IAIK

SCIENCE ▪ PASSION ▪ TECHNOLOGY

TU Graz

Lukas Furtner

Exploiting Fine-Grained CFI for Data Integrity

BACHELOR'S THESIS

Bachelor's degree programme: Information and Computer Engineering

Supervisor

Robert Schilling

Institute of Applied Information Processing and Communications
Graz University of Technology

Graz, September 2022

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Lukas Furtner

Acknowledgements

This work would not exist without the help of the people mentioned below.

I would like to thank Robert Schilling, who supervised this thesis. His support for this thesis was priceless, as he was always open to questions and suggestions.

Furthermore, I would like to express thanks to my friends and fellow students, who encouraged me to proceed with this course of study as this was not easy all the time.

Finally, I am very grateful to my whole family. I want to thank all of them for the tremendous support and the steady encouragement that they offered me.

Abstract

Fault attacks became more and more critical in the past. As the ideas for new attacks on all kinds of devices, based on hardware and software, grow and grow, the research for countermeasures also rises. The injection of faults into devices makes it possible to break whole systems with the resulting errors. Although some countermeasures exist that can prevent such attacks, they mainly produce large overheads. These overheads can be in code size and runtime, or additional hardware is needed. Because of the large overheads, these countermeasures are often not practicable for small devices. That is because they do not have the place for hardware-based countermeasures nor the Central Processing Unit (CPU)-power for bigger software-based solutions. Furthermore, more than one attack vector can induce a fault in a system, which leads to the point that a combination of countermeasures might be needed, which leads to even more overhead.

In this thesis, we present a lightweight data integrity extension for known data at compile time on top of an existing Control-Flow Integrity (CFI) scheme. The extensions base is FIPAC, which is a purely software-based countermeasure for control-flow attacks, designed for Internet of Things (IoT) devices. With this extension, it is possible to protect systems against control-flow attacks and diverse attacks on data known at compile time. Because of the extension's flexible design, the user can decide which data should be protected. The protected data influences the internal cryptographically signed state calculation of FIPAC. Because of this, we transform a data error into an CFI error. Any subsequent state comparison of FIPAC then detects data errors and control-flow errors. The FIPAC extension, in general, is realized by modifying the FIPAC toolchain. The first part consists of modifying the LLVM compiler so that it supports an additional instruction in inline Assembly (ASM), which is named `inject`. With this instruction, it is possible to choose the desired known data that is protected. The second part is modifying the post-processing tool so that the chosen data influences the calculation of the new FIPAC states.
The evaluation with an Advanced Encryption Standard (AES) encryption from the `Embench` benchmark tests where the round counter is protected shows an additional runtime overhead of 0.0752% on top of FIPAC. To protect every possible known data, we observe an overhead of 0.2555%. According to code size, the overheads are 0.1217% and 0.7299%. Therefore our two-in-one extension fits perfectly for constrained devices like IoT devices.

Keywords: Data Integrity, CFI, FIPAC, Software Countermeasure, ARM

Contents

1 Introduction

These days, computers are very widely spread across the whole world. The variety of products that contain a computational unit is huge [Boh+03]. Such products can, for example, be devices that we use every day like laptops, mobile phones, or smartwatches. Although these examples might be the most known products, computers are placed in many more devices. Someone can find a computer nearly everywhere where some form of computation is needed. That includes traffic control systems, cars, smart home devices, or TVs. Also, modern cars contain about 130 computational devices to serve all their features.
In general, many of the chips that we use for computations are ARM based [Lim]. ARM is a CPU type that uses the Reduced Instruction Set Computer (RISC) architecture. Because of the RISC architecture which provides fewer instructions than the Complex Instruction Set Computer (CISC) architecture, ARM chips do not need that many transistors inside [FL13]. That leads to small die sizes and low power consumption, making ARM chips suitable for small and battery-driven devices like mobile phones or smart home devices. Especially in IoT devices, ARM covers the majority of chips used in this domain.
IoT devices [Mad+15] are devices that connect to the internet and can therefore communicate with each other. This communication allows the devices to get more information from their surrounding. For example, an IoT fridge may then be able to reorder consumed food products from a supermarket. Such complex tasks are not possible without the interconnection between things. That is also the case why IoT is a fascinating concept that will grow in the future.
However, IoT devices also bring some downsides regarding security. If an attacker manages to hack only one of these devices, he may be able to enter the whole system. As IoT devices are usually deployed in the field, physical attacks are possible. Furthermore, there are already some known physical attacks with which an attacker can break these devices. One class of physical attacks are called fault attacks [Bar+06]. Fault attacks use methods like under- or overpowering a system for a short time in order to change the operating conditions of a system [KSV13]. As a result, bit flips may be introduced somewhere in the system. That can happen in various places of the system, for example, in the memory or while transmitting instruction data via the internal bus. When injected correctly, these bit flips can bring a system into a faulty state, which it can not recover. As a consequence, different things can happen to this system. For example, a system stops execution, reveals sensitive data or does faulty computation with secure data [TSW16]. Even worse would be that the attacker gains root privileges [TM17]. Then he can interfere with the network participants to which the device is connected and mount further attacks.

In order to counteract fault attacks, countermeasures are needed [Wit08]. Countermeasures add additional features to a system in order to protect against faults. Countermeasures can be software-based as well as hardware-based. There can also be hybrid techniques that prevent a specific fault attack. A hardware-based countermeasure against an electromagnetic vulnerability would, for example, be to cover the whole system with a Faraday's cage [Aum+02]. This cage around the system, made of electrically conductive material, protects the inside against electromagnetic waves. On the other hand, a software-based countermeasure would be double-checking or redundant execution [MSY06]. That means that the program's calculation parts are always performed twice with a different data location inside the memory. Then every defined time interval, or when the program performs a decision, the two calculations are compared if they still match. That works because timing is crucial for an attack, and it is pretty hard for an attacker to hit the exact time slot for a fault correctly. To induce this fault now for a second time at the exact right timing is considered quite hard.
A specific countermeasure can often only prevent the system from exactly one type of fault attack. Therefore it is often needed to combine a variety of countermeasures in order to get the required hardness against the specified fault attacks. The realization of the protection can therefore be very challenging when considering small devices like IoT devices. However, these devices can also contain sensitive data or connect to other devices which contain sensitive data. If someone attacks an IoT device, he could extend the attack to use the IoT device in order to get access to other parts of a connected system. Therefore, these devices also need protection against fault attacks, especially because they are physically accessible so that hardware-based attacks can happen to them.
Two of the most important things to protect are data or the control-flow [Aum+02]. Corrupted data on one side can lead to incorrect checks. If, for example, the attacker flips an admin flag in memory, a privilege escalation will happen on the next check of this flag. On the other hand, if the attacker manipulates the instruction pointer, so the program skips the check, the same thing can happen. This second approach manipulates the control-flow of a program. In order to avoid the mentioned attacks, the developer needs to consider countermeasures that provide Control-Flow Integrity (CFI) and data integrity.
However, countermeasures often come with high overheads [MSY06]. These overheads can be so high that they are just not feasible, especially when it comes to constrained devices. Therefore, such systems need a special treatment where a relatively lightweight or at least a solution designed for exactly these systems is needed. In addition to this, protecting against more than one type of fault attack could need more than one countermeasure. Then the overhead rises even more and most probably gets too expensive. One solution to make the overhead of countermeasures more feasible is to combine different countermeasures while not really raising the existing overhead of the first countermeasure. The method that we present below does achieve this.

Contribution

In this work, we present a technique for exploiting fine-grained CFI for data integrity on known data during compilation time. Our extension is based on FIPAC and is highly flexible, as the user can decide which compile-time known data should be protected.
We transform data errors into control-flow errors by XORing the protected data to the internal state-based CFI scheme. The protected data then influences the CFI state. That leads to an incorrect CFI state if the data is wrong during execution. Therefore it is possible to check for control-flow violations and data violations when the control-flow state gets compared with the precalculated one.
In order to make the flexible data protection possible, we extend the LLVM compiler by a new instruction called `inject` which has two parameters. The parameters refer to a register and a value. With this instruction, it is possible to pass a register that has to match the corresponding value when executing the inject command. If this is not the case, the program will abort when checking the next internal CFI state. The value passed in the `inject` instruction gets written into a metadata section of the binary. From there, the modified post-processing tool of FIPAC does read the value. This value has an influence on the new CFI state calculation. As the post-processing tool writes these states to the desired state checks in the binary, a corruption of the data will result in an invalid CFI state.
One possible use case of our extension is, for example, to protect the round counter of AES on top of the provided CFI scheme to harden the calculation against faults.
We tested our implementation on the existing AES implementation of the `Embench` benchmark test. To protect every known data value at compile time, we measure an additional overhead of 0.2555% in run time and 0.7299% in code size. Only protecting the loop counter, we discover an additional overhead of 0.0752% in run time and 0.1217% in code size.

Outline

The further part of this thesis is structured as described below. In Section 2, we give an overview of different fault attacks and countermeasures. Also CFI is explained more in detail in Section 2. We describe the design used for the realization of data integrity in Section 3, and we present the details of the implementation in this section. In Section 4 we show the evaluation of the implementation. We discuss the requirements and usage of the extension in Section 5. In the last Section 6, we conclude the whole thesis.

2 Background

In this section, we give a basic overview of fault attacks. Then we describe some successful attacks that happened in the past. In Section 2.2, we discuss some software and hardware-based countermeasures against fault attacks.

2.1 Fault Attacks

In fault attacks, an attacker modifies the operating conditions of a device. That can, for example, be a short power spike on the supply voltage. As a result, unintentional bit flips can happen in the system [KSV13]. These bit flips can then lead to a faulty system state. With this faulty system state, unintentional things which are not within the specification of the system can happen. That can, for example, be leakage of sensitive data or an unsafe calculation of a cryptographic key. Such unintended behavior was already discovered a long time ago, around the 50s to 70s [Bar+06]. For example, the electronic equipment began to show anomalies when launching a nuclear weapon. The source for these anomalies was probably strong electromagnetic radiation while deploying the nuclear weapon. Another unintended behavior happened when researchers used computers in space. As discovered later, the alpha particles in the space influence semiconductor electronics [Bal15]. In these early times, nobody thought that the bit flips could break whole systems, as the researchers considered this topic as unimportant.

2.1.1 Hardware-Based Fault Attacks

In the following part, we list a variety of hardware-based attack vectors that can induce fault attacks [Bar+06].
One type of them are power attacks [KSV13]. Changing the supply voltage of a generic device can lead to instruction skips or something related, like misinterpreting instructions that can break the whole system. That is one of the less expensive methods because power spiking or under-powering a system is not complex.
Another attack vector is based on varying the temperature of an device [Bar+06]. If an attacker can control the temperature of a device, another type of fault can occur, as the electronic elements of a device are temperature-dependent. For example, a resistor gets a higher ohmic impedance as the temperature rises. The same effect appears in semiconductors for a lower temperature. With these aspects, it is possible to get faulty executed instructions that can lead to breaking a system.
The next attack vector is based on clock glitching [KSV13]. Varying the period times of the clock signal may lead to the execution of the following instruction before the old one has finished. As instructions often depend on the one before, like when executing a

conditional branch, the consequence could be executing the wrong branch. This attack is easy to perform if the device uses an external clock.
In addition to this, an attacker can also preform an attack with the help of lasers [Bar+06]. Electronic devices are sensitive to light, as ionization can take place using a laser beam. This fault makes it possible to create bit-flips nearly everywhere where a transistor is in use, like in memory. With the right wavelength, it is possible to get the laser through small protection cases of chips and cause, for example, memory violation. If a program uses this memory location for a password check, the check could succeed even if the password is wrong.
The last attack vector that we mention is based on electromagnetic fields [KSV13]. With the principle of induction and a created electromagnetic field, an inducted voltage can lead to wrong executions or bit-flips in various system components. An attacker can again use these bit-flips to break a system.

2.1.2 Software-Based Fault Attacks

Fault attacks are not only possible via hardware. There also exist some software-based attacks. We mention two of them below.
One attack is called Rowhammer [MK20]. This attack exploits the physical behavior of DRAM cells. The main issue here is the design of DRAM, which organization is in cells and rows. The data cells are physically very close to each other. Because of the short distance, an electronic coupling exists to nearby cells. Therefore, a small current gets induced to nearby cells on each write or read access. So if reading or writing the same value on a memory region repeatedly, nearby memory cells can be flipped. This attack still exists in current DRAM memories. For this type of attack, it is not even needed to have direct access to the system, as there exist remote-based versions like Throwhammer [Tat+18] or Nethammer [Lip+20] that can perform the attack via the network interface.
Another attack is called Plundervolt [Mur+20]. The vulnerability which provides this attack is the possibility to scale down the operating voltage of a processor with a method called Dynamic Voltage Scaling (DVS) [BB00]. With this design, it is possible to turn down the voltage of chips via a register value, for example, to reduce energy consumption. However, an attacker can use this voltage scaling to induce a faulty computation. With Plundervolt, it is possible to break the integrity of Intel Software Guard Extensions (SGX). The attack scales down the voltage while the enclave executes code. That introduces a wrong execution, and as a result, Intel's SGX encryption/authentication can be broken.

2.1.3 Overview of Different Attacks on Generic Devices

To summarize the topic of fault attacks, we mention some performed attacks on generic devices.
The first attack was performed in DirecTV (DTV) with a device called DirecTV Unlooper [Bal15]. In pay-TV, a smart card can grant access to the TV stream. This smart card has a small microprocessor in it, which performs the decryption of the TV stream

in order to be able to watch it. In DTV, expired cards once were disabled by leading the microprocessor's execution into an infinite loop to stop the streaming. With the use of a fault attack, it is possible to escape this loop. After escaping the loop, it is possible to watch the TV stream again.
For automotive firmware, there also exist fault attacks. Researchers found two ways to extract the complete secret firmware of an electronic control unit. The first way uses the `ReadMemoryByAddress` service, where no timeout for the wrong security key exists. That leads to the case that the attacker can perform many voltage glitch tries and does not get a penalty for it. With this method and faulting the security check, an attacker can read 64 bytes of memory. This approach takes about three days until an attacker can extract the full firmware from the system. The second attack exploits the debug interface of the electronic control unit. With available public information about the device, it is possible to enable this debug interface by setting a specific pin. The attacker can use fault value injection to skip the check for wrong memory destinations and extract the target firmware when using this interface. This second approach is much faster, and within a few hours, the whole firmware is known and only needs some reverse engineering.
There also exists a fault attack for the Xbox360, called the `Xbox 360 reset glitch hack` [Fre]. This glitch is a hardware-based timing attack. The developer of the attack found out that the CPU slows down significantly when sending the `CPU_PLL_BYPASS` signal. If a short reset impulse appears while the system is in slow execution, the CPU does not reset and instead acts differently as in the normal process. With this, it is possible to change the return value of the `memcmp` function in a way that it always returns that the two compared memory regions match. The Xbox360 uses the `memcmp` function to check the hash of the boot loader. With this bug, it is possible to run almost any code as the hashes will match all the time.

2.2 Countermeasures

In order to avoid exploitation of fault attacks, a system needs countermeasures. Countermeasures prevent an attack on one specific attack vector. That means that the attack is not executable, or at least not feasible anymore. Countermeasures can be hardware-based as well as software-based.
One general countermeasure is spatial or temporal redundancy [LJK21]. Both methods use redundant computation. That means the system performs computation more than once and compares the different results. If they match, everything is fine. However, if the results do not match, the computation can be stopped, or the system can at least perform some action against the fault. As introducing the same fault twice is very hard, this countermeasure works pretty well against various faults on calculations.
Spatial means that a developer accepts additional space of the chip to perform the redundant computation. For example, an additional CPU is placed on the Printed Circuit Board (PCB). A temporal redundancy does not need additional space, as this type of countermeasure performs the computation twice on the same CPU. Therefore temporal

redundancy introduces an overhead in the execution time.
We discuss additional hardware-based and software-based countermeasures below.

2.2.1 Hardware-Based Countermeasures

This section contains some hardware-based countermeasures to prevent fault attacks [KSV13].
One of the most common hardware-based countermeasures is a shield that covers sensible parts of the chip which do computation. There are two different realizations of shields. These are called active and passive shields [Mun+19]. A passive shield consists of a metal plate, whereas the active shield is some mesh around the chip. Through this mesh, a small signal flows so that it is possible to detect a corruption of the mesh. If a corruption takes place, the sensible parts of the chip can be informed or shut down. The idea behind shields is that there is no direct access to the hardware that does calculations, so a modification of it can not occur.
Another spatial hardware-based countermeasure is the placement of various detectors on a chip. These detectors may detect, for instance, lights or anomalous frequencies. As it is possible to introduce faults with lasers, light detectors can counteract them [Mat+20]. Someone can use the same approach to counteract clock glitching, as a sensor can detect these glitches because of their anomalous frequencies. If a detector then detects an anomaly, the system can react accordingly, for example, by deleting sensitive memory.

2.2.2 Software-Based Countermeasures

In this section, we describe some software-based countermeasures to counteract fault attacks.
The first countermeasure is based on constants and decisions [Wit08]. Software developers often use `0` for false, such as `1` for true. That is a bad aspect regarding fault resistance, as fault attacks mainly flip bits, or set them all. Better constants to use are, for example, `0xA6` for true and `0x59` for false, as the Hamming distance is 8 (all 8 bits differ from the other constant). That makes it very difficult for such an attack to manipulate the correct bits. The same aspect counts for sensitive branch decisions. It is relatively easy to convert a `0` to a `1`. For more complex numbers, this is not the case. Considering a bigger data type is an option to improve security even more.
Another countermeasure is the detection of manipulated data [GC14]. A developer can use a hash function to prevent the manipulation of private data between initialization and later usage. The system can detect any manipulation by hashing the data at a later point in execution again and checking if the resulting hash matches the old one. An excellent point to create the first hash is directly after initializing or reading new data. Before using the data, a system can then check if a manipulation happened. If a manipulation occurs, the system can then react accordingly.
Failing by default can also harden a system against fault attacks [Wit08]. This is because in programs there are often parameters with specific limits in their value range. Therefore `switch` constructions often tend to use the default case for the last possible values without a proper check. That makes a program more vulnerable to fault attacks, as, for

example, data manipulations that bring a parameter out of its defined range run into the default case. Therefore `switch` constructs should always have a default case that leads to a failure so the program can detect out-of-range manipulations. The same aspect is important for `if-else` constructions.

2.2.3 Countermeasures for Control-Flow

Data is not the only thing in a system that an attacker can manipulate with fault attacks. It is also possible to attack the control-flow of a program. That means that a program's execution flow can be changed so that it behaves in a way it was not designed for. In order to prevent a control-flow attack, a countermeasure should prevent any malicious redirection of the execution flow. If this is achieved, Control-Flow Integrity (CFI) is given. In general, CFI can further be split into coarse-grained, and fine-grained CFI [Aba+09]. In this section, we first briefly introduce the basics of a control-flow. Then we describe coarse-grained and fine-grained CFI more in detail. Finally, we describe how CFI with FIPAC works.

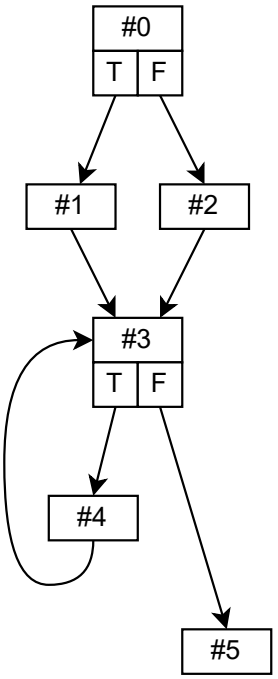
Control-Flow Basics

In general, a program consists of many instructions. These can either be instructions that do not affect the control flow or instructions that have an impact on the control-flow of the program. For example, in the programming language C, the code line `c = a + b;` is compiled to some sequential operations, whereas statements like "if" or "while" will have a jump or branch instruction in their ASM code. When executing sequential statements, the CPU fetches one instruction, decodes and executes it, and then raises the instruction pointer so that it points to the memory location of the following instruction. On the other side, when executing instructions that impact the control-flow, like a `branch` or `jmp` instructions, the program counter is changed to another memory location. Therefore the next executed instruction is not the one following in memory.
One way to visualize a control-flow is to use a Control-Flow Graph (CFG) [CHW02]. The CFG consists of nodes and pointers. Each node represents a set of sequential statements and is also called a basic block. Inside basic blocks, there are only instructions/statements that do not affect the control-flow, which means they are executed sequentially and do not contain control-flow instructions like `jmp` or `branch`. Whenever an instruction affects the control-flow, at least one pointer exists that starts from this location and points to every location where the program can continue. That means, for example, if there is an `if` statement, a pointer points to the basic block that executes if the statement inside is true. A second pointer points to the basic block that executes if the statement is false. An adaption is possible for other control-flow statements like loops or direct jumps.
In Figure 2.1b, we show a CFG that shows the control-flow of the C-Pseudo code in Figure 2.1a. In this graph, the blocks enumerated from 0 to 5 are basic blocks, purely containing logical computations. Constructing a CFG does not need much information about the actual code. That is because of the clear definition of control-flow changing statements. For example, an `if` instruction can only have two following basic blocks,

```
void function(...)
{
    // #0
    if(...)
    {
        // #1
    }
    else
    {
        // #2
    }
    // #3

    while(...)
    {
        // #4
    }
    // #5
}
```

(a) C Pseudo-Code.



(b) Control-Flow Graph.

Figure 2.1: Pseudo Code and its Control-Flow Graph.

the one if the condition is true and the one if the condition is false. The basic block with index `5` has no pointer pointing away from it, which means that there has to be a termination. That is the case in the function, as a return follows after this basic block. It is also possible to construct this graph for a whole program. For graphs of a whole program, the smaller graphs representing one function must be inserted every time the function is called in the code. For graphs of a whole program, someone must insert the smaller graphs representing one function every time a function call to this function exists in code.

Coarse-Grained Control-Flow Integrity

Coarse-grained CFI can protect a program from software-based attacks on the control-flow. That includes the integrity for code pointers of function calls, as well as forward edge or backward edge CFI.
Protecting backward edges means that return addresses of functions must only point to the real caller of the function. A solution for this is the usage of a shadow or safe stack for return addresses [SZW08]. For a safe stack, a separate stack that is protected from the original stack saves the return addresses. Therefore, there is a separation of the addresses and the rest of the local data stored in a program's stack. Then buffer overflows on local data do not affect return addresses anymore. A shadow stack uses an additional stack to

store addresses there in addition to the normal stack. Before returning from a function, the system compares the address from the shadow stack with the normal stack. Only if the addresses match the return succeeds. Otherwise, a control-flow manipulation will occur with the return. Therefore the program can react accordingly.
CFI on forwarding edges is a more complex topic, as it needs to ensure that only the correct function is allowed to execute. For this part, there exist four more precise levels of security.
The lowest level contains that only functions can execute after an indirect call. That ensures that a jump is only possible to actual entry points of functions. On this level, it is not possible anymore for an attacker to jump to an arbitrarily address in memory when attacking function pointers. For this level, a hardware-based realization would be Intel's Control-flow Enforcement Technology (CET) [SGS19]. This technique uses a new instruction called `endbr64`. This instruction is then placed at every starting point of a function by the compiler. As every possible entry point is marked, a call instruction only succeeds if the jump gets verified in the next cycle by the `endbr64` instruction. That avoids all calls to random positions in memory. CET is already supported by new Intel, or AMD CPUs. This level does not need much overhead, but the security aspect is still a little imprecise.
The second-lowest level only allows calling functions with the same signature as the function pointer. The realization of this level needs a little bit more overhead. A software-based CFI scheme would be the sanitizer for CFI on clang [Tic+14]. With the compile flag *-fsanitize=cfi*, the clang compiler places additional checks for control-flow violations into the code and only allows functions with the same signature. This level can already prevent some type-confusion attacks.
The third level only allows functions that can be assigned to the function pointer.
The fourth and highest level only allows a call to the function defined in the code. Any other jump to somewhere else than the expected function is forbidden. This level is fully precise as it provides complete code pointer integrity. On the other side, this level is also the most expensive one regarding overhead.
It is pretty hard for level three and four to achieve the desired code pointer integrity without double execution and special checks, which lead to a tremendous overhead. However, if it is possible to ensure no violation on code pointers, attacks like Return-Oriented Programming (ROP) [Roe+12] or Return To Libc (ret2libc) [El] are not possible anymore.

Fine Grainded Control-Flow Integrity

An attacker can also attack a system with hardware-based attacks. In addition to software-based attacks, fault attacks can also introduce instruction skips into a system. A more precise CFI scheme, called fine-grained CFI, can protect against this threat. These schemes focus on the instruction or basic block level.
In order to achieve CFI on the basic block level, software-based as well as hardware-based solutions already exist. However, most of the techniques are software-based approaches, as the significant advantage here is that the system does not need any

additional requirements. Furthermore, software-based approaches work on most of the existing processors. Hardware-based approaches would probably need a new CPU, so, for example, recompiling the code with the CFI extension in it is by far more practicable. Most of the techniques for basic block integrity use some signatures assigned to the different basic blocks. During run time, these signatures are then somehow compared with a current state that depends on the executed basic blocks to recognize faulty control-flows. One example that is software-based would be Enhanced Control-Flow Checking using Assertions (ECCA) [Alk+99]. This scheme uses different primes for the signatures. Some mathematical operations on them then perform confirmation of the signatures. If the scheme detects an error, it performs a division-by-zero exception. The system can then handle this exception and shut down sensible parts. Based on the same idea, but with optimization on performance, would be "Control-Flow Checking by Software Signatures" [OSM02], SWIFT [Rei+05], or "Yet Another Control-Flow Checking using Assertions" [Gol+03].
Purely hardware-based concepts are much harder to design, as the reference values are hard to get. On the other side, the software can stay the same, which can be helpful if, for example, some sources are not available anymore. In order to get the reference values, these solutions often contain a learning phase or need at least one execution that is not faulty. One hardware-based example is "Online Signature Learning and Checking" [MS91]. This concept generates the reference values during testing the software. A downside of this solution is that it needs 100% code coverage in testing, which is usually not the case in practice. The second example "Dynamic Continuous Signature Monitoring" [SUG11] is a little bit different. The scheme does not need a learning phase, as it creates the reference values in old executions. That requires that no control-flow violation occurs in the first execution of the desired basic block.
In order to achieve instruction-level granular CFI, a system needs hardware support. We describe two schemes that support instruction-level granular CFI below.
The first method is Sponge-based Control-Flow Protection (SCFP) [Wer+18]. SCFP is a scheme that provides fine-grained CFI by encrypting and authenticating all the instructions of the desired software during compilation. With the help of a hardware extension between the fetch and decode stage of a CPU, it is possible to decrypt and authenticate the instructions during execution. If a fault attack manipulates an instruction, this is recognized by SCFP, as the authenticated decryption will fail. SCFP does not only protect against fault attacks that manipulate instructions or the control-flow, but it also prevents code reuse or code injection.
Another mechanism is called SOFIA [De +17]. This method also uses encryption and decryption during compilation and run time. SOFIA first constructs the according CFG for the software. Then it encrypts the instructions based on the control-flow information in the CFG. During run-time, SOFIA performs the decryption in hardware by considering previously executed instructions and the current program counter. As the decryption of a manipulated instruction fails, SOFIA provides instruction-level granular CFI and prevents code reuse or code injection.

Control-Flow Integrity with FIPAC

In this section, we describe the basics of FIPAC [SNM21] to present the extension for data integrity in the next section. FIPAC is an efficient, purely software-based technique to provide CFI on basic block granularity. It can protect against software-based as well as hardware-based fault attacks. FIPAC uses ARM Pointer Authentication to provide a cryptographically signed control-flow graph. Therefore it is available for ARMv8.6-A devices or newer ones. With a small extension, it is also possible to use FIPAC on ARM architectures starting with ARMv8.3-A.
FIPAC uses an internal control-flow state stored in the general-purpose register x28. In order to identify each basic block, FIPAC assigns a unique constant to every basic block. This constant is the address where the basic block starts. Whenever the program enters a basic block, the internal state gets updated with the help of the `pacia` instruction. For this update, FIPAC uses the unique constant. The `pacia` instruction computes a cryptographically signed Pointer Authentication Code (PAC) and XORs the upper 16 bits into the pointer (in this case, the internal state) that is in the given register. With this technique, FIPAC creates a link between succeeding blocks. FIPAC precalculates the internal state in its post-processing tool. Therefore the program can check if the internal state is equal to the actual state during execution. That ensures that no Control-Flow Violation (CFV) on basic block level can occur, as the execution state would not match the precalculated one if there is a manipulation of the control-flow. In order to check for a CFV, FIPAC supports three different checking policies.

1. At the end of the program
2. At the end of every function
3. At the end of every basic block

When reaching a checkpoint, the state gets checked with the `autiza` instruction. If the state still matches the precalculated one, the check succeeds. Otherwise, a trap is activated, and the program stops execution.
However, this does not fully work regarding indirect or direct jumps. For example, as shown in Figure 2.2 when using a `if-else` construct in the code. There is a merge point of basic blocks 2 and 3 with different states because of the unique constants for each basic block used in state updates. In order to match the different states again, FIPAC introduces `patches`. These `patches` are modifiers for the state so that it is possible to merge them at merge points in the CFG. The same problem occurs with indirect function calls and indirect jumps. FIPAC solves this with the same approach of `patches`.
FIPAC consists of two components, the LLVM compiler extension, and the post-processing tool. The LLVM compiler extension inserts the required instructions for updating the internal state and checking the state with the precomputed one. The value zero gets inserted as a placeholder for patch and check values. The post-processing tool then performs the state precomputation. After calculating the states, the post-processing tool replaces the placeholders with the correct `patches` and states to compare. After this step, the binary is ready to run.

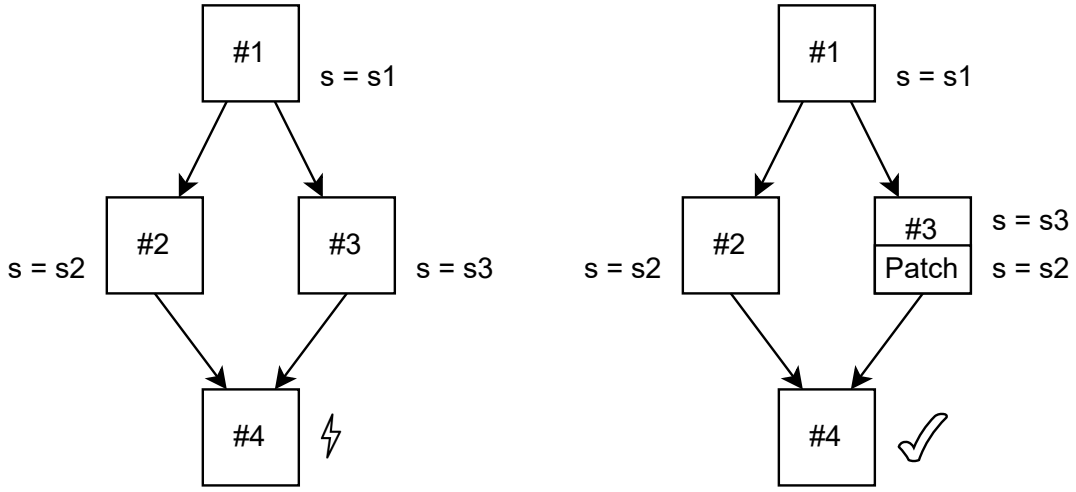


Figure 2.2: FIPAC Patch Values.

3 Design and Implementation

In this section, we first mention the general design of the extension. After the design, we present our implementation on top of FIPAC.

3.1 Design of the CFI-Extension for Data Integrity

This section describes the concept for the data integrity extension on FIPAC. In order to combine CFI and data integrity in one countermeasure, the protected data needs to influence the internal CFI state. Then it is possible to detect a violation on one of the protected properties in one single check instead of needing a separate check for CFI or data integrity. In our design, we focus on known data at compile time. That has the advantage that a tool can precalculate a modification of the internal CFI state with this data, as it knows the desired values of the variables at this time. As FIPAC also uses precalculation for the internal states, we adapt this calculation to consider the known data influences on the state. In order to transform data errors into control-flow errors, we use the XOR instruction. This instruction is easy to consider when calculating the states and does not break the signed control-flow graph of FIPAC. In order to provide a precise selection for the user to decide on the protection of which known data, we introduce a new instruction, called `inject`. This instruction has two parameters, a register and an immediate value. We pass the immediate value to the post-processing tool. The post-processing tool uses the value for the precomputation of the internal state. The value of the passed register in `inject` is XORed with the internal state during runtime. Therefore, the register's value has to match the immediate value passed to the post-processing tool. Otherwise, the next CFI check fails, and the scheme detects the data violation. We describe more details of the implementation in Section 3.2.

3.1.1 Advantages and Disadvantages of using Data Integrity and CFI simultaneously

Combining different countermeasures within one scheme can provide a much lower overhead than introducing two separate countermeasures. That is because the scheme can then combine several checks regarding data or control-flow. Assuming that every check regarding runtime or control-flow is equal, a combined scheme can reach a total speedup of 50 percent for the checks.
A downside of the combination is that the system can not differentiate between data or control-flow violations. However, this is not a significant drawback, as the exact source of violation often does not need to be known. It is by far more important to know if a

```
...
size_t i;
for(i = 0; i < 10; ++i)
{
  AES_round();
}
inject(i, 10);
...
```

Figure 3.1: AES Round Protection with Inject.

violation is present so that the system can perform desired actions. It is not essential to know if the violation has occurred on data or the control-flow, as long as it is detected. In general, the speedup of the combination does not only increase performance. It also makes the combined countermeasure more viable, especially in small devices with limitations regarding space or calculation power.

3.1.2 Usage Example

This section describes a situation where this extension is possible to use. For example, in an AES encryption, a control-flow violation can already break the encryption. One way to achieve this would be to skip the whole encryption. Therefore it is already useful to consider a CFI scheme to ensure that an attacker can not manipulate the desired control-flow. However, in AES, 10, 12 or 14 rounds of the Substitution–Permutation Network (SPN) are used for encryption (depends on the keysize). Even if the control-flow is protected, it is possible to early exit these rounds with a fault attack. That again leads to the fact that an attacker may recover at least a part of the key. That can lead to a complete security breach. However, AES has a fixed round counter, known at compile time. Therefore protecting this counter fits into our presented design. With the help of our extension, the system can check the round counter after the loop finishes. An early exit happened if the round counter does not match the value given to the inject instruction. Therefore the next CFI check detects the violation and can stop execution in order to protect the key of AES. The pseudocode on how to use the `inject` instruction is shown in Figure 3.1.

3.2 Details of Implementation

In this section, we describe the extensions we add to the toolchain to achieve the ability to provide data integrity. FIPAC saves the internal state in the general-purpose register x28 during execution. Therefore, the data that should be protected needs to be XORed to this register. Then the data to be protected affects the state. In order to match the resulting new states to the check values, the post-processing tool needs to know the predefined value for the datapoint. To deliver the value to the post-processing tool, we use a metadata section in the elf file of the application.

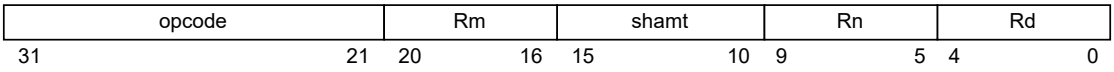


Figure 3.2: Type R Core Instruction.

3.2.1 Overview of the Modifications

In general, we introduce two significant changes. The first one is that we add the new `inject`-instruction to the LLVM compiler. The second change takes place in the post-processing tool. As the `inject` instructions modify the internal FIPAC state, we adapt the post-processing tool in order to calculate the new states correctly. Therefore we save the immediate value from the inject instruction in a metadata section of the elf file by the LLVM compiler. The post-processing tool can then read this metadata section so that it considers the correct value for the precalculation of the new CFI states.

3.2.2 LLVM Compiler Extension

The LLVM Compiler consists of three parts [LA04]. The front-end is the first one, connecting different programming languages with the intermediate representation. In the middle part, there is space for generic optimization methods. The last one is the back-end, which connects the intermediate representation to actual instructions for the target architecture, for example, AArch64.
For our extension, we only use the back-end part. In order to create a new instruction, LLVM uses so-called `tablegen` files. It is possible to declare a new instruction with pseudo-language in these files.
Our introduced `inject` instruction has two parameters, a register and a corresponding 64-bit value. We do not use the value in the binary, as it only contains the check-value for the post-processing tool. The instruction should be a left-shifted XOR of x28 and the given register for the binary representation. ARMv8, in general, supports six different core instruction formats, called `R`, `I`, `D`, `B`, `CB`, or `IW`. The `eor` instruction is in R-format, represented in Figure 3.2 The corresponding execution is $R[Rd] = R[Rn] \oplus R[Rm]$, where $R[Rm]$ is shifted by the amount of bits specified in `shamt`. We show the complete declaration of the `inject` instruction in Figure 3.3.
When using the `inject` instruction in the programming language C as an inline-ASM statement, it gets replaced by the instruction sequence that we show in Figure 3.4. The compiler only inserts the `mov` and `ldr` instructions if the value is not loaded into the desired register from the inject instruction. Therefore, if the program uses the data in the right register, the instruction sequence only consists of a single XOR.
In addition to creating the instruction, we extract the parameter that contains the 64-bit value from the ASM line. We then write the value to the metadata section for the post-processing tool. The post-processing tool later reads and uses the value for the new state calculation. For this purpose, we extend the existing `MetaDataType` of FIPAC by an `uint64_t`. The `MetaDataType` coexists with every machine instruction in the LLVM compiler. Therefore we use this type when required. In order to extract the value, we go

```
// inject instruction that gets converted to eor
def InjectCFI : I<(outs GPR64:$Rd), (ins i64imm:$src),
            "inject", "\t$Rd, $src", "", []>,
            Sched<[WriteISReg, ReadI, ReadISReg]> {

  // parameter 1 has 5 bits
  bits<5> dst;
  // parameter 2 is a 64 bit integer
  bits<64> src;

  // binary representation in machine code
  let Inst{31} = 1;
  let Inst{30-29} = 0b10;
  let Inst{28-24} = 0b01010;
  let Inst{23-22} = 0;
  let Inst{21}    = 0;
  // register given in asm instruction
  let Inst{20-16} = dst;
  // shift left by 48 bits
  let Inst{15-10} = 48;
  // x28
  let Inst{9-5}   = 28;
  // x28
  let Inst{4-0}   = 28;

  let DecoderMethod = "";
}
```

Figure 3.3: TableGen Definition of the Inject Instruction.

through every machine instruction, and if the type is inline ASM, we parse the string and look for `inject`. If we find `inject`, we convert the second parameter to `uint64_t`, which we then save in the `MetaDataType`.
Later in the execution of the compiler, FIPAC creates a list for all the different `MetaDataTypes`. We extend this creation by adding the values for `inject`. FIPAC saves all these entries in a metadata section of the elf file. All these entries start with a two-byte number, representing the type of data that follows. For the `MetaDataType` of `inject`, this type contains the address of the instruction and the known value. We represent these two parameters as a four and eight-byte value. The post-processing tool can later read the values when parsing the elf file. We describe this procedure in the next section.

```
mov x8, xzr
ldr x8, [sp]
eor x28, x28, x8, lsl #48
```

Figure 3.4: Instruction Sequence for Inject.

3.2.3 Post-Processing Tool Extension

FIPAC uses a post-processing tool to insert and calculate the desired patch and check values because it has access to the whole compiled and linked binary. Our solution uses an XOR with the internal state every time a `inject` occurs in the code. Therefore we change the state with every `inject` with the known data. The post-processing tool has to consider this too. Otherwise, the updated state does not match the reference state anymore, and the program would abort execution even if no violation of the control-flow occurred. The LLVM extension writes the injected values to a metadata section. The post-processing tool reads the instruction address and the protected value from this section. The extension performs this within the `ElfMetadataParser::parse` method. The method parses the given metadata section byte by byte and reads the data. The metadata section consists of several concatenated data blocks. In every data block, the first two bytes contain an ID. Based on this ID, FIPAC identifies the data type of the current data block. After the ID, the values for the desired data structure follow. For inject, this is a 4-byte address followed by an 8-byte integer. As the parser does not support reading of 8-byte values by default, we add this functionality. The post-processing tool stores the different values from the inject instructions into an internal vector. The `PACStateCalculator::forwardPass` function then uses this vector. This function calculates the internal state before and after every instruction. Therefore this is the right place to handle the updates from `inject`. If the current instruction address matches one of the addresses in the inject vector, we adapt the state calculation by manipulating the state with the protected value. Because of the modifications, the precalculated states now again match with the states during execution.

4 Evaluation

To give an overview of the impacts of this solution on existing software, we present an evaluation of FIPAC with the data integrity extension in this section. We focus on the two types of overhead in a program, code size and runtime overhead. Code size overhead describes the additional number of bytes introduced by the extension. Runtime overhead does the same for additional execution time. For more general information, we present the overheads in percentage.
We split each overhead section into two different parts. The first part describes the general overhead introduced by FIPAC alone. We mention all three different modes of FIPAC. The difference between these modes is the checking policy. Basic FIPAC only performs one single check of the state at the end of the program. FIPAC fend inserts a check at the end of every function, and FIPAC bb does the same at the end of every basic block. The second part presents the measured overhead for the inject instruction on top of FIPAC. We base the evaluation on the AES encryption from the `Embench` benchmark tests. For the whole evaluation, we use an emulated ARMv8.6 machine. We emulate the machine with the help of qemu on a Linux system with x64 architecture. As the data integrity feature is highly flexible, we measure the insertion of a single `inject` that protects the round counter of the AES benchmark. Additionally, we measure the same AES program when protecting all possible known data.

4.1 Code-Size Overhead

Regarding code size, FIPAC introduces an average overhead of 57.8-100.5%. The minimal overhead is 20%, whereas the maximum overhead is 147% for the `Embench` benchmark tests. The average overhead is 57.8% for FIPAC with one single check, 62.4% for FIPAC with checks at function ends, and 100.5% for FIPAC with checks on every basic block [SNM21].
For the measurement of the code size overhead from the `inject` instruction, we use the Linux command `size`. This command lists the different sizes of the sections for an elf file. The value of the `.text` section represents the code size, so we use this value for evaluation.
As the extension for one inject only introduces the three instructions shown in Figure 3.4, we expect relatively low overhead. We measure the exact overhead of these instructions out of the compiled elf file. The instructions introduce 12 bytes of additional code. In addition to this, we measure the overhead in relation to the compiled binary with FIPAC in it. An insert of one `inject` introduces an additional overhead on top of FIPAC of 0.122%. In the program that protects all known data, this overhead is 0.73%. We also

measure the overheads on top of FIPAC fend and FIPAC bb. We present these values in Table 4.1.

4.2 Run-Time Overhead

Regarding run time, FIPAC introduces an average overhead of 61.4-211.0%. The minimal overhead is 13.6%, whereas the maximum overhead is 663.2% for the `Embench` benchmark tests. The average overhead is 61.5% for FIPAC with one single check, 68.0% for FIPAC with checks at function ends, and 210.0% for FIPAC with checks on every basic block [SNM21].
For the measurement of the code size overhead from the `inject` instruction, we use the Linux command `time`. This command displays the time that a program needs to terminate. In order to get meaningful measurement results, we first change the internal loop counter of the benchmark test. This loop counter represents the number of repetitions for the desired code sequences. In our case this sequence is an AES encryption. We change the loop counters to a value that the AES test compiled with FIPAC needs about 15 seconds of execution time in order to avoid short-term influences like interrupts as good as possible.
As the extension for one inject only introduces the three instructions shown in Figure 3.4, we expect a pretty low overhead. We measure the overhead in relation to the compiled binary with FIPAC in it. Inserting one `inject` introduces an additional run time overhead on top of FIPAC of 0.0752%. In the program that protects all known data, this overhead is 0.2555%. We also measure the overheads on top of FIPAC fend and FIPAC bb. We show these values in Table 4.2.

Program	**Overhead on Top of Fipac** [%]	**Overhead on Top of Fipac Fend** [%]	**Overhead on Top of Fipac BB** [%]
aes-one-inject	0.1217	0.1183	0.1035
aes-all-injects	0.7299	0.7098	0.6208
Average	**0.4358**	**0.4140**	**0.3421**

Table 4.1: Code-Size Overheads of Inject.

Program	**Overhead on Top of Fipac** [%]	**Overhead on Top of Fipac Fend** [%]	**Overhead on Top of Fipac BB** [%]
aes-one-inject	0.0752	0.0750	0.0550
aes-all-injects	0.2555	0.2551	0.1870
Average	**0.1653**	**0.1650**	**0.1210**

Table 4.2: Run-Time Overheads of Inject.

5 Discussion

In this section, we discuss the usage of this extension, how someone can adapt the extension for other fine-grained CFI schemes, and hardware requirements.

Usage of the Extension. The extension can protect every compile time known data. In combination with the provided CFI scheme, the final product provides solid protection with low overhead. Therefore the extension perfectly fits constrained devices like the ones in the IoT domain. For example, the data integrity on known data protects against any early exit of a loop, where the loop count is known. AES encryption does, for example, have such a loop where early exits could break the security, as a key recovery may be possible. With the help of the extension, a developer can prevent this. Furthermore, a developer can also protect several constants in order to ensure no modification by the outside.
However, the extension can not protect unknown compile time data. For example, acquired data by any user input does not fit into the scheme of the extension. That is because then it is impossible to precalculate the internal state.

Adaption for other CFI Schemes. It is possible to adapt the presented exploitation to other CFI schemes. However, there are a few requirements. One requirement is that there has to exist some internal state, which the data can influence during runtime. In addition to this, it must be possible to calculate the influences on the state by the known data. If these two aspects are possible and the manipulation of the internal state does not introduce any cryptographic weaknesses, it is possible to realize the extension.

Hardware Requirements of FIPAC with Data Integrity. The extension does not introduce any further hardware requirements, as is only uses basic instructions like load, move, or XOR to manipulate the state. Therefore it is possible to use FIPAC with data integrity on known data on present or upcoming ARM systems starting with ARMv8.6-A. FIPAC does also present a small extension that allows to use it on ARM architectures starting with ARMv8.3-A [SNM21].

6 Conclusion

As the attack vectors for attacking a system with fault or software attacks rise, all types of devices need countermeasures. Most of the time, one countermeasure is not enough for good protection, so developers need to use more than one countermeasure. That introduces a significant overhead. Especially in IoT devices, this is not affordable because of the limited size and computation power.

In this work, we exploit a fine-grained CFI scheme to provide data integrity for known data at compile time in addition to the CFI. Transforming data violations into control-flow violations requires no separate check for data violations, as our scheme recognizes them as control-flow violations. Because of our flexible design approach, the user can decide on the data he wants to protect. Our solution does not negatively affect the cryptographically signed control-flow scheme that FIPAC provides. The evaluation with an AES implementation where we protect all known data shows a runtime overhead of 0.2552%. The evaluation shows that our final product that serves a two-in-one countermeasure is quite efficient in relation to the provided security aspects.

Acronyms

AES	Advanced Encryption Standard
ASM	Assembly
CET	Control-flow Enforcement Technology
CFG	Control-Flow Graph
CFI	Control-Flow Integrity
CFV	Control-Flow Violation
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
DTV	DirecTV
DVS	Dynamic Voltage Scaling
ECCA	Enhanced Control-Flow Checking using Assertions
IoT	Internet of Things
PAC	Pointer Authentication Code
PCB	Printed Circuit Board
ret2libc	Return To Libc
RISC	Reduced Instruction Set Computer
ROP	Return-Oriented Programming
SCFP	Sponge-based Control-Flow Protection
SGX	Software Guard Extensions
SPN	Substitution–Permutation Network

Bibliography

[Aba+09] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. "Control-flow integrity principles, implementations, and applications". In: *ACM Transactions on Information and System Security (TISSEC)* 13.1 (2009), pp. 1–40.

[Alk+99] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, and J.A. Abraham. "Design and evaluation of system-level checks for on-line control flow error detection". In: *IEEE Transactions on Parallel and Distributed Systems* 10.6 (1999), pp. 627–641. DOI: 10.1109/71.774911.

[Aum+02] Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and J-P Seifert. "Fault attacks on RSA with CRT: Concrete results and practical countermeasures". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2002, pp. 260–275.

[Bal15] J. Balasch. *Introduction to Fault Attacks*. https://www.cosic.esat.kuleuven.be/summer_school_sardinia_2015/slides/Balasch.pdf. 2015.

[Bar+06] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. "The Sorcerer's Apprentice Guide to Fault Attacks". In: *Proceedings of the IEEE* 94.2 (2006), pp. 370–382. DOI: 10.1109/JPROC.2005.862424.

[BB00] Thomas D. Burd and Robert W. Brodersen. "Design Issues for Dynamic Voltage Scaling". In: *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*. ISLPED '00. Rapallo, Italy: Association for Computing Machinery, 2000, pp. 9–14. ISBN: 1581131909. DOI: 10.1145/344166.344181. URL: https://doi.org/10.1145/344166.344181.

[Boh+03] Jrgen Bohn, Vlad Coroama, Marc Langheinrich, Friedemann Mattern, and Michael Rohs. "Disappearing Computers Everywhere - Living in a World of Smart Everyday Objects". In: (Apr. 2003).

[CHW02] Keith D Cooper, Timothy J Harvey, and Todd Waterman. *Building a control-flow graph from scheduled assembly code*. Tech. rep. 2002.

[De +17] Ruan De Clercq, Johannes Götzfried, David Übler, Pieter Maene, and Ingrid Verbauwhede. "SOFIA: software and control flow integrity architecture". In: *Computers & Security* 68 (2017), pp. 16–35.

[El] Saif El Sherei. *Return to libc*.

[FL13] Thomas Flik and Hans Liebig. *Mikroprozessortechnik: CISC, RISC Systemaufbau Assembler und C*. Springer-Verlag, 2013.

[Fre] Free60.org. *Reset Glitch Hack*. http://free60.org/wiki/Reset_Glitch_Hack. [accessed 2021-02-24].

[GC14] Hui Gan and Long Chen. "An Efficient Data Integrity Verification and Fault-Tolerant Scheme". In: *2014 Fourth International Conference on Communication Systems and Network Technologies*. 2014, pp. 1157–1160. DOI: 10.1109/CSNT.2014.235.

[Gol+03] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante. "Soft-error detection using control flow assertions". In: *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*. 2003, pp. 581–588. DOI: 10.1109/DFTVS.2003.1250158.

[KSV13] D. Karaklajić, J. Schmidt, and I. Verbauwhede. "Hardware Designer's Guide to Fault Attacks". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.12 (2013), pp. 2295–2306. DOI: 10.1109/TVLSI.2012.2231707.

[LA04] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[Lim] ARM Limited. *Inside the numbers: 100 billion arm-based chips*. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/inside-the-numbers-100-billion-arm-based-chips-1345571105. [accessed 2022-05-09].

[Lip+20] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. "Nethammer: Inducing Rowhammer Faults through Network Requests". In: *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*. 2020, pp. 710–719. DOI: 10.1109/EuroSPW51379.2020.00102.

[LJK21] Seungkwang Lee, Nam-Su Jho, and Myungchul Kim. "Table Redundancy Method for Protecting Against Fault Attacks". In: *IEEE Access* 9 (2021), pp. 92214–92223.

[Mad+15] Somayya Madakam, Vihar Lake, Vihar Lake, Vihar Lake, et al. "Internet of Things (IoT): A literature review". In: *Journal of Computer and Communications* 3.05 (2015), p. 164.

[Mat+20] Kohei Matsuda, Sho Tada, Makoto Nagata, Yuichi Komano, Yang Li, Takeshi Sugawara, Mitsugu Iwamoto, Kazuo Ohta, Kazuo Sakiyama, and Noriyuki Miura. "An IC-level countermeasure against laser fault injection attack by information leakage sensing based on laser-induced opto-electric bulk current density". In: *Japanese Journal of Applied Physics* 59.SG (Feb. 2020), SGGL02. DOI: 10.7567/1347-4065/ab65d3. URL: https://doi.org/10.7567/1347-4065/ab65d3.

[MK20] Onur Mutlu and Jeremie S. Kim. "RowHammer: A Retrospective". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.8 (2020), pp. 1555–1571. DOI: 10.1109/TCAD.2019.2915318.

[MS91] H. Madeira and J.G. Silva. "On-line signature learning and checking: experimental evaluation". In: *[1991] Proceedings, Advanced Computer Technology, Reliable Systems and Applications*. 1991, pp. 642–646. DOI: 10.1109/CMPEUR.1991.257464.

[MSY06] Tal G Malkin, François-Xavier Standaert, and Moti Yung. "A comparative cost/security analysis of fault attack countermeasures". In: *International Workshop on Fault Diagnosis and Tolerance in Cryptography*. Springer. 2006, pp. 159–172.

[Mun+19] Yeongjin Mun, Hyungseup Kim, Byeoncheol Lee, Kwonsang Han, Jaesung Kim, Ji-Hoon Kim, Byong-Deok Choi, Dong Kyue Kim, and Hyoungho Ko. "Secure integrated circuit with physical attack detection based on reconfigurable top metal shield". In: *Journal of Semiconductor Technology and Science* 19.3 (2019), pp. 260–269.

[Mur+20] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1466–1482. DOI: 10.1109/SP40000.2020.00057.

[OSM02] N. Oh, P.P. Shirvani, and E.J. McCluskey. "Control-flow checking by software signatures". In: *IEEE Transactions on Reliability* 51.1 (2002), pp. 111–122. DOI: 10.1109/24.994926.

[Rei+05] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. "SWIFT: software implemented fault tolerance". In: *International Symposium on Code Generation and Optimization*. 2005, pp. 243–254. DOI: 10.1109/CGO.2005.34.

[Roe+12] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. "Return-Oriented Programming: Systems, Languages, and Applications". In: *ACM Trans. Inf. Syst. Secur.* 15.1 (Mar. 2012). ISSN: 1094-9224. DOI: 10.1145/2133375.2133377. URL: https://doi.org/10.1145/2133375.2133377.

[SGS19] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. "Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity". In: *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '19. Phoenix, AZ, USA: Association for Computing Machinery, 2019. ISBN: 9781450372268. DOI: 10.1145/3337167.3337175. URL: https://doi.org/10.1145/3337167.3337175.

[SNM21] Robert Schilling, Pascal Nasahl, and Stefan Mangard. "FIPAC: Thwarting Fault- and Software-Induced Control-Flow Attacks with ARM Pointer Authentication". In: *CoRR* abs/2104.14993 (2021). arXiv: 2104.14993. URL: https://arxiv.org/abs/2104.14993.

[SUG11] Makoto SUGIHARA. "A Dynamic Continuous Signature Monitoring Technique for Reliable Microprocessors". In: *IEICE Transactions on Electronics* E94.C.4 (2011), pp. 477–486. DOI: 10.1587/transele.E94.C.477.

[SZW08] Saravanan Sinnadurai, Qin Zhao, and Weng fai Wong. *Transparent runtime shadow stack: Protection against malicious return address modifications*. 2008.

[Tat+18] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "Throwhammer: Rowhammer Attacks over the Network and Defenses". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 213–226. ISBN: ISBN 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/tatar.

[Tic+14] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM". In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. San Diego, CA: USENIX Association, 2014, pp. 941–955. ISBN: 9781931971157.

[TM17] N. Timmers and C. Mune. "Escalating Privileges in Linux Using Voltage Fault Injection". In: *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 2017, pp. 1–8. DOI: 10.1109/FDTC.2017.16.

[TSW16] N. Timmers, A. Spruyt, and M. Witteman. "Controlling PC on ARM Using Fault Injection". In: *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 2016, pp. 25–35. DOI: 10.1109/FDTC.2016.18.

[Wer+18] Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. *Sponge-Based Control-Flow Protection for IoT Devices*. 2018. DOI: 10.48550/ARXIV.1802.06691. URL: https://arxiv.org/abs/1802.06691.

[Wit08] Marc Witteman. "Secure application programming in the presence of side channel attacks". In: (Jan. 2008).