning and end of each basic block, as well as one randomly inserted between). The checkpoints actually redirect the control flow to the ConFirm core module to compare the HPC value with those stored in the database containing valid values. If the check fails ConFirm will report a deviation, which could be used to run a fault sequence, such as "rebooting the system, generating an alarm and disabling the device".

If the database is stored in RAM how is it updated when new FW is released?.

The third paper [20] approaches secure execution from a slightly different direction: remote attestation. It aims to provide remote attestation of an application's control flow path during operation. The paper excludes physical attacks and instead focusses on execution path attacks, they assume that the subject device features data execution prevention (DEP) and a secure trust anchor that provides an isolated measurement engine and can generate the fresh authenticated attestation report.

Look up DEP

The method builds a hash of the path taken from node to node which is then reported to the verifier. Loops are dealt with in a novel manner to work around the issue posed due to the infinite hash available when the number of iterations of a loop is set dynamically.

Read more into this Could this be a good project basis?

The fourth paper [69] lists various attacks, ranging from buffer overflow attacks to physical attacks. Their method measures inter-procedural control flow, intra-procedural control flow and instruction stream integrity. Control flow monitoring is provided by an additional hardware element which tracks instruction addresses and compares them to known acceptable values stored in lookup tables. Instruction steam integrity monitoring utilises the lookup tables in addition to corresponding hash values of the basic clock. If a violation is discovered it is reported to the processor which should then terminate execution of the current program. Details of the violation are included in the report to the processor to enable a finer-grained view of the violation.

3.5 Primitives

3.5.1 Control-flow Graphs

The use of control flow checking for accidental program flow changes is considered in [61], but still presents the basic principle. It describes basic blocks, control-flow between blocks, how these make up program graphs and finally how these graphs can be used to indicate control-flow error. The paper presents two existing error detection methods but finds faults in both so presents a novel solution addressing the previous solutions' shortcomings.

Chapter 9 of [35] contains a wealth of information on data-flow and will provide a good basis for understanding both the essence of data(control)-flow and how compilers use them for code optimisation. This chapter should be referenced in order to gain a meaningful understanding of control-flow.

3.5.2 PUFs

A huge amount of prior research has been undertaken into PUFs and their application towards security in embedded systems. One very good overview is the PhD thesis [74], which provides a thorough examination into most aspects of PUFs including the types, analysis of each type in terms of uniqueness and reproducibility and uses including entity-authentication and key generation.

Much of the literature ([28], [65], [70], [71], [66] and [67]) already described explain the use of PUFs and their reasoning behind their choice in PUFs.

3.6 Conclusion

In this literature review we have seen the reasons why the security of firmware of embedded devices is an important matter which needs to be addressed. We then saw reviews of existing solutions and introductions to primitives used. After an introduction to potential physical attacks we then provided an in depth review of solutions which enabled the binding of hardware and software and solutions which provide secure software. Finally we looked at two primitives which will be useful to this project.

3.7 Important Principals

This section provides an insight into what we need to know in order to solve any problems.

It will be broken up into several sections: the first section "Introduction to Embedded Systems" will attempt to define what an embedded systems is, specifying unique characteristics. The second section "Problem description and requirements" will identify and discuss security problems, codifying these into a set of requirements. The third section "Common Attacks" gives a deeper analysis of attacks used to exploit the problems identified in the second section. The fourth section "Control Flow Integrity" introduces us to the notion of control flow integrity (CFI) and examines, from a high level, existing implementations for providing CFI. The fifth section "Data Flow Integrity" discusses a data-focusses parallel of CFI. The sixth section "Existing Solutions" will identify, discuss and analyse existing solutions and compare them to the requirements attained in the second section. Finally the conclusion will provide a brief summation of the previous sections.

3.8 Introduction to Embedded Systems

In this section we will define embedded systems, their uses, understand their key properties or criteria and describe some common implementations.

3.8.1 A Definition of Embedded Systems

Embedded systems are small-form, low power computer systems.

From Embedded Systems Design (Arnold S Berger): VS a PC:

- Dedicated to specific task (PCS are generic computing platforms) - A change of task will usually require redesigning an entire system
- Supported by a wide array of processors and processor architectures
- Usually cost sensitive
- Real-time constraints - if has an OS will be RTOS
- Implications of software failure far more serious than desktop systems (due to their usage)
- Often have power constrains
- Often operate in extreme conditions
- Far fewer system resources
- Often store object code in ROM
- Require specialized tools and efficiently design methods
- Often have dedicated debugging circuitry.

3.8.2 Uses of Embedded Systems

Embedded systems are ideal for applications where the computer systems has one role - for example in traffic lights, where the flow of traffic needs to be monitored and the timing of the light sequence needs to be controlled.

The low power consumption of embedded systems makes them ideal for use in medical devices which can only have small batteries such as insulin injection pumps, or as part of a collection of sensors which are served by a low power bus.

The small-form (and corresponding low-weight) means they are a good candidate for use in aeronautics, such as sensor controllers on aircraft or flight computers in missiles.

Consumer IoT also makes use of embedded systems, exploiting their low-cost, footprint and power consumption to add smart functions to what have historically been simple devices such as kettles or fridges.

3.8.3 Key Properties and Criteria of an Embedded System

An embedded system could have a small form factor, low computing power (in comparison to full computer systems such as PCs), low power consumption and resistant to hostile environments.

3.8.4 Common Implementations of Embedded Systems

A large number of manufactures has an interest in embedded systems. A big player is ARM with their Cortex-M Series of CPUs, these are commonly used in applications such as...

Another angle are field programmable gate arrays (FPGAs) which are essentially software defined circuits. These can be set up in a variety of ways, but in modern applications they often utilise soft-core processors - where FPGA code is used to define a processor (for example following the RISC-V instruction set).

System on Chips (SoCs) are a good example of an embedded system - here everything is on a chip.

3.9 Problem Description, Assumptions and Requirements

Instructions can be skipped, replaced or be re-ordered. Malicious code can be injected or return-oriented programming can make use of existing code simply by redirecting return addresses.

Skipping instructions can be used to bypass checks.

Replacing instructions can be used to extract data or pass checks etc.

Re-ordering instructions can be used to bypass cryptographic protections (similar to ECB vulnerabilities) to achieve the attackers goals.

Attackers goals could be: changing program flow, ...

Assumptions could include: attacker having direct access to memory, not internal registers. They cannot break cryptographic protections.

Requirements:

- An attacker cannot change the sequential order of instructions.
- An attacker cannot change the execution order of blocks
- An attacker cannot re-order instructions
- An attacker cannot skip instructions
- An attacker cannot subvert control flow to existing code (libc...)

3.10 Common Attacks

Physical attacks - power ...

Return to lib-c

Return oriented programming

3.11 Control Flow Integrity

Fine grained, course grained.

Flow down to low level - sequential Basic blocks

Control flow graphs

Shadow return address tables

Program counter

3.12 Data Flow Integrity

Data flow integrity is a measure used to ensure that data only flows in the correct direction from authorised callers. This can be used to ensure only the correct caller has changed an important variable - for example one which a decision is based on (e.g. if...), this can also be used to protect the stack. One example of this maintains a table for each piece of data accessed with the address of last accessor. If the last accessor is not within the legal list a security failure exists and an exception is thrown.

3.13 Existing Solutions

Software - Microsoft (talked about in some sort of paper). Other software implementations

Hardware - program counter, encrypted computing, all of the CFG examples

Additional notable - NX bit, canaries

Dynamic taint analysis - detect control-data and non-control-data attacks. No need for source code. But false positives, high overhead and require hardware support.

Problems: PUFs CFG generation Checking code being altered

Compare to objectives

3.14 Conclusion

Chapter 4

Comparison of Solutions

Chapter 5

Theoretical Solution

Chapter 6

Practical Implementation

Chapter 7

Security Analysis

Chapter 8

Conclusion and Further Work

Appendix A

Appendix Title

This is my appendix

Bibliography

[1] R. de Clercq and I. Verbauwhede, "A survey of Hardware-based Control Flow Integrity (CFI)," vol. 1, pp. 1–27, 2017. arXiv: 1706.07257. [Online]. Available: http://arxiv.org/abs/1706.07257.

[2] M. Kayaalp, M. Ozsoy, N. A. Ghazaleh, and D. Ponomarev, "Efficiently securing systems from code reuse attacks," *IEEE Transactions on Computers*, vol. 63, no. 5, pp. 1144–1156, 2014, ISSN: 00189340. DOI: 10.1109/TC.2012.269.

[3] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi, "LO-FAT: Low-Overhead Control Flow ATtestation in Hardware," 2017, ISSN: 0738100X. DOI: 10.1145/3061639.3062276. arXiv: 1706.03754. [Online]. Available: http://arxiv.org/abs/1706.03754%7B%5C%%7D0Ahttp://dx.doi.org/10.1145/3061639.3062276.

[4] R. Anderson and M. Kuhn, "Tamper Resistance — a Cautionary Note," pp. 1–11, 1996. [Online]. Available: http://www.cl.cam.ac.uk/%7B~%7Drja14/Papers/tamper.pdf.

[5] O. Kömmerling and M. G. Kuhn, "Design Principles for Tamper-Resistant Smartcard Processors," *USENIX Workshop on Smartcard Technology*, pp. 9–20, 1999.

[6] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic Analysis: Concrete Results," pp. 251–261, 2007. DOI: 10.1007/3-540-44709-1_21.

[7] J.-J. Quisquater and D. Samyde, "ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards," in *ACM Transactions on Embedded Computing Systems*, 3, vol. 18, Apr. 2001, pp. 200–210. DOI: 10.1007/3-540-45418-7_17. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3323876.3284361%20http://link.springer.com/10.1007/3-540-45418-7%7B%5C_%7D17.

[8] H. Bar-el and H. Choukri, "The Sorcerer ' s Apprenctice ' s Guide to Fault Attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/1580506/%7B%5C%%7D0Ahttp://www.hbarel.com/media/blogs/hagai-on-security/Sorcerers%7B%5C_%7DApprentice%7B%5C_%7DGuide.pdf.

[9] S. P. Skorobogatov and R. J. Anderson, "Optical Fault Induction Attacks," pp. 2–12, 2007. DOI: 10.1007/3-540-36400-5_2.

[10] S. P. Skorobogatov, "Semi-invasive attacks-a new approach to hardware security analysis," *Technical report, University of Cambridge, Computer Laboratory*, no. 630, p. 144, 2005, ISSN: 1476-2986. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.228.2204%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf.

[11] J. Blömer and J.-P. Seifert, "Fault Based Cryptanalysis of the Advanced Encryption Standard (AES)," in, 2003, pp. 162–181. DOI: 10.1007/978-3-540-45126-6_12. [Online]. Available: http://link.springer.com/10.1007/978-3-540-45126-6%7B%5C_%7D12.

[12] R. Anderson and M. Kuhn, "Low cost attacks on tamper resistant devices," in, 1998, pp. 125–136. DOI: 10.1007/BFb0028165. [Online]. Available: http://link.springer.com/10.1007/BFb0028165.

[13] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the Importance of Eliminating Errors in Cryptographic Computations," *Journal of Cryptology*, vol. 14, no. 2, pp. 101–119, Mar. 2001, ISSN: 0933-2790. DOI: 10.1007/s001450010016. [Online]. Available: http://link.springer.com/10.1007/s001450010016.

[14] S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine," *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2003-Janua, pp. 154–165, 2003, ISSN: 10816011. DOI: 10.1109/SECPRI.2003.1199334.

[15] D. H. Habing, "The use of lasers to simulate radiation-induced transients in semiconductor devices and circuits," *IEEE Transactions on Nuclear Science*, vol. 12, no. 5, pp. 91–100, 1965, ISSN: 15581578. DOI: 10.1109/TNS.1965.4323904.

[16] D. Samyde, S. Skorobogatov, R. Anderson, and J. J. Quisquater, "On a new way to read data from memory," *Proceedings - 1st International IEEE Security in Storage Workshop, SISW 2002*, pp. 65–69, 2003. DOI: 10.1109/SISW.2002.1183512.

[17] U. Maurer, "Differential Power Analysis," *Advances in Cryptology — CRYPTO' 99*, vol. 1666, p. 785, 1999, ISSN: 0302-9743. DOI: 10.1007/3-540-48405-1. [Online]. Available: http://www.springerlink.com/content/cdp6u8xpenkkx08m.

[18] S. Mangard, *Power analysis attacks : revealing the secrets of smart cards*. New York: Springer, 2007, ISBN: 0387308571.

[19] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," *CCS '07 Proceedings of the 14th ACM conference on Computer and communications security*, pp. 552–561, 2007. DOI: 10.1145/1315245.1315313.

[20] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-FLAT: Control-Flow Attestation for Embedded Systems Software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, New York, New York, USA: ACM Press, 2016, pp. 743–754, ISBN: 9781450341394. DOI: 10.1145/2976749.2978358. arXiv: 1605.07763. [Online]. Available: http://arxiv.org/abs/1605.07763%20http://dl.acm.org/citation.cfm?doid=2976749.2978358.

[21] R. P. Lee, K. Markantonakis, and R. N. Akram, "Ensuring Secure Application Execution and Platform-Specific Execution in Embedded Devices," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 3, pp. 1–21, Apr. 2019, ISSN: 15399087. DOI: 10.1145/3284361. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3323876.3284361.

[22] J. L. Danger, A. Facon, S. Guilley, K. Heydemann, U. Kuhne, A. Si Merabet, and M. Timbert, "CCFI-Cache: A transparent and flexible hardware protection for code and control-flow integrity," *Proceedings - 21st Euromicro Conference on Digital System Design, DSD 2018*, pp. 529–536, 2018. DOI: 10.1109/DSD.2018.00093.

[23] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "HCFI," in *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy - CODASPY '16*, New York, New York, USA: ACM Press, 2016, pp. 38–49, ISBN: 9781450339353. DOI: 10.1145/2857705.2857722. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2857705.2857722.

[24] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "Hafix," *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2015. DOI: 10.1145/2744769.2744847.

[25] R. de Clercq, J. Götzfried, D. Übler, P. Maene, and I. Verbauwhede, "SOFIA: Software and control flow integrity architecture," *Computers and Security*, vol. 68, no. 2, pp. 16–35, 2017, ISSN: 01674048. DOI: 10.1016/j.cose.2017.03.013.

[26] D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin, "Strategy without tactics," *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2016. DOI: 10.1145/2897937.2898098.

[27] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security - CCS '05*, New York, New York, USA: ACM Press, 2005, p. 340, ISBN: 1595932267. DOI: 10.1145/1102120.1102165. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1102120.1102165.

[28] R. P. Lee, K. Markantonakis, and R. N. Akram, "Binding Hardware and Software to Prevent Firmware Modification and Device Counterfeiting," *Proceedings of the 2nd ACM International Workshop on Cyber-Physical System Security - CPSS '16*, pp. 70–81, 2016. DOI: 10.1145/2899015.2899029. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2899015.2899029.

[29] Ú. Erlingsson and F. B. Schneider, "SASI enforcement of security policies," no. September, pp. 87–95, 2004. DOI: 10.1145/335169.335201.

[30] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002, ISSN: 00189529. DOI: 10.1109/24.994926.

[31] A. Sharma, "SoftWare Implemented Fault Tolerance (SWIFT)," 2012.

[32] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," *Proceedings - 9th IEEE International On-Line Testing Symposium, IOLTS 2003*, pp. 137–143, 2003. DOI: 10.1109/OLT.2003.1214380.

[33] A. Edwards, A. Srivastava, and H. Vo, "Vulcan: Binary transformation in a distributed environment," p. 12, 2001. [Online]. Available: https://www.microsoft.com/en-us/research/publication/vulcan-binary-transformation-in-a-distributed-environment/.

[34] A. Srivastava and A. Eustace, "ATOM: a system for building customized program analysis tools," *WRL Research Report (ACM SIGPLAN Notices)*, vol. 29, no. 6, pp. 196–205, 1994, ISSN: 03621340. DOI: 10.1145/773473.178260. [Online]. Available: http://dl.acm.org/citation.cfm?id=773473.178260.

[35] A. V. Aho, *Compilers : principles, techniques, and tools.* Second edi. Pearson custom library, 2014, ISBN: 9781292024349.

[36] D. C. Atkinson and S. Clara, "Call Graph Extraction in the Presence of Function Pointers," *Computer Engineering*,

[37] D. Wagner and R. Dean, "Intrusion detection via static analysis," pp. 156–168, 2002. DOI: 10.1109/secpri.2001.924296.

[38] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2003-Janua, pp. 62–75, 2003, ISSN: 10816011. DOI: 10.1109/SECPRI.2003.1199328.

[39] R. Gopalakrishna, E. H. Spafford, and J. Vitek, "Efficient intrusion detection using automaton inlining," *Proceedings - IEEE Symposium on Security and Privacy*, pp. 18–31, 2005, ISSN: 10816011. DOI: 10.1109/SP.2005.1.

[40] N. Burow, "Control-Flow Integrity : Precision , Security , and Performance," vol. V,

[41] N. Carlini, A. Barresi, E. T. H. Zürich, M. Payer, D. Wagner, T. R. Gross, E. T. H. Zürich, N. Carlini, A. Barresi, D. Wagner, and T. R. Gross, "Sec15-Paper-Carlini," 2015.

[42] J. Kinder and D. Kravchenko, "LNCS 7148 - Alternating Control Flow Reconstruction," *Verification, Model Checking, and Abstract Interpretation*, pp. 267–282, 2012.

[43] S. Mao and T. Wolf, "Hardware support for secure processing in embedded systems," *IEEE Transactions on Computers*, vol. 59, no. 6, pp. 847–854, 2010, ISSN: 00189340. DOI: 10.1109/TC.2010.32.

[44] W. He, S. Das, W. Zhang, and Y. Liu, "No-Jump-into-Basic-Block," pp. 1–6, 2017. DOI: 10.1145/3061639.3062291.

[45] I. Corporation, "Control-flow Enforcement Technology Preview," *Intel Specifications*, no. June, pp. 1–136, 2017. [Online]. Available: http://intel.com/.%7B%5C%%7D0Awww.intel.com/design/literature.htm..

[46] Q. P. Security, "Pointer Authentication on ARMv8.3 Design and Analysis of the New Software Security Instructions," no. January, 2017.

[47] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-Pointer Integrity," 2014. [Online]. Available: http://infoscience.epfl.ch/record/204783.

[48] A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh, "Cryptographically Enforced Control Flow Integrity," pp. 941–951, 2014. arXiv: 1408.1451. [Online]. Available: http://arxiv.org/abs/1408.1451.

[49] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," *Proceedings - IEEE Symposium on Security and Privacy*, pp. 48–62, 2013, ISSN: 10816011. DOI: 10.1109/SP.2013.13.

[50] A. Mahmood and E. J. Mccluskey, "Concurrent Error Detection Using Watchdog Processors—A Survey," *IEEE Transactions on Computers*, vol. 37, no. 2, pp. 160–174, 1988, ISSN: 00189340. DOI: 10.1109/12.2145.

[51] M. Werner, E. Wenger, and S. Mangard, "Protecting the Control Flow of Embedded Processors against Fault Attacks," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9514, 2016, pp. 161–176, ISBN: 978-3-642-37287-2. DOI: 10.1007/978-3-319-31271-2_10. arXiv: 9780201398298. [Online]. Available: http://link.springer.com/10.1007/978-3-319-31271-2%7B%5C_%7D10.

[52] M. H. Nguyen, T. B. Nguyen, T. T. Quan, and M. Ogawa, "A hybrid approach for control flow graph construction from binary code," *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, vol. 2, no. 2, pp. 159–164, 2013, ISSN: 15301362. DOI: 10.1109/APSEC.2013.132.

[53] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries - Tool paper," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5123 LNCS, pp. 423–427, 2008, ISSN: 03029743. DOI: 10.1007/978-3-540-70545-1_40.

[54] D.-I. J. Kinder, "Static Analysis of x86 Executables Statische Analyse von Programmen in x86 Maschinensprache," no. November, 2010. [Online]. Available: https://infoscience.epfl.ch/record/167546/files/thesis.pdf.

[55] C. Shepherd, G. Arfaoui, I. Gurulian, R. P. Lee, K. Markantonakis, R. N. Akram, D. Sauveron, and E. Conchon, "Secure and Trusted Execution: Past, Present, and Future - A Critical Review in the Context of the Internet of Things and Cyber-Physical Systems," in *2016 IEEE Trustcom/BigDataSE/ISPA*, IEEE, Aug. 2016, pp. 168–177, ISBN: 978-1-5090-3205-1. DOI: 10.1109/TrustCom.2016.0060. [Online]. Available: http://ieeexplore.ieee.org/document/7846943/.

[56] E. D. Bryant, M. J. Atallah, M. R. Stytz, M. J. Atallah, E. D. Bryant, and M. R. Stytz, "A Survey of Anti-Tamper Technologies," *The Journal of Defense Software Engineering*, no. November, pp. 12–16, 2004.

[57] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation - Tools for software protection," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 735–746, 2002, ISSN: 00985589. DOI: 10.1109/TSE.2002.1027797.

[58] N. Theissing, D. Merli, M. Smola, F. Stumpf, and G. Sigl, "Comprehensive Analysis of Software Countermeasures against Fault Attacks," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pp. 404–409, 2013, ISSN: 15301591. DOI: 10.7873/DATE.2013.092. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6513538.

[59] S. Drimer, "Volatile FPGA design security – a survey," *University of Cambridge*, pp. 1–51, 2008. [Online]. Available: http://www.cl.cam.ac.uk/%7B~%7Dsd410/papers/fpga%7B%5C_%7Dsecurity.pdf.

[60] DAU, "Defense Acquisition Guidebook," pp. 1–969, 2011. [Online]. Available: https://www.dau.mil/tools/dag.

[61] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *Proceedings. 16th IEEE Symposium on Computer Arithmetic*, vol. 16, IEEE Comput. Soc, Nov. 2003, pp. 581–588, ISBN: 0-7695-2042-1. DOI: 10.1109/DFTVS.2003.1250158. [Online]. Available: http://ieeexplore.ieee.org/document/1250158/.

[62] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont, "Software Fault Resistance is Futile: Effective Single-Glitch Attacks," *Proceedings - 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016*, pp. 47–58, 2016. DOI: 10.1109/FDTC.2016.21.

[63] M. S. Kelly, K. Mayes, and J. F. Walker, "Characterising a CPU fault attack model via run-time data analysis," in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, IEEE, May 2017, pp. 79–84, ISBN: 978-1-5386-3929-0. DOI: 10.1109/HST.2017.7951802. [Online]. Available: http://ieeexplore.ieee.org/document/7951802/.

[64] C. H. Gebotys, *Security in embedded devices*, ser. Embedded systems. New York ; London: Springer, 2010, ISBN: 144191529x.

[65] A. Schaller, T. Arul, V. Van Der Leest, and S. Katzenbeisser, "Lightweight anti-counterfeiting solution for low-end commodity hardware using inherent PUFs," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8564 LNCS, pp. 83–100, 2014, ISSN: 16113349. DOI: 10.1007/978-3-319-08593-7_6.

[66] M. A. Gora, A. Maiti, and P. Schaumont, "A flexible design flow for software IP binding in FPGA," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 719–728, 2010, ISSN: 15513203. DOI: 10.1109/TII.2010.2068303.

[67] E. Simpson and P. Schaumont, "Offline HW / SW Authentication for Reconfigurable Platforms," *Cryptographic Hardware and Embedded Systems (CHES)*, pp. 1–13, 2006.

[68] X. Wang, C. Konstantinou, M. Maniatakos, and R. Karri, "ConFirm: Detecting firmware modifications in embedded systems using Hardware Performance Counters," *2015 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015*, pp. 544–551, 2016, ISSN: 1933-7760. DOI: 10.1109/ICCAD.2015.7372617.

[69] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 12, pp. 1295–1308, Dec. 2006, ISSN: 1063-8210. DOI: 10.1109/TVLSI.2006.887799. [Online]. Available: http://ieeexplore.ieee.org/document/4052340/.

[70] S. Kleber, F. Unterstein, M. Matousek, F. Kargl, F. Slomka, and M. Hiller, "Secure Execution Architecture based on PUF-driven Instruction Level Code Encryption," *Cryptology ePrint Archive, Report 2015/651*, 2015. DOI: cr.org/2015/651.

[71] F. Kohnhäuser, A. Schaller, and S. Katzenbeisser, "PUF-Based Software Protection for Low-End Embedded Devices," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9229, 2015, pp. 3–21, ISBN: 9783319228457. DOI: 10.1007/978-3-319-22846-4_1. arXiv: arXiv:1506.07739v2. [Online]. Available: http://link.springer.com/10.1007/978-3-319-22846-4%7B%5C_%7D1.

[72] R. P. Lee, K. Markantonakis, and R. N. Akram, "Provisioning Software with Hardware-Software Binding," in *Proceedings of the 12th International Conference on Availability, Reliability and Security - ARES '17*, New York, New York, USA: ACM Press, 2017, pp. 1–9, ISBN: 9781450352574. DOI: 10.1145/3098954.3103158. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3098954.3103158.

[73] S. Kleber, F. Unterstein, M. Matousek, F. Kargl, F. Slomka, and M. Hiller, "Design of the Secure Execution PUF-based Processor (SEPP)," *Workshop on Trustworthy Manufacturing and Utilization of Secure Devices, TRUDEVICE 2015*, no. 2, pp. 1–5, 2015. DOI: 10.18725/OPARU-3255.

[74] R. Maes, *Physically Unclonable Functions: Constructions, Properties and Applications (Fysisch onkloonbare functies: constructies, eigenschappen en toepassingen)*, August. 2012, ISBN: 9789460185618. [Online]. Available: https://lirias.kuleuven.be/handle/123456789/353455.