Student Number: 100905113
Luke Atherton

Towards Auditing of Control-Flow-Integrity (CFIA)

Supervisor: Konstantinos Markantonakis

Submitted as part of the requirements for the award of the
MSc in Information Security
at Royal Holloway, University of London.

I declare that this assignment is all my own work and that I have acknowledged all quotations from published or unpublished work of other people. I also declare that I have read the statements on plagiarism in Section 1 of the Regulations Governing Examination and Assessment Offences, and in accordance with these regulations I submit this project report as my own work.
Signature:
Date:

Abstract

This is my abstract

Dedication

This is my dedication

Declaration

This is my declaration

Acknowledgements

Thanks everyone!

Contents

Listings

List of Figures

List of Tables

BBL Basic Block

SS Shadow Stack

CFG Control-Flow Graph

CFI Control-Flow Integrity

Chapter 1

Introduction

1.1 Control Flow

Control flow is how instructions within software applications transition from one to another. Applications can be logically broken down into basic blocks (BBLs) through which the executing application traverses. The tracing of these transitions forms the control flow taken.

1.1.1 Basic Blocks

The principle of basic blocks creates a useful way of breaking down the instructions of an application into manageable chunks containing sequences of instructions. A basic block is a set of instructions which sequentially run from beginning to end. For example take the pseudo-code in listing 1.1.

```
1  void GreaterThanPlusOne(int firstNum, int secondNum)
2      {
3          string message;
4          firstNum++;
5          if (firstNum > secondNum)
6          {
7              message = "firstNum plus one is greater than secondNum";
8          }
9          if (firstNum <= secondNum)
10         {
11             message = "firstNum plus one less than or equal to secondNum"
12         }
13     }
```

Listing 1.1: An example of code which can be broken down into basic blocks

The instructions from `string message;` (line 3) up to and including `if (firstNum > secondNum)` (line 5), *basic block 1*, will execute sequentially (barring an exception being thrown). This is a basic block. The contents within the first `if` (line 5) and also the second `if` (line 9) parenthesis are also basic blocks, *basic blocks 2 and 3*.

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 basic block 2 to basic block 3 as there is no way that under normal operation both of these sequences of instructions would execution after one another.

Other examples of programming control flow statements which could form basic blocks are switch cases, loops, break statements and functions, as the these control the diversion of instruction execution depending on the current running circumstances (e.g. variable state).

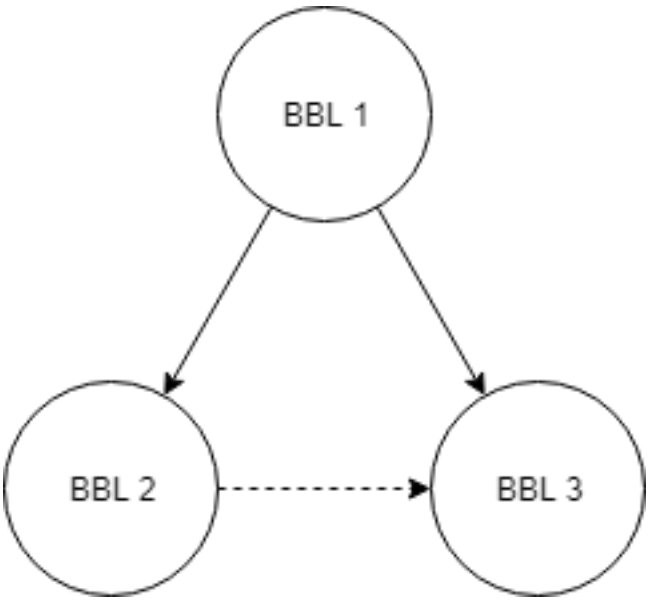

Figure 1.1: The control-flow graph for code in listing 1.1

1.1.2 Control-flow graphs

Control-flow graphs (CFGs) are the graphical representation of the series of valid transitions between basic blocks within an application, see figure 1.1 for a CFG of listing 1.1. It is conceivable that a CFG can become enormous as the size of an application grows, as the number of different valid paths along the graph will rapidly increase. (need to find reference for this) . Valid control flow is where the flow which follows a CFG for a given application, for example using the graph (figure 1.1) for the code outlined in listing 1.1 BBL 1 $\rightarrow$ BBL 2 and BBL 1 $\rightarrow$ BBL 3 are both valid flows, however BBL 1 $\rightarrow$ BBL 2 $\rightarrow$ BBL 3 is invalid. To create BBL 1 $\rightarrow$ BBL 2 $\rightarrow$ BBL 3 an attacker could change the value of `secondNum` to one which is greater than or equal to `firstNum` once the first `if` statement has been passed.

1.1.3 Types of branches

In order to understand control-flow it is important to understand the difference between the various branch types: direct, indirect, unintended and function returns.

Direct

A direct branch is one where the next instruction address is hard coded with the `jump` instruction, this means the execution will flow to a pre-defined location.

This can be violated by hardware attacks such as glitching, which forces the skipping of instructions, or memory attacks, where the destination address is swapped for another therefore causing the execution to flow along an unintended path.

Attacks on direct branches can often be overlooked by control-flow integrity (CFI) protection mechanisms as the requirements of needing direct access to memory or performing glitching attacks often lie outside of the scope of these protections, as they view attacks on indirect branches and function returns as larger threats.

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 might not appear as indirect in the source code, however when compiled has the possibility of indirectness. Indirect branches are usually targets of buffer overflows.

Unintended

De Clerq [2] describes an unintended 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 [3]. For more information see section 4.2 of Kayaalp's work [3] which describes unintended branches in detail. The threat of unintended branches is currently not such a problem in embedded systems as they tend to use fixed length instruction sets (such as RISC). However ARMv8, for example, has two sets of encoding which can be switched between, and as embedded systems' capabilities converge with those of general purpose computers we may see the introduction of variable length instruction encoding in the near future.

Function Return

When a function returns to its calling section of code the return address is retrieved from the stack and inserted in to the program counter (`pc`). This presents a problem as the return address is not hard coded and according to a CFG, a function could legally return to any of the locations in code which call that function. Tracking the entire path taken along the CFG allows the checking of calling location, making 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 $\oplus$ write). This is a well-explored area in computer security with various existing solutions including stack canaries, NX bit and Address Space Layout Randomisation (ASLR).

Break statements

Should these be included?

1.1.4 Loops

Loops are problematic in terms of CFGs. Modelling a loop which could run indefinitely is simply not feasible if full coverage is to be achieved. Various methods has been used to approach this, with some [4] simply compacting loops (this presents issues where an attacker can continually run a loop until they reach the desired outcome, such as performing an exhaustive search on passwords) while others [5] track loop metadata such as iteration count for each unique path taken through a loop.

1.2 Attacks on hardware

1.2.1 Introduction

Attacks on hardware are amongst the hardest attacks to mitigate against, as having direct access to memory or even cache memory can enable an attacker to bypass many protections (for example software-based protections). Hardware attacks can be used to capture the leaking of cryptographic keys, which presents a problem, particularly for attestation where a typical attestation report is signed using a signing key or when a symmetric key is used for encrypting or applying a MAC to the attestation report. The basis for this section stems from [6] where attacks on hardware which are applicable to embedded systems are well explored.

1.2.2 Invasive Attacks

An invasive attack occurs when a microprocessor is removed and attacked directly through physical methods. In theory any microprocessor can be attacked in this way however it does require expensive equipment and a large investment in time.

One example of an invasive attack is probing bus lines between blocks on a chip (with a hole being made in a chip's passivation layer). Here, secret information is derived by observing information sent from one block to another. An extreme example of an invasive attack is using a focused ion beam to destroy or create tracks on the chip's surface. This could be used to reconnect disconnected fuses (this is a threat to the use of PUFs (see 3.5.2 for more details) where fuses are used to deactivate PUF derivation circuits). Fuses can also be used to turn off test modes which are used to read/write to memory addresses during manufacture. Modern solutions avoid this vulnerability by removing test circuits from the chip when it is cut from the die during the manufacturing process [7],[8].

1.2.3 Semi-Invasive Attacks

Semi-invasive attacks require the surface of chip to be exposed, however the security is compromised without directly modifying the chip. Examples of semi-invasive attacks include observing electro-magnetic eminations using a suitable probe [9],[10], injecting faults using laser [11] or white light [12]. Numerous more have been discussed in literature [13].

Fault Injection

Variations in supply voltage [7],[14] may cause processors to misinterpret or skip instructions, this is applicable to control-flow as the misinterpreting or skipping of instructions are attacks used to subvert the control-flow of applications. Variations in external clock [7],[15],[8] can cause data to be misread (i.e. data is attempted to be read before memory has time to latch-out correct value). This can lead to missing of instructions, which can be an attack vector if software is written in an insecure manner such as aborting an operation if an `if` check is successful. Extremes of temperature [16],[17] can cause unpredictable effects in microprocessor. Two examples of effects obtained [11] are random modification of RAM cells due to overheating and read write temperature thresholds in most non-volatile memory (NVM) not coinciding. In this case if the temperature is set to a level where write ops work but read do not a number of attacks can be mounted. Laser light [18] can be used where light arriving on a metal surface induces a current, which if intense enough could induce a fault in a circuit. White light [7] has been proposed as alternative to laser-based attacks [12], however as it is not directional it is challenging to focus on a particular areas of a microprocessor and therefore provide a targeted attack. Electromagnetic flux [19] has been used to change values in RAM, where strong eddy currents can affect microprocessors although this has only been observed in insecure microprocessors.

```
int CheckNumberOfAttempts(int attemptCount, int attemptLimit)
{
    if (attemptCount => attemptLimit)
        return -1;
    ContinueLogin();
    return 1;
}
```

Listing 1.2: An example of code which is vulnerable to skipping of instructions

The effects of fault injection include resetting data where data is forced to a blank state, data randomisation where data can be changed to a new random value and modifying of opcodes where instructions executed on chip's CPU are changes[7] which often has the same effect as previous examples but additionally allows for the removal of functions and breaking of loops. Countermeasures given in [11]

1.2.4 Non-Invasive Attacks

Non-invasive attacks can be used to derive information without modification of hardware through information that leaks during computation of a given command, or attempting to inject faults in manners other than light. Examples include monitoring power consumption [20],[21] and injecting faults by glitching the power supply [7],[11].

1.3 Software attacks

Attacks on software represent a likely threat to the security of embedded systems as they can be carried out remotely or simply stumbled upon (fuzzed). Once an exploit is discovered it is highly repeatable and can be crafted into a number of different attack methods. Embedded systems can be more vulnerable than other computer systems due to the language used to write applications - C. As C provides the developer with memory operations it also presents the attacker with an opportunity to exploit any bugs in the code, usually manipulating memory.

1.3.1 Commons software attacks methods

The aim of a **buffer overflow attack** is to manipulate control-flow information stored on the stack and heap of a program in order to achieve further objectives, examples of such information could be return addresses or other variables on which the basis of decisions are made (e.g. `if` statements).

When performing an **injected code attack**, control flow is diverted to existing injected code (usually as data). To mitigate against this, NX bit can be implemented which marks data memory as non-executable.

Existing code such as system functions or sequences of code which, when combined, have unintended consequences can be be the target of a deviated control-flow. This is known as a **code re-use attack**. See also return oriented programming and return-to-libc attacks.

An attacker can string together (ordinarily benign) existing code sequences to form gadgets, which can result in malicious program actions. This attack is referred to as **return-oriented programming (ROP)**. This is done through changing the return addresses on the stack to point to each snippet of the code sequence.

An attacker replaces the return address on the stack to one which contains subroutines which already exist in memory (**return-to-libc**) such as functions which allow the execution of shell commands. `libc` refers to the C standard library.

Pointer subterfuge is where the value of a pointer is modified by an attacker, there are various methods of achieving this and various points of attack. The simplest example is overrunning a buffer which is next to data in memory which is later assigned to a pointer.

```
int main(){
  int *ptr = ...;
  int val = ...;
  char buf[16];

  gets(buf);

  *ptr = val;
  return 0;
}
```

Listing 1.3: An example of code which is vulnerable to pointer subterfuge [1]

Non-control data attacks involve corrupting data which is 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 (yet still allow an attacker to achieve their objectives), this is discussed in detail by Shacham [22].

1.4 Problem Description

1.4.1 Introduction

CFI has been a persistent problem in computer security, with examples of attacks including return oriented programming, stack overflows and hardware-based attacks (see 1.2). Several means of defence have been designed to address some or all of the problematic results (see 3.4). These include:

Prevention

Where violation of the control-flow integrity is prevented before it occurs. Methods include:

- Read $\oplus$ Write memory, which prevents the execution of data memory and the overwriting of memory containing executable instructions.
- Encrypted instructions, which enforce decryption based on a particular order of instructions following the application CFG.

Detection

Where processes are put in place to detect a violation of control-flow integrity and act upon detection, in this instance the intention is to catch the violation before too much damage has been done. Methods include:

- Stack canaries, which detect ROP attempts therefore preventing violation of CFI.
- Shadow stacks, which aid in preventing return-oriented programming by detecting ROP attempts and assist in preventing violation of CFI
- Software-based CFI, which adds checking of calling IDs between each basic block to ensure they are permitted within the CFG. This was first discussed by Adabi et. al [4] and has been subject to a large amount of subsequent research.

Attestation

Where a third party (verifier) receives a report, which could only have come from the device (prover), stating either that the control-flow integrity has not been violated or describing the taken control-flow (therefore allowing the verifier to compare against the CFG to be sure that control-flow integrity has been maintained). Example methods include:

- Basic-block IDs are hashed to create a hash chain representation of the control-flow path taken.
- Instructions are intercepted by secondary hardware monitor to take flow measurements, comparing to the valid instructions as described by the CFG.

Data-flow integrity

Data flow integrity [23] 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. inside an `if` statement), this can also be used to protect the stack. In [23] the solution 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.

Dynamic taint analysis [24] is another method used to detect and prevent control-data and non-control-data attacks by labelling data as untrusted and tracking that data to make sure it is not used to determine program execution. With dynamic taint analysis there is no need for source code however it does yield false positives, has a high overhead and requires additional hardware support as noted by [23].

1.4.2 Auditing control-flow

CFI prevention and detection has proven to lead to a slight computational overhead [2], while attestation requires an always-on connection. What if we can store the control flow of an application on a hard disk, which can be retrieved later? This could be useful to prove to a third party that a disputed result of a computation is correct (or at least the steps taken during the computation was correct), or providing a record of the control-flow which has been taken during the execution of a vitally important operation.

Keeping audit records of control-flow could also be useful for spot-checking inaccessible devices which have no internet connection, such as industrial control systems (ICS) components or medical devices which can only be communicated with during medical check-ups.

So, while we would not provide immediate detection or real-time attestation, the contents of the audit file can be compared with the CFG for an application at a later time. This could also be useful for reducing computing requirements as comparisons could be appropriately scheduled for the efficient use of processor time, rather than being performed on an ad-hoc basis as required by some attestation methods.

1.5 Project Objectives

The objectives of this project are to investigate and implement a solution for audit of the control flow of applications.

The solution will statically attest the application before including this in the result in dynamic results of the control-flow monitoring. It will store this result and a signature / MAC on the long-term storage of the device.

1.5.1 Introduction

In this section we will define the requirements which we will aim to meet with the solution produced as a result of this project. These have been built up using the literature for CFI and attestation methods as well as the requirements identified from the attack model.

1.5.2 Practical Requirements

1. Compatible with compiled binaries: where the original source code does not need adjusting to be compatible with the solution.
2. Compatible with external libraries: similar to compatibility with compiled binaries but with the added constraint of no access to the external library source code.
3. Compatible with embedded systems: a crucial element as the solution is aimed at solving the problem for embedded devices.
4. Granularity: the solution should (at a minimum) be able to keep track of transitions between basic blocks (including direct branches).

1.5.3 Security Requirements

1. **An attacker must be detected if they change the sequential order of instructions.** Re-ordering instructions can be used to bypass cryptographic protections (similar to ECB vulnerabilities) to achieve the attackers goals.
2. **An attacker must be detected if they change the execution order of basic blocks.** Re-ordering BBLs can be used to execute code in a manner which violates the control-flow graph.
3. **An attacker must be detected if they skip instructions** Skipping instructions can be used to bypass checks.

4. **An attacker must be detected if they subvert control flow to existing code (e.g. return-to-libc)** Subverting control-flow to external existing code could enable an attacker to perform functions not permitted through normal usage of the application.

1.5.4 Attacker Assumptions

We assume that attacker may have direct access to memory but not internal registers, they cannot break cryptographic protections and that cryptographic keys can be safely stored. This level of attacker has been referred to as the Dolev-Yao attacker model [25] by [26].

1.6 Project Structure

This project will be broken down into three sections. Chapter 2 describes Control-Flow Integrity including construction of control-flow-graphs, the prevention of CFI corruption, the detection of CFI corruption and how CFI is attested. Chapter 3 looks into the background existing literature surrounding the subject. Chapter 4 identifies some key requirements we need and compares the existing solutions against them. Section 5 will describe the theory behind the solution while section 7 will describe the practical implementation . In chapter 6 we will perform a security evaluation on the proposed solution. Chapter 8 will provide the conclusion and a recommendation for directions for further work.

Chapter 2

Embedded Systems and Control Flow Integrity

2.1 Important Principals

This section provides an insight into the key principles around which this project is focused.

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.

2.2 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.

2.2.1 A Definition of Embedded Systems

Embedded systems are small-form, low power computer systems. A useful way of defining them is to compare an embedded system with a PC [79]:

- Dedicated to specific task (PCs are generic computing platforms) where 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 it has an OS it will be RTOS. It should be noted that with the introduction of IoT devices this has become less of a constraint, with standard Linux distributions such as Ubuntu becoming more prominent.

- 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.
- Software is often stored in ROM.
- Require specialized tools and efficient design methods.
- Often have dedicated debugging circuitry.

2.2.2 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, for example automotive systems.

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.

2.2.3 Common Implementations of Embedded Systems

A large number of manufacturers have an interest in embedded systems. A big player is ARM with their Cortex-M Series of CPUs, one example of these being used is in the Pebble smartwatch[1].

Another common base for embedded systems 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).

2.3 Introduction

A Control-Flow Integrity (CFI) security policy is one which states that software execution must follow a path of a Control-Flow Graph (CFG). CFI can be provided in a number of ways including signature modelling, shadow stacks and through the enforcement of basic block transitions along a CFG.

2.3.1 Control Flow Graphs

Abadi et al. [4] introduced CFI as a method to protect the secure execution of software, using static analysis of an application binary to create a Control Flow Graph (CFG). Only control flow transfers which follow the CFG are permitted.

Clerq and Verbauwhede [2] posed CFGs as part of a solution for tracking instructions causing control flow transfers, however Lee et al. [26] note that they do not focus on sequential transitions. They argue that the method used in [2] (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. They state that when implementing a scheme, instruction-dependant transitions are

[1] `https://www.ifixit.com/Teardown/Pebble+Teardown/13319#s45416`

simpler to process than instruction-independent transitions. SEP [26] and SOFIA [27] go one step further to provide integrity protection for each individual instruction.

Adabi et al. [4] state that CFGs can be defined by analysis-type methods or explicit policies. Examples of analysis-type methods include source-code analysis, binary analysis or execution profiling. An example of an explicit security policy method is writing as security automata as seen in [28]. What is this???

CFGs have been used to provide protection against soft faults (single-event upsets) using software based methods [29],[30],[31]. The solution put forward by Adabi et al. [4] restricts control flow through in-lined labels and checks. The CFG is embedded through a set of static, intermediate bit patterns in program code. The problem with this is that they are evaluated at the destinations of all branches and jumps but not at the sources. These fail to prevent jumps into the middle of functions (e.g. ones which bypass security checks such as access control).

The method described in [4] ensures that whenever an 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. Dynamic checks are enabled through the the use of static analysis and machine-code rewriting, forming an instrumentation process. Machine-code is rewritten using modern tools for binary instrumentation in order to overcome the resulting new memory addresses which are the result of the rewriting process [32],[33].

Generating a CFG from static analysis is troublesome and often an over-approximation is used which is not fully precise [34],[35], in their survey [36] classifies the precision of computing CFG using different static analysis techniques.

2.3.2 Hardware-based CFI

Hardware-based CFI is provided by the addition of dedicated hardware which is either included in the execution pipeline or intercepts the instruction flow from memory to the processor. We provide a survey of leading solutions in 4.1.

2.3.3 Software-based CFI

Software-based CFI was first introduced [4], which has been the foundation for further research surveyed in [36], which provides a detailed insight into the technical properties of a number of software-based CFI solutions.

2.3.4 Alternatives to Basic Block CFG-based CFI

Signature modelling

Here if a series of executed instructions do not follow the trace of the programs CFG the control-flow integrity has been lost. Yang et al. [37] describe this method as implemented in software. They also performed a thorough analysis of various existing solutions. The methodology of signature modelling was implemented in hardware in the paper written by Werner et al [38].

A problem with signature modelling is that the number of unique sequences of instructions (and their accompanying signatures) accumulate very quickly, requiring more storage if they are to be compared against. This is a particular problem when considering embedded systems due to their inherent limitation on resources. Another problem is the necessity to compute signatures in real-time, which would require additional processing and thereof an increase in processing overhead. Although, as we have seen in other methods [39] [26], decryption is a common method of ensuring CFI, so perhaps this is not an unreasonable expectation.

Verification on transition

Shadow Stacks

Some [40] have argued that ensuring CFI without the use of shadow stacks (SSs), also know as shadow call stacks (SCSs) is an impossible task and that"the use of a shadow stack is mandatory for any practical CFI deployment". A shadow stack consists separate stack, stored in memory (protected or unprotected, depending on the solution), which contains just the return addresses. This solves the problem of ROP, especially when working with commonly called functions or system functions as the implementation of the shadow stack ensures the calling function also has to be the return function (at least until `longjmp` is used, which adds a further complication). There are some flaws with shadow stacks which include:

- Storage of shadow stacks can be cumbersome - especially if they are to be secure (as they would have to be if memory access is an assumed capability of an attacker).
- Maintaining a shadow stack in multi-threaded systems will be troublesome - searching through the SS for a process ID will add additional time overhead to any processing.
- Recursive functions can add multiple entries of the same return address to the SS, this is an unnecessary addition so is reduced down to a single bit flag (or counter) by some solutions [40]. This can raise the problem of manipulating the number of recursions by an attacker, however it is argued by some that this is a very limited attack. Ref for number of iterations not being bad?
- Instructions such as `longjmp` will skip back through multiple return addresses, so this needs to be taken into account - either by removing the instruction from the ISA or another method such as iterating through the SS until the return address is identified (with the above addresses then removed from the SS to maintain a proper representation of the stack).
- It has been found that shadow stacks can introduce an overhead of up to 13% [41], which when compared to the overhead values of 5-10% described in [42] as the limits for adoption proves to be problematic.

Other alternatives

Branch limitation is an alternative to CFG [43] [44]. Another alternative is Code Pointer Integrity described in [45] and implemented in [46],[47] and [42].

Generalised path signature analysis is described by [48] while [38] implemented it using an ARM Cortex-M3 microprocessor.

Chapter 3

Literature Review

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 [49] focusses on existing mature solutions, while others [50], [51] and [52] 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 [53] 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, 3.4.2 and 4.3.1.

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 [49]. 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 [54]). 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 [50], 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 [55].

A similar survey [51] 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 [52]. 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 [38] and Assertions for Control-Flow Checking (ACFC) seen in [56].

3.2.3 FPGA Security

An excellent high-coverage survey on FPGA security is provided in [53], 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 [57]. 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 [58], 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 [59].

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 ([54], [60], [61] and [62]) and software or secure software execution 3.4.2 ([38], [63], [64] and [65]). Hybrids of the two approaches are presented in [66] and [67].

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 [54]. 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 [54] 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 [54] 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 [75].

The second piece of work we consider is [60] 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 [60] 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 [60] 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 [66] has not been published in a well-established journal however the authors have been invited to present their findings in [76]. 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 [66] 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 [67] 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 [61] 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 [62] 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 [38] 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. 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. 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

The second paper [63] (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 [64] 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) such as read $\oplus$ write and a secure trust anchor that provides an isolated measurement engine and can generate the fresh authenticated attestation report.

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 [65] 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 [56], 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 [77] 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 Physical Unclonable Functions (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 [78], 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 ([54], [60], [66], [67], [61] and [62]) 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 and attestation of code. Finally we looked at two primitives which were prominent in many of the solutions examined.

Chapter 4

Comparison of Solutions

4.1 Comparison of solutions

4.1.1 Established Criteria

We will consider each solution described in 4.1 against a set of requirements closely resembling those defined in 1.5.1 and present the results in table 4.1.

4.2 Requirements

(1) Works with compiled binaries: software does not need to be recompiled to be compatible with the solution.

(2) Works with external libraries: libraries can be used without need access to their source code.

(3) Can be used to bind software and hardware: would allow for future functionality of binding software to hardware.

(4) Works with embedded systems

(5) Immediate identification: whether the solution can identify a breach in CFI as soon as it occurs.

(6) Not reliant on TPM: Embedded systems are resource-constrained and therefore may not have TPMs.

4.3 Properties

(1) Hardware based: The solution is implemented through hardware.

(2) Software based: The solution is implemented through software.

(3) Source code modification required: Source code needs to be changed for implementation.

(4) Granularity: How granular the control-flow is examined, e.g. individual instructions or basic blocks.

(5) Provisioning method: How software is updated.

(6) How it is secured?: What is providing the security.

(7) CPU Overhead

(8) Storage overhead

(9) Memory overhead

(10) Execution time overhead

(11) Compatible architectures

(12) Action after identification of a CFI violation

(13) Prevention / Detection / Attestation

(14) CFG or other: Whether the solution makes use of CFGs

Introduction

Here we will provide an in-depth review of a number of control-flow integrity and control-flow attestation mechanisms.

C-FLAT

C-FLAT [64] uses ARM TrustZone to facilitate control-flow monitoring and attestation. Binaries are re-factored to enable normal-world programs to log control-flow changes with a measurement engine operating within the secure world (via trampolines set up in normal world). Control-flow edges are hashed together to form a hash of the complete control flow. Loops are handled as their own sub-program which require separate meta data to be gathered and included in the attestation report. This solution requires code instrumentation but also requires programs to not use `LR` as a general purpose register, where the use of the `-ffixed-lr` compilation option in GCC[1] is recommended, for example.

LO-FAT

LO-FAT [5] implements a hardware-based solution which monitors control-flow instructions by collecting the program counter and instruction executed for each clock cycle, then filtering out branch, jump and return instructions. LO-FAT follows a similar method as C-FLAT where a hash is built up of each control-flow operation, along with special meta data for loops. It uses additional hardware components (designed onto an FPGA - Virtex-7 XC7Z020) to operate alongside the main processing unit. Through the use of these components (almost co-processors) the solution is able to run alongside normal operations, therefore removing the burden of additional instruction executions and the requirement for software instrumentation (a process of changing the contents of software to meet the requirements of the solution).

Secure-Execution Processor (SEP)

SEP [26] uses encryption of each instruction which takes the previously executed instruction as an input to the decryption algorithm. Decryption is performed in the processor pipeline which therefore prevents the malicious diversion of control-flow, if an attempt is made to alter the control-flow the processor will attempt to execute a garbage instruction, the result of which will be to hit the 'kill switch'. The solution goes into great detail in how to deal with particularly troublesome instructions. Unfortunately due to the large memory and execution overhead (235%), the limited instruction set and the use of a slow encryption mechanism, this solution still has a long way to go before it can be properly considered, however the principal and elegance of the solution is promising. This solution requires code instrumentation in order to deterministicly encrypt each instruction.

[1] `https://gcc.gnu.org/onlinedocs/gcc-4.6.1/gcc/Global-Reg-Vars.html`

CCFI-Cache

CCFI-Cache [81] implements a hardware-based solution where control-flow metadata (including the number of instructions in the BBL, the valid destination addresses for the current BBL and a hash value of the instructions of the contained within the BBL). This is fetched by the cache (CCFI-cache) and checked by the checker (CCFI-checker). The metadata and BBLs are padded out to ensure that their length matches (empty regions for the metadata and nops for the BBL), this enables the checker to know if either the metadata or BBL has been modified in a manner which affects the length. The checker checks that the destination of a BBL matches the pre-computed metadata and that the hashed instructions within the BBL match the stored hash. It also utilises a shadow stack to ensure the correctness of return addresses. If a violation is detected an interrupt on the CPU will be triggered. In this author's opinion this solution provides the best coverage for protecting control-flow integrity. This solution requires code instrumentation.

HCFI

HCFI [81] implements a shadow-stack on additional hardware. It does not describe how it ensures integrity of forward edges but it does provide a thorough analysis of backward-edge protection through the use of a shadow stack. It is not clear if this solution requires code instrumentation or not.

HAFIX

HAFIX [39] (like HCFI [81]) focuses on backward-edge protection through their implementation of a shadow stack. HAFIX and HCFI fight it out over how to deal with recursive functions etc. This solution requires code instrumentation.

SOFIA

SOFIA [39] works in a similar manner to SEP [26], where instructions are decrypted using program counters of previously executed instructions. SOFIA handles instructions with multiple callers by creating a multiplexor block, which allows two predecessors, so multiple predecessors can be used by creating layers of multiplexor blocks. The fast expansion of multiplexor blocks could prove problematic in terms of size and operating time of applications. The solution requires code instrumentation (for creating the multiplexor blocks and encryption of instructions).

Strategy without tactics (Sullivan)

In Strategy without tactics [39], the author uses trampolines to deal with multiple callers, where a single destination is replaced with a trampoline. Through adding a trampoline the solution is able to transform a call source/destination pair into a unique call source/trampoline pair which solves the problem of basic blocks with a number of possible predecessors. From the trampoline, a direct jump is issued to the original destination. A label state stack records backward-edges and a label state register records forward-edges.

Learnings

It has been noted on several occasions that backward edges cannot be fully protected without the use of shadow stacks - as returns may be legal for many functions (e.g. `printf`). Keeping an audit record of control-flow would eliminate the requirement for a shadow stack as the historic tracking of the control-flow would indicate if a backward edge has been exploited.

Table 4.1: Requirement Comparison of Control-flow Integrity and Control-flow Attestation

Scheme	Requirements						
	(1) Works with compiled binaries	(2) Works with external libraries	(3) Can be used for binding	(4) Works with embedded systems	(5) Immediate identification		(6) Not reliant on TPM
C-FLAT [64]	●	●	○	●	ARM Cortex A	○	○
LO-FAT [5]	●	●	○	●		○	●
SEP-LEE [26]	○	○	●	●		● On jump or return	●
CCFI-Cache [81]	○	○	○	●		○ End of BBL or number of executions reached	●
HCFI [40]	○	○	○	●		●	●
HAFIX [39]	○	○	○	●		●	●
SOFIA [27]	○	○	●	●		●	●
Sullivan [82]	○	●	○	●		●	●

● = meets requirement; ◐ = partially meets requirement; ○ = does not meet ; - = not enough information

Table 4.2: Comparison of Control-flow Integrity and Control-flow Attestation schemes properties

Scheme	Properties 1-5				
	(1) Hardware-based	(2) Software-based	(3) Source-code modification required	(4) Granularity	(5) Security provided by
C-FLAT [64]	●	●	Control flow instructions re-written to go via trampolines	Basic Block	TPM and Digital Signatures (for report)
LO-FAT [5]	●	○	○	Basic Blocks	LO-FAT Memory is protected
SEP-LEE [26]	●	○ Reassembly	●	Sequential or BBLs	PRINCE
CCFI-Cache [81]	●	◐	●	Sequential (Only number or hash of instructions)	Meta-data in ROM, Hardware Security for controller
HCFI [40]	●	○	●	BBLs	-
HAFIX [39]	●	○	●	Backwards edge	-
SOFIA [27]	●	○	●	Sequential instructions or flows	Encrypt+MAC (RECTANGLE-80)
Sullivan [82]	●	○	●	BBLs	Attacker cannot modify code

● = has property; ◐ = partially has property; ○ = does not have property; - = not enough information

Table 4.3: Comparison of Control-flow Integrity and Control-flow Attestation schemes properties

Scheme	Properties 6-14								
	(6) CPU overhead	(7) FPGA area overhead	(8) Memory overhead	(9) Time overhead	(10) Compatible architectures	(11) Action on identification	(12) Prevention or detection or attestation	(13) Uses CFG or another method	(14) Success rate
C-FLAT [64]	-	-	-	0.03 - 0.13%	32-bit RISC	-	Attestation	CFG	-
LO-FAT [5]	0%	20%	-	0%	PULPINO RISC-V	-	Attestation	CFG	-
SEP-LEE [26]	-	-	235% (160 vs 68 operations)	235% (160 vs 68 operations)	SEP (based on Picoblaze but reduced from 18 bit to 16 bit)	Kill switch	Prevention and detection	Sequential or BBL	-
CCFI-Cache [81]	-	10%	x2 + 9-30% for nops	2-63%	PicoRV (Implementation on RISC-V ISA)	Interrupt	Detection	CFG + No of instructions + hash of instructions	Small number of ROP and JOP attacks will still work
HCFI [40]	-	2.5%	-	1-6%	Leon3 / SPARC V8	Interrupt	Detection	CFG and Shadow Stack	-
HAFIX [39]	-	2-8%	-	2%	Intel Siskiyou Peak, SPARC	-	Detection	CFG	80%
SOFIA [27]	13.7% Cycle overhead, 84.6% Clock speed reduction	28.2%	-	110%	Leon3 / SPARC V8	Processor reset	Prevention and detection	Sequential and BBLs	-
Sullivan [82]	-	1.78%	-	1.75%	Leon3 / SPARC V8	Termination of process	Detection	CFG	Dependant on CFG policy (assumed 100%)

● = has property; ◐ = partially has property; ○ = does not have property; - = not enough information

4.3.1 Attestation of code integrity

SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust

SMART [68] provides a simple method of attesting the state of a user application through slight hardware alterations. As an input from the verifier it takes a nonce, start memory address and end memory address (as well as additional parameters for an optional start position of user application). The prover replies with a SHA-1 HMAC of the requested code space.

Important capabilities required by SMART are:

- Secure HMAC key storage - this has not been directly addressed here however options are presented such as the key being hard-coded at production time (and never changed again) or by implementing a secure means of modifying by an authorized party, but not reading (as in the key can only be read by SMART). The second option was deferred to being future work.
- Secure HMAC key access - only SMART code on ROM can access the HMAC key. This is provided by registers only allowing access to the HMAC key while the program counter is located within SMART application space.
- It is not possible to enter or exit SMART instructions other than in the beginning or end. For example, if the program counter (`pc`) is outside of SMART its previous location must also be outside of SMART (or the last instruction), and if the `pc` is inside SMART its previous location must be at the start or also within SMART.
- Smart code is not editable, it is stored on ROM.

SMART was implemented on low-end microcontroller units (MCU)s such as MSP430 or AVR. After implementation it was realised that only the memory access controller needed to be modified, therefor making the solution possibly compatible with black box processors such as low-end ARM cores.

To critique the method - it uses SHA1 which is now outdated. It also could succumb to cold-boot attacks but the authors state that due to the typical MCU design where the processor and memory are a single package meaning that memory could not be accessed directly.

VIPER: Verifying the Integrity of PERipherals Firmware

VIPER [69] is a software only solution designed to provide attestation for peripherals firmware, where the host CPU queries peripherals. It uses combinations of checksums, hashes and time-frame based checks to ensure peripheral firmware is correct. The method consists of two parties: the verifier (host CPU) and the prover (peripheral). The verifier will have access to a "checksum simulator" which contains copies of correct firmware for each peripheral and is able to generate random numbers. The checksum simulator is first invoked, generating challenges (nonces) and the expected correct response from the prover by simulating the verification procedure on the correct peripheral firmware. This process is used to gain confidence that the verification/hash code is correct on the peripheral. The verifier then starts the process with the peripheral, requesting that the peripheral resets itself to a known-good state. Then the verifier starts a timer and sends the challenges, the peripheral then calculates the checksum over its checksum and hashing code and returns the result to the verifier. Finally the peripheral will compute the hash over the full contents of its memory, which is then sent to the verifier program for validation.

A significant proportion of this solution is focused on how latency needs to be taken into account for a live attestation protocol as the timing of the checksum calculations need to be precise enough to prevent the use of a proxy device to generate valid responses. This is also an important consideration when dealing with peripherals with different processing capabilities (consider a keyboard vs. a graphics card) as the response time will be significantly different. As this solution carries out attestation on multiple devices it is stated that it will attest high powered peripherals first, with slower peripherals subject to attestation in the later stages of the process. Additional methods are used to ensure use of a proxy is detectable such as increasing the proxy communications overhead and implementing continuous checksum computation. As much as the

solution aims to address the latency and proxy system issue, time/delay-based protocols are still vulnerable to exploitation [74], though it must be noted that the solution presented in [69] presents some promising protections against such attacks.

The paper [69] also provides an excellent description of the principles of designing a checksum function for the prover. These principles include using all available registers as part of the checksum calculation, the checksum should follow a pseudo-random pattern when reading from memory (preventing predictability for any attacker) and that the program counter (`pc`) should be included in the calculation (if accessible).

Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems

Pioneer [70] is a software-only solution which works in a similar way to VIPER but is designed to attesting legacy systems. A checksum is calculated throughout the execution of the verifying (testing mechanism) code, the resulting execution time is compared to an expected execution time known to the verifier. This proves that the testing mechanism is correct. The testing mechanism then builds a hash of the target code.

SWATT: SoftWare-based ATTestation for Embedded Devices

SWATT [71] is a software-based solution which uses a random number generator to generate random memory addresses which are attested - this means an attacker cannot predict which regions they need to leave intact. They also use time measurement to ensure an attacker is not manipulating the results. This author questions whether that time-based checking is a reliable component, especially when attestation is taking place remotely, for example, over a network. Do I care? This paper also references solutions which use secure coprocessors which are used during system initialisation to bootstrap trust, examples of these are TCG (Trusted Computing Group, formerly known as TCPA) and Next-Generation Secure Computing Base (NGSCB, formerly Palladium) which may be of interest at a later point as we will be able to see if they produce a useful output after startup has occurred.

Software-Based Remote Code Attestation in Wireless Sensor Network

AbuHMed at al [72] (again software-based) focusses on filling empty memory with predictable contents (such as ciphertext generated by the verifier) in an attempt to prevent an attacker from utilizing free space for malicious instructions.

4.3.2 Dynamic attestation of code integrity

Remote Attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence

Kil et al [73] aim to solve a different problem of attestation - that of dynamic attestation. It uses various different components to measure run-time statistics such as data invariants. They state that they aim for full coverage of the source code rather than all execution paths, the problem of achieving coverage is described as the equivalent to solving the halting problem. By being heavily interwoven with the Linux kernel they track all system calls. It heavily relies on the TPM to do the heavy lifting of signing attestation reports and isolating records of violations. This solution utilises a Merkel hash tree for use in authenticating input files for the proc file.

4.3.3 Attacks on software-based static attestation

On the Difficulty of Software-Based Attestation of Embedded Devices

This paper [74] that software-based attestation is a problematic solution. They attack SWATT [71] by adding a hook into the attestation code which swaps out malicious code to data memory, as a result they recommend covering all memory in attestation processes (which they note would not be easy in embedded

systems). They find that the time overhead relied upon by SWATT [71] is imprecise and that their attacks fit within the tolerance. The paper also attacks solutions which fill empty memory with noise, as they found they could compress legitimate applications to make room for malicious ones.

4.3.4 Conclusions on attestation

To provide attestation one usually requires a nonce and an area of memory to attest. The problem of offline attestation (audit) is that a previous good report could be cloned. To overcome this we should consider sequence numbers - where they need to be unique each time the process occurs, these would be stored alongside the HMAC prior to being encrypted. Another option could be using a time-stamp, however the time value would have to be strongly protected and secure in order to prevent an attacker from changing the time to the desired time when the attack is planned in order to create a "good" report. Chaining reports together may provide protection against this.

It seems that a large amount of the focus of the software-only mechanisms focus on proving the integrity of the verification software, which is a legitimate concern. Secure operating conditions or self attestation should be able to provide for this.

A key learning from the solutions and the attack described is the importance of attesting all parts of memory.

Chapter 5

Proposed Solution

5.1 Building CFGs

5.1.1 The problem behind building CFGs

C-FLAT [64] notes that efficiently computing a generic program's CFG, and accompanying exploration of all possible execution paths is an open problem. However it is also noted that static embedded application software is typically far simpler and it is therefore possible to reasonably compute a CFG.

5.1.2 Existing tools

LLVM

LLVM provides various methods of building CFGs which could be of use, for example:
`https://llvm.org/docs/Passes.html#dot-cfg-print-cfg-of-function-to-dot-file`,
`https://llvm.org/docs/Passes.html#simplifycfg-simplify-the-cfg`,
`https://llvm.org/docs/Passes.html#view-cfg-view-cfg-of-function`

Building CFGs in LLVM is not simple and is not a main feature of LLVM. The dot format is widely used in representing CFGs and further research is required to find whether it can be made to be compatible with control-flow tracing.

Vulcan

Vulcan [32] is used by [83] and the seminal work [4]. Though it has since been taken out of production by Microsoft and is currently unavailable.

Jakstab

Jakstab [84],[85] can also be used to create a dot file containing the CFG of x86 executables, so may need enhancement to support instruction set architectures (ISAs) commonly used in embedded systems such as ARMv8-M or RISC-V. It was successfully used in [86] to construct CFGs for x86 Complex Instruction Set Computer (CISC) instructions. The last development activity on this was 31st March 2017. The tool's author has been trying to improve on the tool with the aim of creating under-approximated CFGs rather than over-approximated ones (which can lead to vulnerabilities in resultant implementations) [35].

Others

In C-FLAT [64] the authors create their own analysis tool to achieve their branch-based requirements rather than create entire CFG models. They also use Capstone disassembly engine [1] however this is more for use dissasembling code rather than CFG building.

LiteHAX [87] uses the 'angr' [2] [88] framework to generate the CFG outputting a networkx [89] DiGraph. This project is still under active development. The output of this process would need to be compatible with the dynamic attestation. angr offers two types of computation methods - CFGFast (a static CFG) and CFGEmulated (a dynamic CFG). During assessment of angr, utilising networkx utilities, we have found that only single edges are identified. For example 5.1

```
<CFGNode rejected+0x16 0x400713[10]> <CFGNode 0x400570L[6]> {'jumpkind': 'Ijk_Call', '
    ins_addr': 4196115, 'stmt_idx': 'default'}
<CFGNode main+0xa0 0x4007bdL[10]> <CFGNode accepted 0x4006ed[14]> {'jumpkind': 'Ijk_Call', '
    ins_addr': 4196285L, 'stmt_idx': 'default'}
<CFGNode main+0xa0 0x4007bdL[10]> <CFGNode main+0xaa 0x4007c7L[2]> {'jumpkind': 'Ijk_FakeRet
    ', 'ins_addr': 4196285L, 'stmt_idx': 'default'}
```

Listing 5.1: Example of results from angr CFG analysis of fauxware in form of .edgelist output from networkx utilities

IDA Pro is a popular tool for reverse-engineering which is able to create CFGs, however due to its high licence cost we have been unable to evaluate its suitability. The high cost also prohibits its utilisation as part of the tool chain.

The recently released (March 2019) Ghidra [3] leverages CFGs to assist in carry out its core function as a reverse-engineering tool, and while we have not found a method of extracting the CFGs created, this may be possible in the future as development is ongoing.

Control-flow analysis techniques used by [4] are described in [77],[90],[91] which run at compile time. A list of tools used by the solution proposed in [4] are described in further detail in [32](Vulcan), [77],[92] [93],[91].

5.2 Tracing Control-Flow

5.2.1 Using existing hardware

Recording of every executed instruction is infeasible as the overhead added would be too large if no additional hardware is included, as each instruction processed would require a corresponding context switch and further processing. If each instruction is hashed in real time the computational overhead would increase or if each instruction is recorded in storage prior to processing the memory/storage/IO capacity would see a significant overhead increase. The use of basic blocks allows us to deal with multiple instructions at a time, and therefore reduce the potential overhead dramatically.

CFG granularity can be broken down into three levels [64]:

1. Entire functions
2. Basic Blocks ending in a direct branch (this would cover most ROP attacks)
3. Basic Blocks ending in any branch instruction, this would provide the highest level of detail

To achieve the best coverage for the solution level 3 (Basic Blocks ending in any branch instruction) would be the optimum granularity to aim for.

We can build up a hash of the flow by taking the ID of the source node and hashing that with the previous hash. At the start (ID_1) the previous hash would be replaced with zero or a nonce. With the resulting chain appearing as: $H_1 = H(ID_1, 0)$ (or $H_1 = H(ID_1, nonce)$), $H_2 = H(ID_2, H_1)$ ... $H_n = H(ID_n, H_{n-1})$.

[1] http://www.capstone-engine.org/
[2] http://angr.io/
[3] https://ghidra-sre.org

The solution would implement a dedicated method in the secure world where a save is triggered. This will enable software authors an opportunity to save at key points - for example after a key decision is made. Hashes could also be stored along with variables used if we are able to implement that.

BLAKE-2 [4] has been suggested as a lightweight hash function, however this is only one such potential candidate.

Challenges of CFG tracking

Loops are a considerable issue when it comes to tracking control flow, as the number of possible paths could be overwhelming, especially if the loops contain branch instructions or function calls.

To overcome the problems presented by loops, C-FLAT [64] begins with treating loops as sub-programs. Where each run of a loop has a hash derived from it, and the total number of each time that hash was calculated is stored (therefore storing the number of times that exact sequence occurs). The result of this is the growth of the stored audit (as each loop will need to be referenced separately), when tested by C-FLAT was 1527 bytes for a path containing 16 loop invocations and 18 unique loop paths (this could probably be calculated at compile time, with a warning presented to the developer). Another path containing 12 loop invocations and 14 unique paths resulted in a 1179 byte result. Another second study (which is not as well documented by the paper) produced a maximum result of 490 bytes. The previous hash is also included to show where the was called from.

Return call matching is used where call and return edges are indexed during static analysis, so only valid routes are authorised. Break statements need to be dealt with in a special manner as they are an additional place where a loop can exit. Section 4.2 of C-FLAT [64] describes this in detail.

Instrumentation

Branches are identified, with direct jump addresses stored in a branch table then replaced with the address of the trampoline assembly code which, when executing, triggers the measurement engine to be add the source address and destination address to the hash chain before redirecting the program counter to the corresponding entry on the branch table.

Trampolines

A trampoline enables redirecting execution to a single piece of simple and secure code without needing to add too many instructions during instrumentation. A trampoline can carry out a number of functions but in C-FLAT [64] it manages the return address register as well as the register holding the destination address in indirect branches.

Hardware-assisted

A co-processor designed specifically for the purpose of providing audit of control-flow would offload the heavy lifting and therefore have a negligible impact on normal operation. Examples of designs which could be tailored to provide auditing are provided in [2]. A hardware-assisted solution could keep track of instructions executed (in the form of a hash), to add further granularity.

5.3 Contents of audit files

5.3.1 Other running processes

When implemented on an OS, the save will query the running processes from the OS and add them to the control-flow manifest. Query process IDs will be a different process between operating systems, so we will evaluate which OS will potentially be used prior to finalising the process. While there has been some work

[4] `https://blake2.net`

done in regards to enabling CFG to work with operating systems it will need investigating prior to any further planning.

Variables

The solution will implement a function where the software can assert (send data to secure world) variables currently in use, either when saving or as an additional instruction through the solution's API. External data or internal data will be tracked using asserts.

Previous hash

A hash chain will be built to prove the control-flow was tied to the previous control flow and the static attestation report.

Security

The resulting audit files will either be signed or have an HMAC created for them, they will be encrypted depending on requirements. A chain of files may be built up by including hashes of the previous file in each audit file, with the first also containing the initial attestation report.

5.4 Provisioning

5.4.1 Instrumentation

The existing binary will be instrumented either prior to loading or during the loading (bootloading process).

5.4.2 Initial attestation

All software will be scrutinised with an initial attestation upon initial commissioning.

5.4.3 Secure world

Secure world applications need to be signed before they are loaded as per the design of ARM TrustZone

5.5 ARM TrustZone

5.5.1 An introduction to ARM TrustZone

ARM TrustZone is a method used to separate secure applications from non-secure ones in two separate worlds, where memory can be designated as *secure* or *not secure*. Execution is transferred from each world either through interrupts or direct function calls. ARM TrustZone is very well described by Ngabonziza et al [80].

Secure world TrustZone applications have access to normal world memory, this will enable static attestation. A secure world application would need to supply the normal world application with a nonce to make sure the response is not a replay.

Chapter 6

Security Analysis

6.1 Security Analysis

The solution presented fits with the security requirements in the following ways:

1. **Works with compiled binaries**: The solution works with compiled binaries, where the binaries go through an instrumentation process.
2. **Works with external binaries**: As with compiled binaries, external binaries will have to undergo instrumentation.
3. **Works with embedded systems**: ARM TrustZone has been implemented on Cortex M processors which are widely used in embedded systems. Key storage and external storage will need to be in place for an embedded system to make use of this solution.
4. **Granularity**: The solution provides granularity down to a basic block (BBL) level.

Obvious vulnerabilities in the solution are the breach of the 'secure world' where the measurement engine is manipulated to create 'good' attestation files, deletion of files and resulting loss of audit capabilities (a backup method would somewhat alleviate this problem) and cloning of audit files. Cloning of audit files represents a significant problem due to the fact that this is an offline process, so a method will need to be used to ensure freshness of files, for example a tamper-proof date-time stamp.

Granularity to an instruction level can only be provided by a hardware solution which intercepts each instruction.

Chapter 7

Practical Implementation

7.1 Practical implementation

Complete software solution, i.e. assertions to secure world OS? Such as open tee...
Return address monitoring - how do they instrument with shadow stacks?

Chapter 8

Conclusion and Further Work

Bibliography

[1] W. Wang, "Non-Stack Buffer Overflow and Pointer Subterfuge The Memory Layout of Process in Linux," pp. 1–17, 2016.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] K. E. Mayes and K. Markantonakis, *Smart Cards, Tokens, Security and Applications*. Boston, MA: Springer US, 2008, ISBN: 9780387721972.

[7] 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.

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

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

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," *OSDI '06: 7th USENIX Symposium on Operating Systems Design and Implementation*, pp. 147–160, 2006. [Online]. Available: http://delivery.acm.org/10.1145/1300000/1298470/p147-castro.pdf?ip=219.93.16.130%7B%5C&%7Did=1298470%7B%5C&%7Dacc=ACTIVE%20SERVICE%7B%5C&%7Dkey=69AF3716A20387ED.F88D28E23EDFEDD1.4D4702B0C3E38B35.4D4702B0C3E38B35%7B%5C&%7D%7B%5C_%7D%7B%5C_%7Dacm%7B%5C_%7D%7B%5C_%7D=1543461791%7B%5C_%7D73e60222650cce67aedd1d1f613cbb6e.

[24] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation," p. 49, 2009. DOI: 10.1145/1655108.1655117.

[25] D. Dolev and A. C. Yao, "On the security of public key protocols," in *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*, IEEE, Oct. 1981, pp. 350–357. DOI: 10.1109/SFCS.1981.32. [Online]. Available: http://ieeexplore.ieee.org/document/4568353/.

[26] 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.

[27] 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.

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

[29] 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.

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

[31] 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.

[32] 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/.

[33] 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.

[34] 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.

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

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

[37] M. Yang, H. Wang, Y. Zheng, and Z. Jin, "Graph-tree-based software control flow checking for COTS processors on pico-satellites," *Chinese Journal of Aeronautics*, vol. 26, no. 2, pp. 413–422, 2013, ISSN: 10009361. DOI: 10.1016/j.cja.2013.02.019. [Online]. Available: http://dx.doi.org/10.1016/j.cja.2013.02.019.

[38] 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.

[39] 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.

[40] 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.

[41] T. H. Y. Dang and D. Wagner, "The Performance Cost of Shadow Stacks and Stack Canaries Time of Check to Time of Use," pp. 555–566,

[42] 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.

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

[44] 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..

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

[46] 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.

[47] 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.

[48] 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.

[49] 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/.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

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

[56] 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/.

[57] 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.

[58] 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/.

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

[60] 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.

[61] 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.

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

[63] 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.

[64] 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.

[65] 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/.

[66] 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.

[67] 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.

[68] R. For, R. Of, and N. Of, "SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust,"

[69] Y. Li, J. M. Mccune, and A. Perrig, "VIPER : Verifying the Integrity of PERipherals' Firmware," *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, vol. 22, no. October, pp. 3–16, 2011, ISSN: 09581669. DOI: 10.1145/2046707.2046711. [Online]. Available: http://dl.acm.org/citation.cfm?id=2046711.

[70] A. Seshadri, M. Luk, A. Perrig, L. van Doom, and P. Khosla, "Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems," pp. 253–289, 2007. DOI: 10.1007/978-0-387-44599-1_12.

[71] A. Seshadri, A. Perrig, L. Van Doom, and P. Khosla, "SWATT: SoftWare-based ATTestation for embedded devices," *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2004, pp. 272–282, 2004. DOI: 10.1109/SECPRI.2004.1301329.

[72] T. AbuHmed, N. Nyamaa, and D. H. Nyang, "Software-based remote code attestation in wireless sensor network," *GLOBECOM - IEEE Global Telecommunications Conference*, pp. 1–8, 2009. DOI: 10.1109/GLOCOM.2009.5425280.

[73] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence," *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 115–124, 2009. DOI: 10.1109/DSN.2009.5270348.

[74] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the Difficulty of Software-Based Attestation of Embedded Devices," 2009.

[75] 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.

[76] 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.

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

[78] 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.

[79] A. Berger, *Embedded systems design : an introduction to processes, tools, and techniques.* Lawrence, Kan. : Berkeley, CA : CMP Books ; 2002, ISBN: 1578200733.

[80] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "TrustZone explained: Architectural features and use cases," *Proceedings - 2016 IEEE 2nd International Conference on Collaboration and Internet Computing, IEEE CIC 2016*, vol. 7, pp. 445–451, 2017. DOI: 10.1109/CIC.2016.065.

[81] 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.

[82] 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.

[83] L. Davi, A. Dmitrienko, M. Egele, F. Thomas, T. Holz, R. Hund, S. Nurnberger, and A.-R. Sadeghi, "MoCFI: A framework to mitigate control-flow attacks on smartphones," *Symposium on Network and Distributed System Security*, 2012. [Online]. Available: http://www.researchgate.net/publication/228517736%7B%5C_%7DMoCFI%7B%5C_%7DA%7B%5C_%7DFramework%7B%5C_%7Dto%7B%5C_%7DMitigate%7B%5C_%7DControl-Flow%7B%5C_%7DAttacks%7B%5C_%7Don%7B%5C_%7DSmartphones/file/79e4150814635b1821.pdf.

[84] 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.

[85] 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.

[86] 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.

[87] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "LiteHAX," *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2018. DOI: 10.1145/3240765.3240821.

[88] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, May 2016, pp. 138–157, ISBN: 978-1-5090-0824-7. DOI: 10.1109/SP.2016.17. [Online]. Available: http://ieeexplore.ieee.org/document/7546500/.

[89] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," *7th Python in Science Conference (SciPy 2008)*, no. SciPy, pp. 11–15, 2008.

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

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

[92] 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.

[93] 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.