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

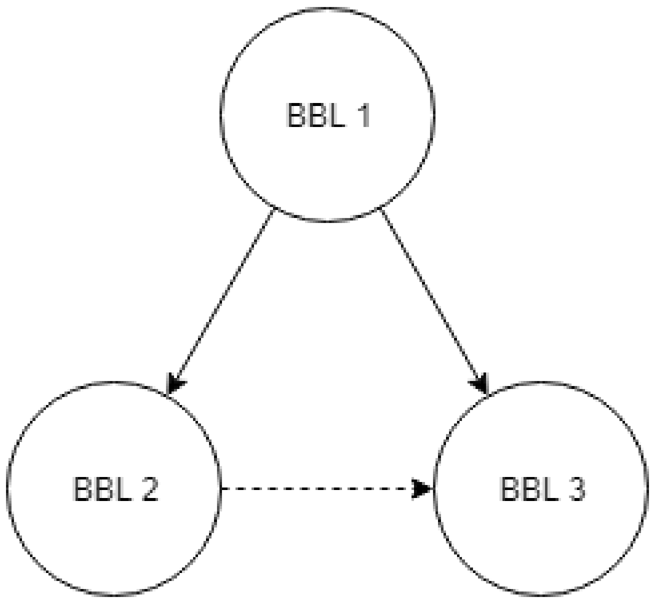

Figure 1.1: The control-flow graph for code above 1.1

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 1.1 for a CFG of 1.1. It is conceivable that a CFG can become enormous as the size of an application grows (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 for the code described above BBL 1 → BBL 2 and BBL 1 → BBL 3 are both valid flows, however BBL 1 → BBL 2 → BBL 3 is invalid. To create BBL 1 → BBL 2 → BBL 3 an attacker could change the value of `secondVum` 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 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.

In what common ways are these violated? Heap / buffer overflows in software?

Unintended

De Clerq [1] 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 [2]. For more information reference section 4.2 of Kayaalp's work [2] which describes unintended branches in detail. Does this affect embedded systems architectures?

Function Return

When a function returns to its calling section of code it references the stack for its return address. The interesting part here is that it is not hard coded and according to a CFG a function could legally return to any 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.

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 not feasible with CFGs. Various methods has been used to approach this, with some [20] 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 [3] track loop metadata such as iteration count.

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 a symmetric key is used for encrypting or applying a MAC to the attestation report.

1.2.2 Invasive Attacks

An invasive attack occurs when microprocessor removed and attacked directly through physical methods.In theory any microprocessor can be attacked in this way however 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 what does it stand for, reference discussion on PUFs 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, however this vulnerability has now been removed as test circuits are removed from the chip when it is cut from the die during the manufacturing process [4] [5].

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 [6], [7], injecting faults using laser [8] or white light [9]. Numerous more have been discussed in literature [10].

Fault Injection

Variations in supply voltage [4],[11] 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 [4],[12],[5] 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 [13],[14] can cause unpredictable effects in microprocessor. Two examples of effects obtained [8] are random modification of RAM cells due to overheating and read write temperature thresholds in most NVM what does this stand for? 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 such as?. Laser light [15] 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 [4] has been proposed as alternative to laser-based attacks [9], however as it is not directional it is a challenge to apply to particular portions of microprocessor and therefore provide a targeted attack pretty sure Mayes was successful with this. Electromagnetic flux [16] has been used to change values in RAM, where strong eddy currents can affect microprocessors although this has only been observed in insecure microprocessors.

The effects of fault injection include reset 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[4] which often has the same effect as previous examples but additionally allows for the removal of functions and breaking of loops.

Countermeasures given in [8]

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 given command, or attempting to inject faults in manners other than light. Examples include monitoring power consumption [17], [18] and injecting faults by glitching the power supply [4], [8].

1.3 Software attacks

Buffer overflow

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).

Injected code

When making use of injected code, control flow is deviated to existing injected code (usually as data). To mitigate against this NX bit can be implemented, this marks data memory as non-executable.

Code re-use attack

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. See also return oriented programming and return-to-libc attacks.

Return Oriented Programming (ROP)

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

return-to-libc

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

Pointer subterfuge

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 assigned to a pointer.

Non-control data attacks

These 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 [19].

1.4 Problem Description

1.4.1 Introduction

CFI has long been an issue, with examples of attacks including return oriented programming, stack overflows and hardware-based attacks link to attacks sections. Several means of defence have been designed to address some or all of the problematic results link to lit review and comparisons. 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.

- 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 [20] and has been subject to a large number of further research elsewhere.

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

1.4.2 Auditing control-flow

CFI prevention and detection has proven to lead to a high computational overhead, 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 keeping a record of the control-flow taken during a vitally important operation has been carried out.

Keeping audit records of control-flow could also be useful for spot-checking inaccessible devices which have no internet connection, such as 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 the requirements identified from the attack model.

1.5.2 Requirements

1. Works with compiled binaries: where the original source code does not need adjusting to be compatible with the solution.
2. Works with external libraries: similar to compatibility with compiled binaries but with the added constraint of no access to the external library source code.
3. Can be used to bind software to hardware: this would allow for the solution to that the software running on the devices is the software specially designed and deployed for said device.
4. Works with embedded systems: a crucial element as the solution is aimed at solving the problem for embedded devices.
5. Granularity: the solution should (at a minimum) be able to keep track of transition between basic blocks (including direct branches).

1.6 Project Structure

This project will be broken down into three sections. Section 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. Section 3 looks into the background existing literature surrounding the subject. Section 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 6 will describe the practical implementation. In section 7 we will perform a security evaluation on the proposed solution. Section 8 will provide the conclusion and a recommendation for directions for further work.

Chapter 2

Control Flow Integrity

2.1 Introduction

2.2 Control Flow Integrity

2.2.1 Introduction

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

2.2.2 Control Flow Graphs

Abadi et al. [20]introduced CFG as a method to protect code, using static analysis of a application binary to create Control Flow Graph (CFG). Only control flow transfers within the CFG are permitted.

Clerq and Verbauwhede [1] posed CFGs as a solution for instructions causing control flow transfers. Lee et al. [22] note that they do not focus on sequential transitions. They argue that the method used in [1] (where forward edges are described as control flow transfers caused by jumps and calls and backward edges are described as those caused by returns) is disadvantageous as by considering all jumps and calls equally there is a loss in distinction between jumps to register-determined and instruction-determined locations, and they state that when implementing a scheme instruction-dependant transitions are simpler to process than instruction-independent transitions. This is described in reference to [28]. What are sequential transitions vs control flow transfers?

2.2.3 Others?

[20] Describes jump labelling.

[1] Describes Shadow Call Stacks

[26] Describes SOFIA

2.2.4 Introduction to CFGs

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

CFGs have been used to provide protection against soft faults (single-event upsets) using software based methods [30] [31] [32]. [20] restricts control flow through in-lined labels and checks. The CFG is embedded through a set of static, intimidate 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 middle of functions (e.g. ones which bypass security checks such as access control)

The method described in [20] ensures that whenever a machine-code instruction instruction transfers control, it targets a valid destination as determined by the CFG (ahead of time). When the destination is determined at runtime this must go through a dynamic check.

Static checks in [20] are made by rewriting machine code using modern tools for binary instrumentation to get around the resulting new memory addresses [33] [34].

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

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

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

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

2.2.5 Hardware-based CFI

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

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

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

Seems similar: [43]

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

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

2.2.6 Software-based CFI

[40] Performs a survey

2.2.7 Building Control Flow Graphs

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

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

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

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

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

Chapter 3

Associated background

3.1 Static Attestation

3.1.1 Literature review - attestation of code integrity

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

SMART [55] 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 properties required by SMART are:

- Secure 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 key access - only SMART code on ROM can access the key. This is provided by registers only allowing access to the 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. E.g. if the 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 MCUs 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 states 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 [56] is a software only solution designed to provide attestation for peripherals’ firmware, where a the host CPU queries peripherals. It uses combinations of checksums, hashes and time-frame based checks to ensure peripheral firmware is correct. I have not found this solution to be particularly gripping however I may return to it if it suddenly makes sense to me.

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

Pioneer [57] is another software-only solution which works in a similar way to VIPER but is designed to attesting legacy systems. A checksum is through the verifying code, and is compared to an expected execution time, the checksum appears to also cover control flow of the verification code, which is interesting. 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 [58] is another 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, although I have to comment that time-based checking is questionable especially when attestation is taking place remotely ,e.g. over a network.

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 [59] (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. This is a general method used in a few solutions (such as. . .)

3.1.2 Literature review of dynamic, but not control-flow, attestation

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

Kil et al [60] 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 (which apparently is the equivalent to solving the halting problem, but what is the halting problem?). 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. They also mention Merkel hash tree for use in authenticating input files for the proc file.

This has been very different to other attestation methods previously examined so perhaps I'm not able to fully comprehend it as of yet, but it contains many tools and references which could yet prove fruitful.

3.1.3 Attacks on software-based static attestation

On the Difficulty of Software-Based Attestation of Embedded Devices

This paper [61] proves that my fears around software-based attestation were not unfounded. They attack SWATT [58] 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 note that the time overhead relied upon by SWATT [58] is imprecise and that their attacks fit within the tolerance. The paper also attacks solutions which fill empty memory with noise, they found they could compress legitimate applications to make room for malicious ones.

3.1.4 Conclusion

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 I have considered 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 might also provide protection against this but I have not fully decided about this yet.

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.

Apparently one must always state that attackers are assumed to be unable to break cryptographic primitives.

Perhaps the solution will need a certain amount of hardware assistance, could this be where ARM TrustZone comes in?

Merkel hash tree could be a good method of storing or tracking the history of control flow.

It is important to attest all parts of memory. This will be a focus for future work.

3.2 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.

3.3 Subject-matter Surveys

Many surveys on solutions which increase security for firmware/software in embedded systems have been completed. One such survey [62] focusses on existing mature solutions, while others [63], [64] and [65] 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 [66] which focuses on FPGA security.

These subject-matter surveys will be discussed in 3.3.1 and 3.3.2 and a deeper analysis of some of the solutions presented will be discussed in 3.5.1 and 3.5.2.

3.3.1 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 [62]. They compare various technologies, some of which are mature and some in their infancy. Studied solutions include: Trusted Platform Module (TPM), Secure Elements(SE), hypervisors and virtualisation (e.g. Java Card and Intel's Trusted eXecution technology), Trusted Execution Environments (TEEs), Host Card Emulation (HCE) and Encryption Execution Environments (E3 - which has also been directly discussed in [28]). The paper's authors set out a series of criteria which solutions are tested against, including such criteria as "Centralised Control', where the trust technology is under the control of the issuer or the maintainer, and "Remote Attestation" where the trust technology provides assurance to remote verifiers that the system is running as expected. The paper goes on to describe each technology in a small amount of detail and populates a matrix of technologies vs. criteria.

In a survey of anti-tamper technologies [63], 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 [67].

A similar survey [64] 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.

3.3.2 Defence against fault injection

A series of high-coverage tests for security protections against fault injection attacks were run and described in [65]. It describes 17 different countermeasures, including: countermeasures protecting the data layer, combinations of data protection methods, countermeasures protecting control flow layer, combinations of control flow protection methods and combinations of data and control flow protection. To test these methods the authors produced a high number of simulated fault injections on a simulator of an ARM-Cortex-M3 processor running a benchmark application representing a bank card.

The experiments found that a combination of redundant condition checks (such as data duplication) and source and destination IDs reached the best coverage with moderate performance overhead. They also found that simple ID-based inter-block control checking were able to outperform more sophisticated (and complex) methods such as Control-Flow Checking by Software Signatures (CFCSS) as seen in [51] and Assertions for Control-Flow Checking (ACFC) seen in [68].

3.3.3 FPGA Security

An excellent high-coverage survey on FPGA security is provided in [66], 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.

3.4 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 [69]. 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 [70], 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 [71].

3.5 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 3.5.1 ([28], [72], [73] and [74]) and software or secure software execution 3.5.2 ([51], [75], [21] and [76]). Hybrids of the two approaches are presented in [77] and [78].

3.5.1 Binding Hardware and Software

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

The first piece of work we consider is [28]. The problem the paper aims to address is device counterfeiting. An example of the requirement for binding of hardware and software is for Graphics Processing Units (GPUs), where GPUs are fabricated and then tested on their operating performance and subsequently graded. Once graded, the GPUs are loaded with firmware which controls their voltage and clockspeed. The paper states that firmware aimed towards the superior graded GPUs could be installed on lesser graded GPUs which would then be sold on as superior GPUs.

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

The method presented in [28] uses a function applied to either previous contents of memory or a randomly generated number to produce a mask which is applied to the program instructions residing in memory. The intention is that the CPU unmasks the contents as part of the execution process prior to actually carrying out the operation. The paper discusses the options for the mask-creating function, suggesting the use of hash-functions, block ciphers or PUFs before finally selecting PUFs due to their intrinsic nature. The act of masking the software has been undecided in this paper, which suggests that either the software is masked prior to loading or is masked during the loading process. The paper's author describes the provisioning process in a further paper [79].

The second piece of work we consider is [72] 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 [72] 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 [72] 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 [77] has not been published in a well-established journal however the authors have been invited to present their findings in [80]. 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 [77] 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 [78] 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 [73] 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 [74] 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.

3.5.2 Secure execution

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

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

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

Should I directly address GPSA and CSM?

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

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

The second paper [75] (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 [21] approaches secure execution from a slightly different direction: remote attestation. It aims to provide remote attestation of an application's control flow path during operation. The paper excludes physical attacks and instead focusses on execution path attacks, they assume that the subject device features data execution prevention (DEP) and a secure trust anchor that provides an isolated measurement engine and can generate the fresh authenticated attestation report.

Look up DEP

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

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

The fourth paper [76] 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.

3.6 Primitives

3.6.1 Control-flow Graphs

The use of control flow checking for accidental program flow changes is considered in [68], but still presents the basic principle. It describes basic blocks, control-flow between blocks, how these make up program graphs and finally how these graphs can be used to indicate control-flow error. The paper presents two existing error detection methods but finds faults in both so presents a novel solution addressing the previous solutions' shortcomings.

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

3.6.2 PUFs

what does PUF stand for?

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 [81], which provides a thorough examination into most aspects of PUFs including the types, analysis of each type in terms of uniqueness and reproducibility and uses including entity-authentication and key generation.

Much of the literature ([28], [72], [77], [78], [73] and [74]) already described explain the use of PUFs and their reasoning behind their choice in PUFs.

3.7 Conclusion

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

3.8 Important Principals

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

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

3.9 Introduction to Embedded Systems

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

3.9.1 A Definition of Embedded Systems

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

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

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

3.9.2 Uses of Embedded Systems

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

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

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

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

3.9.3 Key Properties and Criteria of an Embedded System

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

3.9.4 Common Implementations of Embedded Systems

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

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

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

3.10 Problem Description, Assumptions and Requirements

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

Skipping instructions can be used to bypass checks.

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

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

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

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

Requirements:

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

3.11 Common Attacks

Physical attacks - power ...

Return to lib-c

Return oriented programming

3.12 Control Flow Integrity

Fine grained, course grained.

Flow down to low level - sequential Basic blocks

Control flow graphs

Shadow return address tables

Program counter

3.13 Data Flow Integrity

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

3.14 Existing Solutions

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

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

Additional notable - NX bit, canaries

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

Problems: PUFs CFG generation Checking code being altered

Compare to objectives

3.15 Conclusion

3.16 ARM TrustZone

3.16.1 An introduction to ARM TrustZone

ARM TrustZone is a method used to separate "secure" applications from non-secure ones in two separate worlds, memory can be designated as secure or not secure. Execution is transferred from each world either through interrupts or direct function calls. This page describes the memory aspect quite well. While this stack overflow q and a explores it in a more real life example. This is very well described by Ngabonziza et al [82].

I believe secure world TrustZone applications should have access to normal world memory, this will enable static attestation.

Provisioning is an interesting aspect which I haven't quite worked out out.

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 4

Comparison of Solutions

4.1 Comparison of solutions

C-FLAT

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.

LO-FAT

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. 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 (changing the contents of software to meet the requirements of the solution).

Secure-Execution Processor (SEP)

Uses encryption of each instruction which uses the previously executed instruction as an input to the decryption algorithm. Decryption is performed in the processor pipeline which therefore prevents subverting control-flow, if an attempt is made to subvert 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. Requires code instrumentation.

CCFI-Cache

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 has been modified in a manner which affects the length. The checker checks 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 in interrupt on the CPU will be triggered. In this author's opinion this solution provides the best coverage for protecting control-flow integrity. Requires code instrumentation.

HCFI

Implements a shadow-stack on additional hardware. It does not describe how it ensures integrity of forward edges but is a thorough analysis of backward-edge protection through the use of a shadow stack. I'm not sure if this requires code instrumentation or not.

HAFIX

HAFIX (like HCFI) 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. Requires code instrumentation.

SOFIA

Works in a similar manner to SEP, 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 building up multiplexor blocks. The fast expansion of multiplexor blocks is a concern for size and operating time of applications. I wonder how SEP (Lee) does it. Requires code instrumentation (multiplexor blocks and encryption).

Strategy without tactics (Sullivan)

Uses trampolines to deal with multiple callers, where a single destination is replaced with a trampoline, this addition transforms a call source/destination pair into a unique call source/trampoline pair. From the trampoline, a direct jump is issued to the original destination (REFACTOR THE LAST TWO SENTENCES AS THEY WERE COPIED). A label state stack records backward-edges. Label state register records forward-edges. I struggle to fully comprehend this solution so could be worth re-reading.

Learnings

Backward edges cannot be protected against without the use of shadow stacks - as returns may be legal for many functions (e.g. printf). Is this where CF auditing can be useful? By keeping track of the flow taken.

4.2 Introduction

4.3 Requirements

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

4.4 Properties

1. Hardware based
2. Software based
3. Hardware modification required
4. Source code modification required
5. Granularity
6. Provisioning method
7. How it is secured?
8. CPU Overhead
9. Storage overhead
10. Memory overhead
11. Execution time overhead
12. Compatible architectures
13. Action after identification of insecure event
14. Prevention / Detection / Attestation
15. CFG or other

4.5 Solutions

1. C-FLAT [21]
2. LO-FAT [3]
3. CEP-LEE [22]
4. CCFI-Cache [23]
5. HCFI [24]
6. Hafix [25]
7. SOFIA [26]
8. Sullivan [27]

4.6 Conclusion

4.7 Comparison of solutions

See 4.1.

Table 4.1: Table caption here

Scheme	Requirements									Properties										
	Works with compiled binaries	Works with external libraries	Can be used for binding	Works with embedded systems	Immediate identification	Not reliant on TPM	Hardware-based	Software-based	Hardware modification required	Source-code modification required	Granularity	Security provided by	CPU overhead	FPGA area overhead	Memory overhead	Time overhead	Compatible architectures	Action on identification	Prevention or detection or attestation	Uses CFG or another method	Success rate
C-FLAT$^{\dagger *}$	-	-	-	-	-	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
LO-FAT†	◐	◐	◐	◐	-	●	●	●	●	-	●	●	-	●	●	-	●	●	●	●	●
CEP-LEE$^{\dagger *}$	◐	◐	◐	◐	-	●	●	●	●	●	●	●	-	●	-	-	●	●	●	●	●
CCFI-Cache$^{\dagger *}$	●	●	●	●	◐	●	-	-	-	-	-	●	-	-	-	-	●	●	●	●	●
HCFI$^{\dagger *}$	●	●	●	●	◐	●	-	-	-	-	-	●	-	-	-	-	-	●	●	-	●
Hafix$^{\dagger *}$	●	●	●	●	◐	●	-	-	-	-	◐	●	-	-	-	-	●	●	●	-	●
SAFIA$^{\dagger *}$	●	●	●	●	◐	●	-	-	-	-	●	-	-	-	-	-	●	●	●	-	●
Sullivan$^{\dagger *}$	●	●	●	●	◐	●	-	-	-	-	●	-	-	●	●	-	●	●	●	-	●

● = provides property; ◐ = partially provides property; - = does not provide property; †has academic publication; *end-user tool available

Scheme	Requirements									Properties										
	Works with compiled binaries	Works with external libraries	Can be used for binding	Works with embedded systems	Immediate identification	Not reliant on TPM	Hardware-based	Software-based	Hardware modification required	Source-code modification required	Granularity	Security provided by	CPU overhead	FPGA area overhead	Memory overhead	Time overhead	Compatible architectures	Action on identification	Prevention or detection or attestation	Uses CFG or another method	Success rate
C-FLAT [21]	●	●	○	● ARM CORTEX A	○	○	●	●	○	Control flow instructions re-written to go via trampolines	Basic Block	TPM and Digital Signatures (for report)	-	-	-	0.03 - 0.13%	32-bit RISC	-	Attestation	CFG	-
LO-FAT [3]	●	●	○	●	○	●	●	○	●	○	Basic Blocks	LO-FAT Memory is protected	0	20%	-	0	PULPINO RISC-V	-	Attestation	CFG	-
SEP-LEE [22]	○	○	●	●	●On jump or return	●	●	○Reassembly	●	●	Sequential or BBLs	PRINCE	-	-	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 [23]	○	○	○	●	○End of BBL or number of executions reached	●	●	◐	●	●	Sequential (Only number or hash of instructions)	Meta-data in ROM, Hardware Security for controller	-	10%	x2 + 9-30% for nops	2-63%	PicoRV (Implementation on RISC-V ISA)	Interrupt	Detection	CFG + No of instructions + hash of instructions	Small # of ROP & JOP attacks will still work
HCFI [24]	○	○	○	●	●	●	●	○	●	●	BBLs	-	-	2.5%	-	1-6%	Leon3 / SPARC V8	Interrupt	Detection	CFG and Shadow Stack	-
HAFIX [25]	○	○	○	●	●	●	●	○	●	●	Backwards edge	-	-	2-8%	-	2%	Intel Siskiyou Peak, SPARC	-	Detection	CFG	80%
SOFIA [26]	○	○	●	●	●	●	●	○	●	●	Sequential instructions or flows	Encrypt+MAC (RECTANGLE-80)	13.7% Cycle overhead, 84.6% Clock speed reduction	28.2%	-	110%	Leon3 / SPARC V8	Processor reset	Prevention and detection	Sequential and BBLs	-
Sullivan [27]	○	●	○	●	●	●	●	○	●	●	BBLs	Attacker cannot touch code	-	1.78%	-	1.75%	Leon3 / SPARC V8	Termination of process	Detection	CFG	Dependant on CFG policy (assumed 100%)

4.8 Thoughts after comparison

4.8.1 Signature Modelling

When I first learnt of CFI, it was described in relation to control-flow graphs (CFGs). Here if a series of executed instructions do not follow the trace of the program's CFG the control-flow integrity has been lost. The paper [83] did a very good job at describing this method as implemented in software. They also performed a thorough analysis of various existing solutions (page 2). I first saw this methodology in use in hardware in the paper written by Werner et al [51] although I have to admit I got a bit scared off after seeing all of the algebra! I think I'll have to return to this paper at a later date.

A problem with signature modelling is that sequences of instructions (and their accompanying signatures) accumulate very quickly, requiring more storage. Actually now I think about that is that really correct? A cryptographic signature (hash) is the same length for all inputs. Another problem is the necessity to constantly compute signatures, this would add a fair amount of processing overhead. Although, as we will see in future methods, encryption is a common method of ensuring CFI so I suppose perhaps it is not too much to ask for.

4.8.2 Basic Blocks

As a result of investigating CFGs I learnt about basic blocks (BBs). These are sequences of instructions which will run all the way from the start to the end, i.e. there will be no branches within (for example if/else). Various academics have leveraged the principle of BBs to simplify CFI, where source code / assembly is analysed to build basic blocks and determine how they link up. This leads on to the types of branches/jumps (e.g. direct or indirect) which I will focus on in a later blog post. Many authors of papers leverage some sort of method to label which blocks can lead to others (and back again). The issue with this is that it can, again, accumulate for larger programs. Also keeping track of commonly called functions, and where they return to is a tricky problem (as discussed/attempted to address in various papers).

4.8.3 Shadow Stacks

Some [24] have argued that ensuring CFI without the use of shadow stacks (SSs) is an impossible task and that "the use of a a shadow stack is mandatory for any practical CFI deployment". A shadow stack is basically what is says on the tin. A separate stack is stored which contains just the return addresses. This solves the problem of ROP, especially when working with commonly called or system functions as the implementation of the shadow stack ensures the calling function has to be the return function.

Of course, as there always seems to be in academia, there are some flaws with shadow stacks. These 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 it flag (or counter) by some solutions. This can raise the problem of manipulating the number of recursions by an attacker, however it is argued by some reference pending that this is a very limited attack.
- Instructions such as longjmp (not sure about the spelling) will skip back through multiple return addresses, so this needs to be taken into account - ever 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). I cannot work out if the second method is a good one or not, but it makes a lot of sense to me.

4.8.4 Conclusion

I think shadow stacks should be implemented in any solution I adopt - perhaps R. Lee's CEP from 2019. My idea would be to maintain a shadow stack in protected memory and page it out to main memory using encryption to secure the paged version. This shadow stack would make use of the recursive single-bit flag, a process ID is threading is used (although I have to admit that I haven't even considered threading yet - project idea?), and would simply pop entries off if a longjmp-like instruction is called (in the hope that this does not provide an attacker with an advantage, but I bet it would because they always find a workaround.).

I have also rediscovered my love for signature based CFI, which is what I planned on using for an attestation based CFI mechanism (much similar to C-FLAT [21]).

Chapter 5

Theoretical Solution

5.1 Building CFGs

5.1.1 The problem behind building CFGs

C-FLAT [21] 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 therefore possible to reasonably compute a CFG for.

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-c`

I understand that this may not be a simple procedure and will need further investigation.

The dot format is highly used in CFG but I'm not sure if it will map well with what we receive while monitoring it.

Vulcan

Vulcan [33] is used by [84] and the seminal work [20]. Though it has since been taken out of production by Microsoft.

Jakstab

Jakstab [53] can also be used to create a dot file containing the CFG of x86 executables, so this may need features adding in order to support embedded systems. The last development activity on this was 31st March 2017. I tested this out but was unable to produce any CFGs from my test programs (written in C Sharp. My lack of understanding of windows executables.

Others

Kinder [42] has been trying to improve on this where the CFG is under approximated.

In C-FLAT [21] the authors create their own analysis tool to achieve their branch-based requirements rather than create CFGs. They also use Capstone disassembly engine [1] however this is more for use as dissasembling code rather than CFG building.

[1] `http://www.capstone-engine.org/`

LiteHAX [85] uses the 'angr' [2] [86] framework to generate the CFG outputting a networkx DiGraph. This could be good one as it 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). LiteHax is worth another read as it seems to be quite thorough.

IDA Pro.

5.2 Tracing Control-Flow

5.2.1 Using existing hardware

CFGs

Recording of every executed instruction is infeasible as the overhead added would be enormous, computationally if each is hashed as we go or memory/storage if each instruction is recorded prior to processing.

CFG granularity can be broken down into three levels [21]:

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

We will most likely choose option 3.

We can build up a hash of the flow by taking the ID of the source node and hashing that with the previous hash. If at the start the previous hash would be replaced with zero, or a nonce? so H1 = (ID1,0/nonce), H2 = (ID2,H1) etc.

We will implement a special 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 `https://blake2.net` 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 [21] 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 perhaps). 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 to do clever stuff. Read more about it please! 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.

For all of this see section 4.2 of C-FLAT.

[2] `http://angr.io/`

Instrumentation

Branches are identified, with direct jump addresses stored in a branch table. They are then directed to the trampoline assembly code which sends it off to the measurement engine to be measured.

Trampolines

A trampoline is an easy method of handing off from the main.

In C-FLAT [21] the trampoline manages the return address register as well as the register holding the destination address in indirect branches. Reference C-FLAT for a detailed explanation.

5.3 What is stored

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 done in regards to CFG, we will need to investigate this further.

Variables

We will implement a function where the software can assert (send information to secure world on) variables in use either when saving or as an additional instruction through the solution's API.

Again this will be implemented as part of this project.

External data or internal data: I'm not completely sure what this means, but Raja mentioned it.

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.

5.4 Provisioning

5.4.1 Instrumentation

The existing binary will be instrumented either prior to loading or during the loading (bootloading process). Instrumentation means replacing instructions from the compiled code with ones which are used for attestation.

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 (reference some sort of ARM document).

Chapter 6

Practical Implementation

Chapter 7

Security Analysis

Chapter 8

Conclusion and Further Work