Winter 12-22-2021

# Control Flow Integrity for Real-time Embedded Systems

Yuqian Huo

Washington University in St. Louis

McKelvey School of Engineering

Department of Computer Science and Engineering

Thesis Examination Committee:
Ning Zhang, Chair
Christopher Gill
Sanjoy Baruah

# Control Flow Integrity for Real-time Embedded Systems

by

Yuqian Huo

A thesis presented to the Graduate School of Arts and Sciences
of Washington University in partial fulfillment of the
requirements for the degree of

Master of Science

December 2021
Saint Louis, Missouri

# Contents

# List of Tables

# List of Figures

# Acknowledgments

I cannot believe that two years went away this fast and it feels like it was yesterday that my parents were still helping me check I brought everything in my luggage to study abroad. Recalling two years of study at WashU, there are three people besides my family I must give special thanks to.

The first person is Guangyu, we met at communication class in the first semester. He is five years older than me and came to WashU as a master's student with his family to pursue his dream. Guangyu's story always inspires me it is never too late to do anything. I used to have hesitation whether to work or get a Ph.D. after graduation since I transferred my major from Electrical and Information Engineering to Computer Science. He made me brave when I felt upset because I have to make up many things that others might have already learned in their undergraduate study. The most important thing I felt fortunate for is that he recommended professor Zhang to me after knowing I was interested in computer systems. The second person is Jinwen. He is easy-going and always willing to answer any questions and gave me many ideas in system security. He is also very diligent. I can tell he has enthusiasm for system security and he always shows up in the office working. I regard him as a model and idol in studying.

I had been dealing with MSP430 microchip systems throughout almost four years of my undergraduate study and had no idea of what exactly computer security is. Professor Zhang did an excellent job in bringing me to this field. I started working with professor Zhang in early 2021 and took two of his classes both are very helpful to me. CSE 433S Introduction to Computer Security gave me a broad view of what computer security is, and enable me to do some interesting buffer overflow attacks. I gained both interest and knowledge from this course. Compared to introductory level classes, CSE 569S Recent Advances in Computer Security took a more detailed view of what current researchers have done in computer security in various directions. This course is very hard-core that we read two papers per week and I knew that what computer security is like from a researcher's view. Fortunately, I gained more interest in computer security instead of being defeated by those scary terminologies that I have ever heard of. Professor Zhang is an excellent instructor. He gave me 100 percent trust and freedom in choosing my research topic and helped me do this. He always asked me

what I would like to do and gave me lots of advice. He never showed any angry or upset with me when I did not make progress in one project. Nevertheless, he encouraged me countless times saying no matter what happened he will always support me. These words always assist me to continue when I encountered challenges during my research. I also appreciated that he is willing to talk about many interesting things with me and lab members when we have dinner or lunch together. It makes me feel at home and not hesitate to ask him for help.

Besides, I would like to thank my parents and my grandparents who care about how I lived, what I ate, and how I feel when studying abroad. And Yizhe who accompanied me during two years of study. I would not accomplish this work without any of you.

Yuqian Huo

*Washington University in Saint Louis*
*December 2021*

Dedicated to my family.

ABSTRACT OF THE THESIS

Control Flow Integrity for Real-time Embedded Systems

by

Yuqian Huo

Master of Science in Computer Science

Washington University in St. Louis, December 2021

Research Advisor: Professor Ning Zhang

Devices built on embedded systems are widely used in our daily lives. Nowadays, firmware typically uses C and C++ for efficiency and durability. However, those languages are unsafe which can lead to many software and system security issues. Attackers can easily corrupt a system by issuing various memory corruption attacks on a vulnerable program. Control-flow integrity is one of the most prevalent mechanisms used to protect against memory corruption. Most research papers and prototypes focus on using CFI on high-performance chips such as Intel and ARM Cortex-A. However, many embedded systems targeting time critical services are built on resource constrained devices. Many mechanisms cannot work, or have large runtime overhead, when been applied to those embedded systems. This paper presents work applying a CFI policy on resource constrained systems while sustaining security guarantees. We propose a mechanism for applying control-flow integrity in real-time embedded systems to mitigate memory corruption attacks.

# Chapter 1

# Introduction

Embedded systems can be widely seen in various fields nowadays. Those micro-computer controlled applications are used to help a human do a series of tasks by just letting us push one button. The prevalence of Internet-of-Things gives more importance to embedded systems. Besides the devices seen in our day-to-day lives, embedded systems are broadly used in scenarios such as health care, industrial control systems, and avionics. *Paul, et al.* [24] firstly brought the significance of aspect in security in embedded systems in 2004. Two types of security can be applied to embedded systems: software-based embedded security and physical-based embedded security. Physical-based embedded security focused on how to protect the embedded system from being physically attacked by the outside world such as using hammers to hit the smart lock. Software-based embedded security focused on the systems such as communications through the network and the security guarantees of the software installed on the system. In this article, we focused on software-based embedded security.

Real-time embedded systems are a subset of embedded systems. Embedded systems are designed to have few functions on the chip. Combining with some mechanical devices, they are meant to handle one or two services. Real-time embedded systems are designed to handle time-critical services that need to be done in a certain amount of time. Compared with regular embedded systems, real-time embedded systems are used to handle various tasks.

Software on the embedded system, usually we call it firmware, has the same vulnerabilities as the normal software running in our computers. Since the goal of embedded systems is designed to be low power-driven, the firmware on the chip is used to handle various small

tasks and does not need a high-performance CPU to run the program. As a consequence, the efficiency of the program is crucial for the design. C and C++ are the most popular programming languages used on embedded systems. They have a good reputation for their speed and stability. However, they are also known for their unsafe feature [4]. Control-flow Integrity [1] is a popular and effective way that can mitigate attacks on such C and C++ vulnerabilities. Many CFI mechanisms have been published at top-tier conferences, but most of them focused on normal computing systems based on Intel or ARM processors that have great computing performances. This article introduced a mechanism using CFI to ensure the security of the embedded systems which have limited resources on-chip using binary instrumentation.

## 1.1 Motivating Example

To get control of the system, the ultimate goal of attackers is to execute arbitrary code in the system. They can execute their shellcode or execute their wished code snippet by redirecting a pointer to a specific address. If the attackers can redirect the address of the pointer points to, we see this attack as a successful control-flow hijack attack. Figure 1.1 is a very intuitive example to show how the attackers can bend the control flow. From the code we can see that `fp_a` can point to `fooA`, `fp_b` can point to `fooB`. The attackers can issue memory corruption attacks to make `fp_a` points to `fooB` which is not legal in the code. CFI can be used to check this kind of violations and protect the system.

## 1.2 Thesis Structure and Contributions

The rest of this paper is organized as follows. In Chapter 2, we introduce the background of memory corruption, attacks to hijack control flow, CFI, and ARMv8-M. Chapter 3 contains the threat model for our system. We will describe our system implementation in chapter 4 and evaluate our system from performance and security guarantees in chapter 5. Finally, we give the conclusion of our work in chapter 6.

We highlight our contributions below:

```
void fooA(){
}
void fooB(){
}

int main(){
   void (*fp_a) = &fooA;
   void (*fp_b) = &fooB;
   fp_a();
   fp_b();
}
```

Figure 1.1: Control-Flow Hijacking

- **Applied a CFI policy on a real-time embedded system.** We applied CFI policy on an ARMv8-M development board using binary instrumentation.
- **Binary instrumentation for Thumb-2 ARM ISA.** We developed a novel way to do binary instrumentation on Thumb-2 ISA without changing the whole memory layout of the program.
- **Control-flow graph construction from binary.** We retrofitted an open-source reverse engineering tool Ghidra to construct a control flow graph and save it to a secure place using sandboxing.
- **Mechanism has low runtime overhead.** Our implementation enforced forward-edge control-flow integrity and have an optimal runtime overhead.

# Chapter 2

# Background and Related Work

Before diving right into the implementation of our system design, we will introduce our work background and related work in detail in this chapter. This chapter first introduces an essential scientific research direction in software security and some typical attacks in this aspect. Then it presents more details about how control-flow hijack attacks work and how we can protect against this kind of attack. This chapter also introduces the background of the system that we applied our CFI policy on. Finally, it presents the related work of this paper.

## 2.1 Memory Safety

Memory is a very important aspect that we need to pay attention to in computer security. Since C and C++ do no bound checking and the stack is writable and readable, it is easy for the attackers to make their way to get sensitive information on our systems or make a change to our system memory. After changing the contents of the memory, attackers can do lots of things such as getting information and executing arbitrary code.

*Laszlo, et al.* [38] gave an excellent analysis and classification on memory safety. Memory safety can be triggered in two ways: spatial error and temporal error. Spacial errors are caused by indexing out of bounds or using string printing to print out the contents on the stack. Figure 2.1 gives an example of a spacial error in which `scanf` and `strcmp` can let the attacker be authorized to the system even putting in the wrong password. Temporal errors are caused by dereferencing a dangling pointer. Common attacks using temporal errors are

double free attacks and use-after-free attacks. Figure 2.2 shows a double free attack in which the attackers can use the dangling pointer to jump to shellcode and get root privilege of the system.

```
char pass_wd[7] = "hero123";
char input[7];

scanf("%s", input);
if(strcmp(input, pass_wd) == 0){
    authorized();
}else{
   denied();
}
```

Figure 2.1: Spacial Error

```
char *a = malloc(10);
char *b = malloc(10);

free(a);
free(b);
free(a);
```

Figure 2.2: Temporal Error

Here list several ways to protect memory safety and their pros and cons:

- **Modify the code by removing unsafe functions and doing the bound checking.** This method can to some extent make the attacks hard, but it cannot guarantee the security of the system. It also needs a lot of labor and time to modify the code. Code reuse attacks and code injection attacks are still feasible from this protection.
- **Randomization.** Randomization techniques such as Address Space Layout Randomization(ASLR) [20] is one of the most effective and most adopted mechanisms nowadays. It can make it hard for the attackers to execute arbitrary code by randomizing the memory address space. But it has limitations since the attackers still have the chance to use brute force to issue the attack. Some mechanism proposes re-randomize the memory layout after a certain time which theoretically has a stronger security guarantee. But it may face a large runtime overhead.

- **Data Integrity and Data-flow Integrity.** As the name says, these mechanisms [7][26] track the data and the data flow in the program. These mechanisms can achieve better security guarantees. Since data `load` and `store` instructions occupy a huge proportion of a program, they have an unacceptable runtime overhead for modern systems for about 200%.

- **Code Pointer Integrity.** *Volodymyr, et al.* [25] introduced code pointer integrity(CPI) in 2014. CPI separates code pointers to be in a safe region and unsafe region to ensure the integrity of the code pointer. Control-flow integrity can be ensured if code pointer integrity is secured. However, it is hard to guarantee the integrity of the code pointer perfectly and it also poses greater runtime overhead than control-flow integrity.

- **Control-flow Integrity.** Control-flow integrity was first introduced by *Martin, et al.* [1], it aims to protect the targets that pointer points to are legal or not. We will discuss CFI in detail in Chapter 2.3.

- **Data Execution Prevention(DEP) [14].** DEP is used to protect code-injection attacks. It is an effective way to prevent attackers redirect the control flow to the code injected in the program. This plays an important role in nowadays computing systems with ASLR to protect memory safety. However, DEP cannot protect against code reuse attacks as attackers can chain their code gadgets from the existed code in libraries such as libc.

In this article, we focus on using control-flow integrity to protect memory safety.

## 2.2 Control-flow Hijacking Attacks

Control data refers to return instruction or indirect jump or call instructions. Non-control data refers to data other than control data such as user identity, password, or other critical data that may influence the execution of the program. Attackers may change both control data and non-control data to bend the control flow of the program. Figure 2.3 is an overview of control-flow hijacking in which the attackers can bend the control flow from normal execution to attacker aimed malicious code. A basic control-flow integrity policy can only protect

the system from control data attacks. Some CFI mechanisms such as $\mu$CFI[19] add other instrumentations to record the context of the critical data that will influence the program. This to some extent can partially protect non-control data attacks.

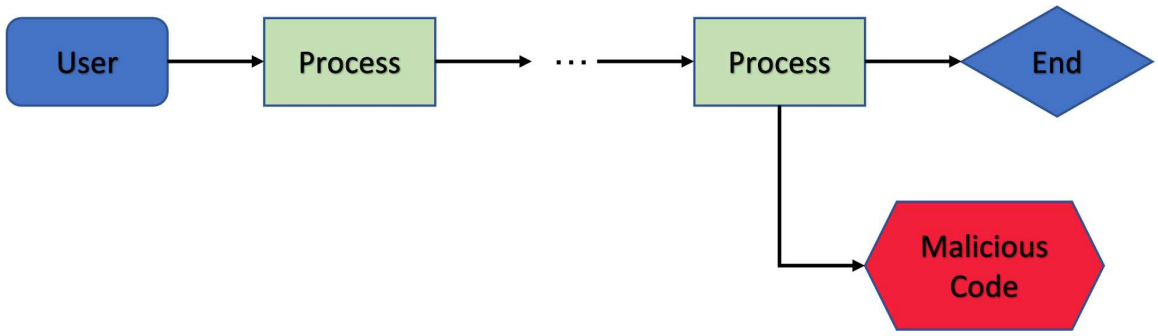


Figure 2.3: Hijack The Control Flow

Here are some examples of control-flow hijacking attacks:

1. Return to LibC attack [37]: redirect control to existing vulnerable code in C library.
2. Return oriented programming [34]: chain the gadgets using return instructions.
3. Jump oriented programming [5]: chain the gadgets by using dispatch gadget.
4. Counterfeit object-oriented programming [36]: chain objects' virtual functions together.

## 2.3 Control-flow Integrity

Control-flow integrity is a security policy that guarantees the control flow at runtime is legal. Typically it will have a control-flow graph(CFG) to refer to at program runtime and check whether the executed control flow matches the CFG. There are three critical elements for control-flow integrity: control-flow graph construction, program runtime enforcement, and runtime overhead.

### 2.3.1 Control-flow Graph

A control-flow graph can be constructed in three ways. If given the firmware's source code, we can use static analysis to construct the CFG called static CFG. Static CFG is an over-approximation of the firmware. Thus, it allows false negative(regard illegal control flow as legal transfer) in CFI. Figure 2.4 gives a scenario of a false negative in CFI. `f` has two call sites: `call_site_a` and `call_site_b` and has two targets(where the pointer can point to) `target_a` and `target_b`. The static CFG constructed by static analysis are pointes-to set and allow `f` point to either `target_a` and `target_b`, which is not always the case. For instance, at function `f`'s call site `a`, it points to `target_b` and it points to `target_a` at function `f`'s call site `b`. If the attacker bends the control flow of `b` at call site a to point to `target_a`, CFI will not raise an alarm since it does not violate the CFG. `target_a` and `target_b` are called Equivalent Class(EC). The ultimate goal of CFI is to let each pointer point to a unique target so that reducing EC can help improve the security guarantee of CFI.

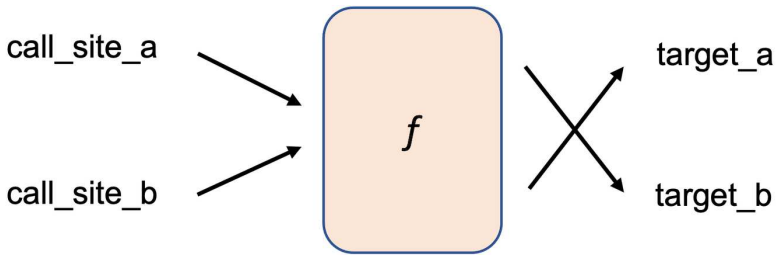


Figure 2.4: Equivalent Class

To solve this issue, many mechanisms propose using context-sensitive or path-sensitive CFI to get a more precise CFG. Some mechanisms propose using hardware [8][10][11][12][19] to track the executed path at program runtime. This is called dynamically constructed CFG. Others [21][22][30] focusing on adding more context such as call site, function and object origin during instrumentation.

If the source code of the firmware is not available, which is usually the case, we need to construct the CFG from the binary. CFG from the source code is more accurate than that from the binary since it is hard to recover the semantics from binary to source code. Recently, getting an accurate CFG from binary is still a hard problem to be solved.

### 2.3.2 Program Runtime

During program runtime, purely software-based mechanisms can enforce CFI policy without hardware support such as $\pi$CFI [30]. Many other mechanisms [8][10][19][22] used hardware to either increase context of the program to get a more precise CFG or use hardware to protect the metadata protection more efficiently. CCFI [29] applied cryptography on the system to enforce CFI. After instrumenting the program, reference monitors check CFI policy according to the CFG at runtime. To ensure the program runtime is secure, many mechanisms use hardware or software isolation to isolate metadata and reference monitor outside the sandboxing.

### 2.3.3 CFI Policy

There are various control-flow integrity policies have been introduced. Although their main topics are all control-flow integrity, they focused on different aspects.

- Forward-edge CFI: Some mechanisms focused on forward-edge control-flow transfer checking which means they only consider indirect jump and call. Those CFI policies without backward edge checking are considered to be context-insensitive CFI.
- Backward-edge CFI: Backward-edge CFI policies are usually focused on optimizing the system to handle return instructions more efficiently. $\mu$RAI [2] replaces all return instructions with a direct jump to avoid using shadow stack. PACStack [27] used ARM Pointer Authentication to check the return policy more accurately and efficiently.
- Coarse-grained CFI: If the policy only allows a few ECs or just one Equivalent Class, it is considered to be coarse-grained. An extreme example of coarse-grained CFI is the policy that allows all return instructions to the set of all return addresses which does not care if they are legal or not. binCFI [42], CCFIR [41] are examples of coarse-grained CFI.
- Fine-grained CFI: Be opposite to coarse-grained CFI, fine-grained CFI [31][32] aims to ensure each indirect control transfer can only transfer to its legal target.

- Context-sensitive CFI: To construct a more precise CFG, the idea of context-sensitivity comes to people's minds. They either use dynamic and concolic execution to construct the CFG or add more instrumentation to the program to get more a more precise relationship of each indirect control transfer to reduce ECs. OS-CFI [22] track the definition and call-site address to mark the origin of the indirect control transfer which reduced the capable number of indirect control transfer targets.
- Path-sensitive CFI: PITTYPAT [12], $\mu$CFI [19] and GRIFFIN [16] used Intel PT [23] to track executed path dynamically to record a dynamic CFG. Those mechanisms improve the context dynamically, however, they are not as precise as OS-CFI and Intel PT may lose some packets if the tracking tasks are too overwhelming.

## 2.4 ARM Cortex-M

Nowadays ARM processors are gaining more attention in our life. Intel processors have a large portion in large technical applications such as the desktop. While ARM is often found in embedded systems and mobile devices. It is noticeable that ARM can also be used in large technical applications since Apple now using ARM for designing their system-on-chip(SOC). ARM and Intel have different Instruction Set Architecture(ISA). ARM uses Reduced Instruction Set Computing(RISC) while Intel uses Complex Instruction Set Computing(CISC).

In this work, we use ARM microprocessors designed for embedded systems. For different purposes, ARM microprocessors have different features. ARM microprocessor family has six microprocessors: Cortex-A, cortex-M, cortex-R, Ethos, Neoverse, and SecurCore. Ethos is a high-performance chip for machine learning, so we will not discuss it. We gave a detailed comparison of ARM Cortex A15, M33, R5, Neoverse V1, and SecurCore SC300 in Table 2.1. Among these processors, we use cortex-m33 for implementation.

### 2.4.1 Thumb-2 Instruction Mode

ARM regular ISA is ARM mode that executes in a 32-bit execution environment. Each instruction takes four bytes of space to finish one instruction. There is another ISA called

Table 2.1: ARM Family Comparison

| **Processor** | **Cortex-A15** | **Cortex-M33** | **Cortex-R5** | **Neoverse V1** | **SC300** |
|---|---|---|---|---|---|
| ARM ISA | • | | • | • | • |
| Thumb ISA | • | | • | | • |
| Thumb2 ISA | • | • | • | | • |
| Multi-core | • | • | • | • | • |
| Single-core | | • | • | • | • |
| Performance | • | | | • | |
| Real-time | | | • | | |
| Microcontroller | | • | | | |
| Low Power | | • | | | |
| MPU | • | • | • | | |
| MMU | • | | | • | • |
| TrustZone | • | • | • | | • |

Thumb [33] which takes 16 bits to complete an instruction. ARM also designs a Thumb-2 mode, which can operate in both 16-bit and 32-bit instruction. Most of the instructions are 16-bit wide and it will change the mode to 32-bit instruction if the 16-bit size instruction cannot hold up the address. Figure 2.5 is a code snippet for our application. Most instructions are two bytes long and the mode change to 32-bit in line 5.

### 2.4.2 ARM TrustZone

Trust Execution Environment(TEE) [35] is a hardware isolation mechanism. Intel-SGX [9] and ARM TrustZone [39] are popular TEE used nowadays for sandboxing. ARM TrustZone is a hardware-based sandboxing method provided with the cortex-m33 development board we use. By using ARM TrustZone, we can divide the whole system into a secure world and normal world. To isolate the critical data, the trusted runtime and all the metadata are stored in the secure world and the running firmware is stored in the normal world. The

```
2e088: 21 9b         ldr r3, [sp, #132]
2e08a: 00 2b         cmp r3, #0
2e08c: 0a d1         bne #20 <xProvisionCertificate+0x120>
2e08e: 0b 48         ldr r0, [pc, #44]
2e090: f7 f7 12 f9   bl  #-36316
2e094: 0a 9b         ldr r3, [sp, #40]
2e096: 5c 6d         ldr r4, [r3, #84]
2e098: 0d a9         add r1, sp, #52
2e09a: 24 9b         ldr r3, [sp, #144]
2e09c: 06 22         movs  r2, #6
2e09e: 03 98         ldr r0, [sp, #12]
2e0a0: a0 47         blx r4
```

Figure 2.5: Thumb-2 16-bit and 32-bit Instruction

system would trigger an alarm if the normal world gets access to the secure world. By using the

Although some scientists point out that ARM TrustZone can be compromised in some cases [6], we rule out this situation from our threat model.

## 2.5 Related Work

Control-flow integrity has always been a hot topic ever since it was first introduced by *Abadi, et al.* [1] in 2009. There are lots of work on developing CFI. Initially, CFI policies are coarse-grained and enforced CFI on static CFGs which are an overestimation of the program such as NaCl-JIT [3]. Then people tried to find ways to construct a more detailed CFG such as PT-CFI [18] and $\tau$CFI [17]. However, fine-grained CFI is not a panacea for control-flow integrity. They cannot achieve the purpose of getting a unique-code target for each indirect control transfer. Then, PITTYPAT [12] and $\mu$CFI [19] used hardware to dynamically construct CFG which improves the path sensitivity. CFI-LB [21] and OS-CFI [22] payed more attention on adding more context to do instrumentation. They add context information of function call site and the address of definition to each function. Most of the aforementioned mechanisms do their instrumentation on source code, and there are several mechanisms focused on doing instrumentation on binary. $\mu$CFI [19] and $\mu$RAI [2] can be used on binary. Nevertheless,

$\mu$CFI requires an Intel processor and $\mu$RAI mostly focused on protecting function return. Our mechanism focused on giving both forward and backward edge protection for real-world real-time embedded systems.

We give a detailed comparison of recent work on CFI in Table 2.2.

Table 2.2: CFI Comparison

| **CFI** | **$\pi$CFI** | **$\mu$CFI** | **OS-CFI** | **$\mu$RAI** | **PARTS** |
|---|---|---|---|---|---|
| Architecture | x86_64 | x86_64 | x86_64 | ARMv7-M | ARMv8-A |
| Coverage | Every ICT | Selected syscalls | Every ICT | Every ICT | Every ICT |
| Context/Path Sensitivity | Function call site | Execution path and constraining data | Origin of pointers and objects | Function call site | Each pointer |
| Metadata storage | Sandboxing | Stored in a different process | Intel MPX | MPU | Pointer Signing |
| Hardware Feature | - | Intel PT | Intel MPX, Intel TSX | MPU | ARM PA[15] |
| Online (CFG) | ✓ | ✓ | | | - |
| Offline (CFG) | ✓ | | ✓ | ✓ | - |
| Binary | | ✓ | | ✓ | |
| Source Code | ✓ | | ✓ | | ✓ |

# Chapter 3

# Threat Model

In our threat model, we consider a powerful attacker has full knowledge of the structure, features, and memory structure of the embedded development board. The attacker can easily get full access to the memory, including doing arbitrary read and write to the memory. Although it is hard for the attackers to easily get all memory control of the system, to validate the feasibility and functionality of our mechanism, we constructed a very powerful attacker. Randomization techniques such as ASLR [20] can be disabled. DEP [14] is deployed. Thus, they can direct the innocent control-flow to their target address(shellcode or chained gadgets) and issue the control-flow hijack attack as they wanted. Our goal of protection is even though the attacker changes the control flow, we can detect and send an alarm to the system.

Our mechanism's trusted computing base includes:

- The processor and other hardware on the development board.
- Our mechanism's instrumentation.
- The running firmware on the development board is not malicious.
- OS Kernel is not malicious.
- We fully trusted ARM TrustZone isolation.

We assume that MPU and ARM TrustZone are provided in the system. We use MPU to set aside privileged code and unprivileged code to enable DEP. ARM TrustZone is prevalent in the ARM Cortex-M development board. It is used in our system for sandboxing, which isolates the metadata from the attackers.

# Chapter 4

# Implementation

The goal of our mechanism is to use control-flow integrity to detect and prevent attackers from doing arbitrary code execution. We use a high-efficiency development board LPC55S69x which uses an ARM Cortex-M33 processor for our design and evaluation. LPC55S69x does not have Memory Management Unit and only has 640 KB on-chip flash memory. For our system we designed:

- An IoT application using MQTT, Amazon AWS, and Lambda letting users control a LED light on the development board using Amazon Alexa voice control.
- Used Capstone to realize binary analysis and rewriting.
- Add secure runtime to the refined binary.
- Retrofit Ghidra to get the control-flow graph from binary.

Our mechanism is based on binary. For those source code based mechanisms, they would design their own compiler using LLVM and use sandboxing or hardware-assisted ways to protect the metadata.

```
2e09a: 24 9b       ldr  r3, [sp, #144]
2e09c: 06 22       movs r2, #6
2e09e: 03 98       ldr  r0, [sp, #12]
2e0a0: a0 47       blx  r4
2e0a2: 21 90       str  r0, [sp, #132]
.
.
.
2e756: 04 b0       add  sp, #16
2e758: 70 47       bx r3
2e75a: 00 bf       nop
```

Figure 4.1: Indirect Control Transfer Instruction

## 4.1 Binary Instrumentation

We used Capstone[1] to analyze binary and rewrite the binary to add the instrumentation. Capstone is a lightweight disassembly framework that is compatible with various platforms and architectures. It is suitable for Intel X86 and ARM instruction for both 32-bit and Thumb mode.

After Capstone dissembling each instruction, we can get the mnemonics for each instruction. For forward-edge instructions, we care about two kinds of mnemonics: `blx` and `bx` as in Figure 4.1. For each sensitive instruction of `blx` and `bx`, we would like to write a hook function that will check the current branch target with the CFG to see if the indirect transfer is legal or not.

Instead of doing an indirect jump, we will rewrite the instruction to do a direct jump to the hook function. Rewriting 32-bit instruction is easy. However, it is hard for ARM Cortex-M to do this since the ISA is Thumb-2 ISA. `blx` and `bx` are 16 bit instruction and `bl` is 32 bit instruction. We will discuss this problem in detail in Chapter 4.3.2 and show how our mechanism resolves this issue.

[1] https://www.capstone-engine.org/

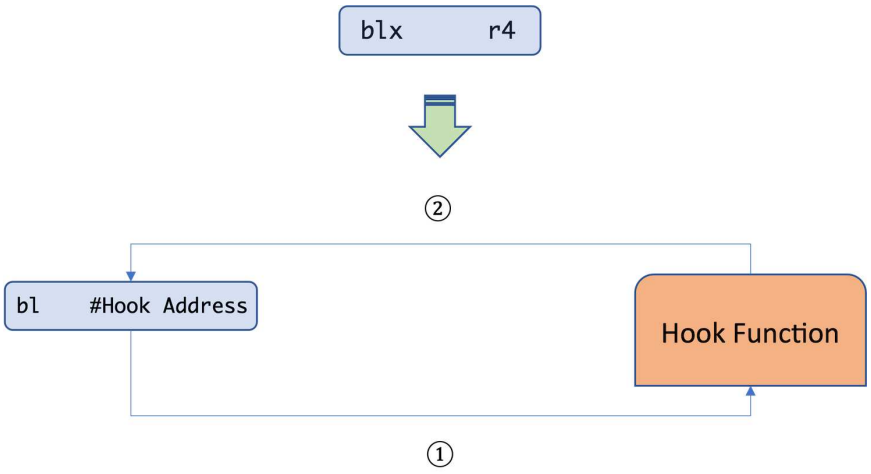


Figure 4.2: Binary instrumentation

## 4.2 System Design

Our system design can be shown in Figure 4.3. We first used Capstone to get the original branch target for each indirect control instruction. After rewriting all the indirect control transfer instructions, we can have the instrumented binary file. We also patched a set of assembly files of running system runtime to the binary. Finally, we used ARM TrustZone as a kind of sandboxing method to protect our metadata and secure runtime.

## 4.3 Smart-home Applications

Our development board LPC55S69 can either be a bare metal system or a real-time embedded system that includes kernel inside. In our implementation, we use FreeRTOS as our kernel to realize real-time tasks. We used MQTT and AWS IoT to design an application in which you can control the lights on the development board with voice control. It can either be used to be one aspect to test our system's runtime overhead or test whether our work is feasible for the embedded system or not.

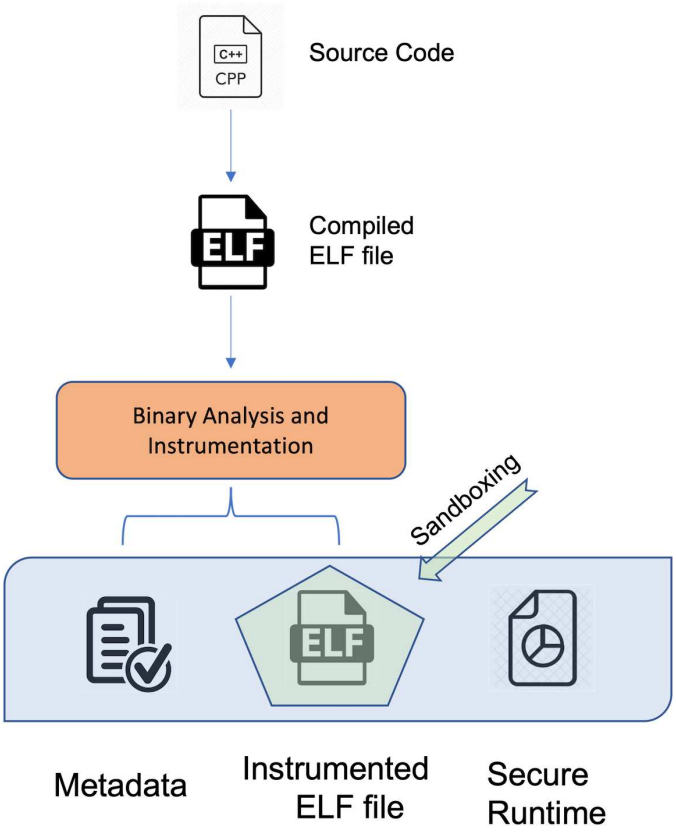


Figure 4.3: System design

### 4.3.1 Binary Analysis

In this section of work, we need to analyze the binary to get (1) the original target before we rewrite the indirect jump instruction. We will store it in a hash table and put the hash table outside the sandbox, (2) use a reverse engineering tool to get the control-flow graph.

We use Capstone and wrote 800 lines of code to get retrieve the hash table, and rewrite the binary. Then we use Ghidra[2], a software reverse engineering suite of tools to construct a full CFG. Finally, we store the CFG outside of the sandbox with other metadata.

[2]https://ghidra-sre.org/

### 4.3.2 Binary Rewriting

In the process of binary rewriting, we encountered a challenge in our project. To enforce our CFI policy, we need to rewrite the instruction with a `branch with link(bl)` to the address of hook function for each indirect control transfer. If our testbed is 32-bit ARM ISA, then there is no problem when doing the binary rewriting.

However, since our testbed is on ARM Cortex-m33, it uses a Thumb-2 instruction set where instructions are a hybrid of 16 bit and 32 bit. Indirect control transfers are `blx` and `bx` which are 16-bit instruction and `bl` is 32-bit instruction to hold the address. Rewriting 16-bit instruction to a 32-bit instruction will change the whole memory layout of the program as shown in Figure 4.4. This way of instrumentation is feasible by moving all the instructions to larger memory addresses. However, it is hard to solve the relative addressing the issue for instructions such as `ldr r3, [pc, #204]` since it contains a PC-relative instruction.

To solve this issue, we look through the details of the disassembly and found a solution without changing the memory layout of the program. First, we found that a large number of `blx` and `bx` instructions have a 16-bit `nop` afterwards, we can use the `nop` to construct a 32-bit `bl` instruction.

Then, for those `blx` and `bx` instructions which do not have `nop` afterwards, we replace the instruction with `b` instruction. Unconditional `branch` instruction in ARM Thumb-2 allows 11 bits to store the label. Since we need to return to the original place for indirect control transfer from the hook function, for each indirect control transfer we define its unique label. For rewriting the binary, we identify the `blx` and `bx` instructions bit code which is `01000111(L)Rm000` and change it to `1101100111110111(bl)` for each bit. We do not need to disassemble and reassemble the binary as many other mechanisms usually do. And we do not need to change the memory layout of the program.

After getting the refined binary, we used *blhost* to rewrite the binary to the development board and use *Putty* as the serial terminal to get feedback from the embedded system.

```
2e09a: 24 9b   ldr r3, [sp, #144]
2e09c: 06 22   movs  r2, #6
2e09e: 03 98   ldr r0, [sp, #12]
2e0a0: a0 47   blx r4              -->  2e0a0: d9 f7 92 fb   bl  HOOK_LABEL
2e0a2: 21 90   str r0, [sp, #132]  -->  2e0a4: 21 90         str r0, [sp, #132]
.
.
.
2e756: 04 b0   add sp, #16         -->  2e882: 04 b0         add sp, #16
2e758: 70 47   bx  lr              -->  2e884: d9 f7 92 fb   bl  HOOK_LABEL
2e75a: 00 bf   nop                 -->  2e888: 00 bf         nop
```

Figure 4.4: Indirect Control Transfer Instruction

### 4.3.3 Secure Runtime

Runtime plays the role of checking when the program runs. We embedded the C and assembly inside the source code and compiled it to the binary considering we have the source code available. The runtime can also be patched to the binary without being added to the source code then compile. Our mechanism chose the first option.

# Chapter 5

# Evaluation and Security Analysis

In this section, we will evaluate our CFI enforcement technique from two major aspects: security and time. We will analyze whether our mechanism can protect the system from control-flow hijack attacks or not. We will also talk about the granularity of our mechanism can protect the system. What kind of control-flow bending our system can detect and what cannot. Then, we will evaluate the runtime overhead and code size increase for our mechanism. It can either show the performance or the applicability of our technique. Finally, we will elaborate on the limitation of our work and discuss future works.

## 5.1 Security Analysis

Since we do not implement shadow stack in our mechanism, our mechanism can only protect forward-edge control-flow hijacking attacks. To check whether our work can enforce the forward-edge CFI, we construct two modern forward-edge control flow bending attacks for evaluation. We also introduce the idea of using *Unique Target* to measure security, that each indirect control transfer can only have a target. We quantify the security guarantee using the number of Equivalent Classes(ECs). If there are violations in the program, our mechanism will print a warning message in the serial terminal. It will not terminate the program since the limitations of the CFG, we will discuss more this in Chapter 5.3.1.

The first attack we test is a double-free attack shown in Figure 2.2 which corrupted the program's memory structure and allow arbitrary writing in the memory. The second example

```
parseString(char* s){}

void exploit_2(char* str){
   char buffer[18];

   strcpy(buf, str);
   parseString(buf);
}
```

Figure 5.1: Stack Overflow

is the format string attack shown in Figure 5.1. Our mechanism successfully detects those attacks.

Equivalent Class is a set of targets that are allowed by indirect control transfer. Coarse-grained CFI has limited time of ECs since they allow multiple targets per indirect control transfer. The optimal result is having the number of ECs equivalent to the number of indirect control transfers which achieve the goal of the unique target.

We measure the total number of Indirect Control Transfer(ICT) without `return` in our real-time program and ECs according to the CFG getting from the binary. The result is shown in Table 5.1.

Table 5.1: Number of ECs after Instrumentation

| **Name** | **Number** |
|---|---|
| **Number of all ICT** | 360 |
| **Number of ECs** | 302 |

We can see that in the worst case, we will have 58 ICTs have two targets so that we have at least 244 ICTs have a unique target. The number of which is bigger than the number of half of all ICTs. But Table 5.1 also means that our mechanism cannot achieve a unique code target. This is since the CFG constructed by the binary is not accurate enough. However, the reality is the most strict mechanism OS-CFI [22] to date cannot achieve the unique target even if they added lots of context information in their mechanism. Since non-control

data attacks can also divert the control flow and CFI cannot defend against non-control data attacks. We can improve our mechanism by improving the performance of CFG.

## 5.2 Runtime Overhead

Runtime overhead is the most important issue to put our focus on. In modern embedded systems' applications, people always care about the reaction time of their devices rather than the security aspects. Time is also the most critical element in real-time applications. If the mechanism cannot meet the requirements for real-time tasks, it is not reasonable for the mechanisms to be employed even if the security is guaranteed. To evaluate the runtime overhead, as every paper did, we use a benchmark called CoreMark to show the runtime overhead. We also evaluate the system using a real-time application to show our work performance. Since there are limited resources in embedded systems, we also show the code size measurement of our work.

### 5.2.1 CoreMark

CoreMark[3] is an industry-level benchmark that is usually used to measure the performance of microcontrollers. We measured the applications given by CoreMark 20 times and got an average of 5.31% of runtime overhead.

### 5.2.2 Real-time Task Measurement

To test our system applicability, we designed a voice control task using AWS and FreeRTOS. We tested this task for lighting up the led and turning off the led consecutively 100 times and measured the average execution time. The measurement result is shown in Table 5.2. Our mechanism has a runtime overhead of 2.96%.

[3]https://github.com/eembc/coremark

Table 5.2: Real-time Task Performance

| With/Without CFI | Latency(ms) |
|---|---|
| **Normal Program** | 1150 |
| **CFI Enforced** | 1116 |

### 5.2.3 Code Size Measurement

We also measures the increase of the ELF file size, and we found that we only have an increase of 1%. I think this is since there are not so many indirect control transfers in the firmware which is also the case in reality.

Table 5.3: Code Size Performance

| With/Without CFI | Size(Bytes) |
|---|---|
| **Normal Program** | 376,164 |
| **CFI Enforced** | 380,004 |

## 5.3 Discussion

In our work, we present a control-flow integrity enforcement technique on forwarding edges on real-time embedded systems using binary instrumentation. Our work has a runtime overhead of around 5.31% for CoreMark and 2.96% for a real-time voice control task. We will discuss the limitations here and will propose the directions of future works.

### 5.3.1 Limitations

Although our work performed great in runtime overhead, it superiors others at the cost of lacking enforcement on backward edges. Past works [22][30] used shadow stacks as a

protection to enforce the indirect control transfer on `return`. However, the shadow stack poses great runtime overhead on embedded systems. Many works [2][27][28] worked on designing a retrofit for shadow stacks, and others find other ways to protect return instruction without using shadow stacks.

Multi-threading is also a direction we did not consider. As claimed by *Xiaoyang, et al.* [40], almost every work that has been proposed did not solve the Time-of-Check Time-of-Use(TOCTOU) vulnerability in a proper, secure, and efficient way. Leaving this vulnerability unsolved may give the attackers a chance to get leaked information and compromise the system.

Another limitation of our work is the control flow graph enforced by CFI is not accurate enough since we just use a reverse engineering tool to recover the control flow graph. Although constructing an accurate CFG is another scientific research topic, and is not our main research goal, using an accurate CFG can improve the security guarantee of our mechanism.

### 5.3.2 Future Works

Aside from integrating the shadow stack and adding multi-thread protection, one direction we can work on is to get an accurate control flow graph from binary. This is a hard question still needs to be solved regarding computer security.

Besides, our work assumes that the OS kernel is not vulnerable. However, it is not always the case. We need to find ways to handle context switches and exception handling. For context switching, registers may store their intermediate value in the memory during context switching, and an adversarial may use another thread to tamper this value to get root privilege or get information leakage. For exception handling, since the attacker can do arbitrary write during the exception to disable MPU thus to disable DEP.

Finally, we can find some more efficient ways to do the binary instrumentation to decrease the space for instrumentation. Many embedded systems are like our systems which do not have large memory on-chip. If the firmware is very large, then it would be better if the binary rewriting have less code size increase.

# Chapter 6

# Conclusion

Embedded systems are vastly adopted in different fields in our life. However, embedded systems are not safe since they usually use unsafe C and C++ languages. An attacker can easily redirect the control flow of a pointer to execute arbitrary code and get root control of the system. Control-flow integrity is an efficient way to protect control-flow hijacking attacks. To protect the system and also achieve small latency, we propose this work to enforce control-flow integrity on resource constrained real-time embedded systems.

Our system is implemented for ARM processors with a Thumb-2 instruction set. Our testbed is LPC55S69. To enforce control-flow integrity, we first used a reverse engineering tool to get a control-flow graph from the binary. Then we use Capstone to analyze the binary and rewrite the binary with CFI enforcement. Our work utilized ARM TrustZone to realize sandboxing, Memory Protection Unit(MPU) to realize Data Execution Prevention(DEP) to protect metadata used by CFI enforcement. Our work has an optimal runtime overhead of around 5.31% and 2.96% for real-time tasks. We also pointed out the limitation of our work and some future research directions in our work.

# References


[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[2] Naif Saleh Almakhdhub, Abraham A Clements, Saurabh Bagchi, and Mathias Payer. urai: Securing embedded systems with return address integrity. In *Network and Distributed Systems Security (NDSS) Symposium*, 2020.

[3] Jason Ansel, Petr Marchenko, Ulfar Erlingsson, Elijah Taylor, Brad Chen, Derek L Schuff, David Sehr, Cliff L Biffle, and Bennet Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 355–366, 2011.

[4] Emery D Berger and Benjamin G Zorn. Diehard: Probabilistic memory safety for unsafe languages. *Acm sigplan notices*, 41(6):158–168, 2006.

[5] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.

[6] Sebanjila Kevin Bukasa, Ronan Lashermes, Hélène Le Bouder, Jean-Louis Lanet, and Axel Legay. How trustzone could be bypassed: Side-channel attacks on a modern system-on-chip. In *IFIP International Conference on Information Security Theory and Practice*, pages 93–109. Springer, 2017.

[7] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160, 2006.

[8] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. Hcfi: Hardware-enforced control-flow integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 38–49, 2016.

[9] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[10] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. Hafix: Hardware-assisted flow integrity extension. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

[11] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2014.

[12] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient protection of path-sensitive control security. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 131–148, 2017.

[13] Aeon Flux. sudo 1.8.0 < 1.8.3p1 − sudo_debug glibc fortify_source bypass + privilege escalation. https: //www.exploit− db.com/exploits/25134/. 2013.

[14] Ying-chun Gao, An-min Zhou, and Liang Liu. Data-execution prevention technology in windows system. *Information Security & Communications Privacy*, 2013.

[15] Zaheer Gauhar et al. Pointer authentication for memory protection: Stack canaries and beyond. 2019.

[16] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. *ACM SIGPLAN Notices*, 52(4):585–598, 2017.

[17] Jens Grossklags and Claudia Eckert. $\tau$cfi: Type-assisted control flow integrity for x86-64 binaries. In *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings*, volume 11050, page 423. Springer, 2018.

[18] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 173–184, 2017.

[19] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R Harris, Taesoo Kim, and Wenke Lee. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1470–1486, 2018.

[20] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[21] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. Adaptive call-site sensitive control flow integrity. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 95–110. IEEE, 2019.

[22] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive control flow integrity. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 195–211, 2019.

[23] Andi Kleen and Beeman Strong. Intel processor trace on linux. *Tracing Summit*, 2015, 2015.

[24] Paul Kocher, Ruby Lee, Gary McGraw, and Anand Raghunathan. Security as a new dimension in embedded system design. In *Proceedings of the 41st annual Design Automation Conference*, pages 753–760, 2004.

[25] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, pages 81–116. 2018.

[26] Yang W Lee, Leo Pipino, Diane M Strong, and Richard Y Wang. Process-embedded data integrity. *Journal of Database Management (JDM)*, 15(1):87–103, 2004.

[27] Hans Liljestrand, Thomas Nyman, Lachlan J Gunn, Jan-Erik Ekberg, and N Asokan. Pacstack: an authenticated call stack. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[28] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. {PAC} it up: Towards pointer integrity using {ARM} pointer authentication. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 177–194, 2019.

[29] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. Ccfi: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 941–951, 2015.

[30] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 914–926, 2015.

[31] Moon Chan Park and Dong Hoon Lee. Random cfi (rcfi): Efficient fine-grained control-flow integrity through random verification. *IEEE Transactions on Computers*, 70(5):733–745, 2020.

[32] Mathias Payer, Antonio Barresi, and Thomas R Gross. Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 144–164. Springer, 2015.

[33] Philippe Robin. Experiment with linux and arm thumb-2 isa. In *CELF Embedded Linux Conf*, 2007.

[34] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

[35] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64. IEEE, 2015.

[36] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015.

[37] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[38] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[39] Johannes Winter. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30, 2008.

[40] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W Hamlen, and Zhiqiang Lin. {CONFIRM}: Evaluating compatibility and relevance of control-flow integrity protections for modern software. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1805–1821, 2019.

[41] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*, pages 559–573. IEEE, 2013.

[42] Mingwei Zhang and R Sekar. Control flow integrity for {COTS} binaries. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 337–352, 2013.