Hardware-Assisted Mechanisms to Enforce Control Flow Integrity: A Comprehensive Survey

Sandeep Kumar
School of Information Technology, IIT Delhi
sandeep.kumar@cse.iitd.ac.in

Diksha Moolchandani
School of Information Technology, IIT Delhi
diksha.moolchandani@cse.iitd.ac.in

Smruti R. Sarangi
Computer Science and Engineering, IIT Delhi
srsarangi@cse.iitd.ac.in

Abstract

Today, a vast amount of sensitive data worth millions of dollars is processed in untrusted data centers; hence, the confidentiality and integrity of the code and data are of paramount importance. Given the high incentive of mounting a successful attack, the complexity of attack methods has grown rapidly over the years. The attack methods rely on vulnerabilities present in the system to hijack the control flow of a process and use it to either steal sensitive information or degrade the quality of service.

To thwart these attacks, the complexity of the defense methods has also increased in tandem. Researchers have explored different methods to ensure the secure execution of an application. The defense methods range from software-only to hardware-only to hybrid defense methods.

In this survey, we focus on the relatively new hybrid form of defense methods where software and hardware work in tandem to protect the control flow of applications. We present a novel three-level taxonomy of these defense mechanisms based on first principles and use them to classify existing defense methods. After presenting the taxonomy, we critically analyze the proposed defense methods, study the evolution of the field and outline the challenges for future work.

Keywords control flow integrity, hardware-assisted security, code reuse attacks, control flow bending attacks

1 Introduction

Today, security has become a first-class design criterion for all computer systems. Many such systems, ranging from smartwatches to personal desktops to large data centers store and process sensitive information related to finance, health, public records and even relationships. Consequently, the stakes are much higher, and any security flaws in these systems/applications can lead to losses amounting to millions of dollars.

The complexity of attacks has gone up in the past few years mainly due to the emergence of sophisticated techniques and stronger financial incentives. These attacks target different aspects of a process's life cycle and steal sensitive information or disrupt service. The defense methods have managed to keep pace with these attacks; most attacks are patched within a few weeks of their detection.

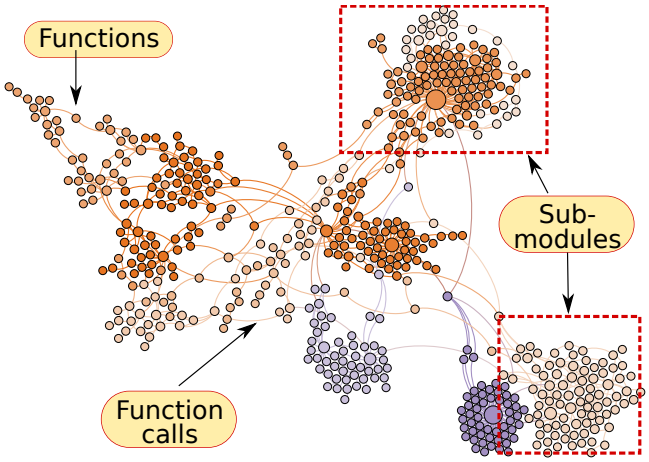


Figure 1. A control flow graph for OpenSSL. The vertices are functions in the application and the edges between two vertices represents a direct function call.

Along with these periodic patches, software vendors have also taken more proactive steps. Researchers in industry and academia have developed a plethora of defense methods anticipating sophisticated attacks. Our analysis shows that even though the final goal is the same – maintain control flow integrity – there is a wide variance in the design of defense methods. Hence, it needs to be an active area of study because the quest to minimize performance and energy overheads without compromising on security will always remain.

Before discussing the defense methods, we need to formally define what it means for an application to be *secure* in the context of this paper. We shall only focus on the *control flow integrity* or CFI. The term CFI was initially introduced by Abadi et al. [7]. It means that the execution of a binary should follow a path that is pre-determined by its input parameters. Given that the execution path consisting of dynamic instructions is typically very long and has a lot of loops, it is typically represented as a control flow graph or CFG where the nodes are functions, and the edges between the nodes are the function calls (see Figure 1 for an example). The CFG of an application is typically generated at runtime by collecting execution traces. We can alternatively define a static variant of the CFG that is independent of runtime

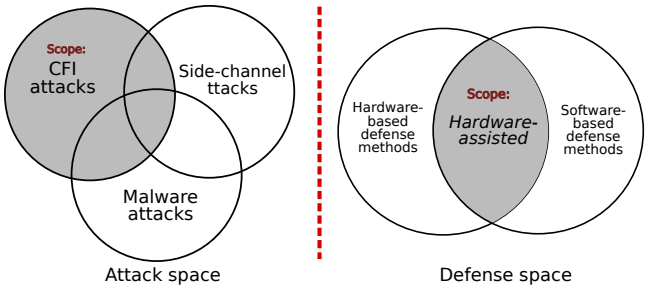


Figure 2. Scope of this survey: We focus on hardware-assisted defense mechanisms for protecting the CFI of a program

parameters and simply captures the function calls embedded in a binary (can be created using static analysis). Abadi et al. [7] argue that tampering with the execution flow of an application is the essential step in most attacks. Hence, ensuring the CFI of an application is of paramount importance. CFI has been used widely in prior work [32], and we also use it as the basis for selecting defense methods for our analyses.

Due to the sheer number of defense methods, classifying them is a non-trivial task. Collectively, these methods work on every aspect of a process's life cycle and leverage all possible hardware support available to them.

Hence, we faced the need to come up with a novel taxonomy that is based on the impact of the defense methods (more details in Section 4). Doing so provides us with mutually exclusive sets of defense methods that capture the high-level structure of the entire research field.

1.1 Scope of the survey

The attack and defense methods are constantly in an arm's race. Their degree of sophistication keeps increasing over time. The most basic approaches in this space are software-based methods. However, such a method can be bypassed if the attacker has physical access to the machine [17, 57] or can leverage other OS or HW-level vulnerabilities. Another negative aspect of such methods is their high performance overheads mainly due to the constant monitoring and additional security checks. Hence, hardware-based security methods have gained traction in the past few years. A hardware-based method has a lower performance overhead, provides a reliable root of trust and makes it much harder for SW-based attacks to be successful. However, a hardware feature takes years to be production-ready, as it has to go through many rounds of design and validation.

Therefore, a hybrid approach combining the efficiency and security of existing hardware mechanisms with the flexibility of software is preferred. Hardware exposes generic features that can be leveraged in very intelligent ways. Hardware modules often collect process telemetry data, and software modules process it to detect anomalous behavior. In this survey, **w**e shall focus on such hardware-assisted defense methods (refer to Figure 2). It is a rapidly growing field and is expected to become more prominent in the near future.

A hardware-assisted or *hybrid* method provides the flexibility of software-based methods and the efficiency and security of hardware-based methods. Moreover, we shall focus on violations of the CFI property where attacks operate by hijacking the control flow with a malicious intent. Note that pure hardware or software-based approaches will not be discussed this survey paper. A few disclaimers are due.

Due to a lack of space, we shall not delve into the details of how particular hardware features are implemented, the challenges faced by the hardware developers, or how they affect the pipeline of a processor. For a detailed analysis of the inner workings of hardware features, we refer the reader to the work done by de Clercq and Verbauwhede [32] and Coppolino et al. [24]. We instead focus on the challenges a security researcher or developer faces such as selecting the right hardware for her needs, configuring it, designing a solution around it, and maneuvering through the challenges or limitations introduced by the hardware.

There is some related work in this area. The work done by de Clercq and Verbauwhede [32] and Coppolino et al. [24] elaborated on many such hardware features and discussed their design and operation in detail. However, both the references do not discuss specific defense methods and where these features are used. Furthermore, they also leave out certain technologies such as hardware-provided trusted execution environments, or TEEs, which have been extensively used in providing security to applications in cloud environments. Other works in this area suffer from similar limitations such as being restricted to a specific class of attack methods, a specific field ([45]), or a specific class of defense methods ([51, 64]). Our survey paper distinguishes itself on the basis of its comprehensive coverage and the connections that it makes between hardware features and software-based defenses that leverage them. The latter aspect is the *key novelty* of this paper.

1.2 Organization of the paper

The rest of the paper is organized as follows. We discuss the relevant background in Section 2. This is followed by a discussion on the related work in this area in Section 3. After this, we present our novel taxonomy of the hardware-assisted defense methods in Section 4, followed by a detailed discussion in sections 5, 6, and 7. We provide remarks on the general trend of the work done by the defense community and point out the gaps that need to be filled by the upcoming work in Section 8. Finally, we conclude in Section 9.

```
int a[] = new int[100];
//Error: IndexOutOfBoundsException
a[200] = 10;
```

Listing 1. Example of a safe programming language (Java)

```
int a[100];
//This is allowed. No error
a[200] = 10;
```

Listing 2. Example of an unsafe programming language (C++)

```
void foo(){
    char buff[4]; // Stack variable
    printf("Input: ");
    gets(buff); // Storing on the stack
}

int main(){
    foo();
    return 0;
}
```

Listing 3. Sample Code to demonstrate buffer overflow attack

2 Background

In this section, we shall discuss the relevant background required for the rest of the paper.

2.1 Safe and unsafe languages

Today, a developer has many choices for the programming language that needs to be used. Examples include C, C++, Java, Python, Rust, Scala, Kotlin, etc. The decision to select a programming language is based on the ease of programming and runtime efficiency. For example, a developer writing an "app" for a mobile device will probably use Java or Kotlin (in Android). Using C++ for such a task will be unnecessarily cumbersome. However, any application that is performance-sensitive will need to be written in C or C++.

Sadly, this performance improvement comes at a cost. One of them is no explicit memory address checking before accessing a memory location. As shown in Listing 1, an array access beyond its bounds will result in an error in Java (a safe language). However, doing so in C++ is allowed – it will lead to an erroneous result, nevertheless it is permissible (see Listing 2).

The first step for any attack on the CFI of an application is to corrupt the memory state of that process. In the absence of explicit memory validations, an attacker can overwrite a process's stack (explained later in Figure 4) or other control data structures and hijack the application's control flow. This attack is known as a buffer-overflow attack and is the basis for other more sophisticated attacks (details in Section 2.2). In safe languages, such kinds of memory corruption attacks are either not possible or difficult to mount. Sadly, there is a concomitant performance cost.

2.2 Attacks on control flow integrity

In this section, we present a brief overview of the attacks on the CFI of a binary. Note that this is not a comprehensive list. We shall discuss the basic ideas on which these attacks are based (see Figure 3). The exact implementations may vary.

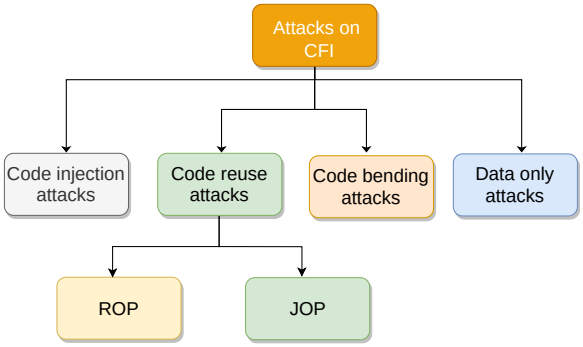
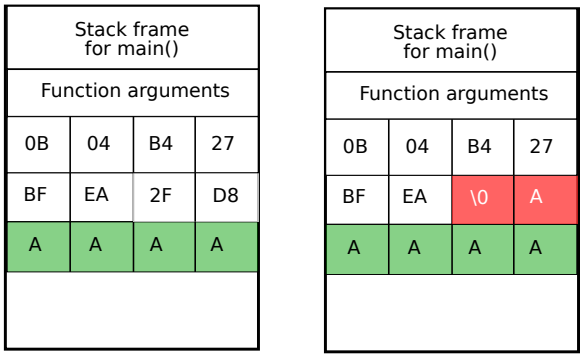


Figure 3. Attacks on the control flow integrity or CFI of an application



(a) Input fits in the buffer **(b)** Input does not fit in the buffer

Figure 4. Example of a buffer overflow attack on the code shown in Listing 3

2.2.1 Code injection attacks

Code injection (CI) attacks exploit the lack of memory checks to mount an attack. First, a CI attack corrupts the stack of a process by overwriting parts of it with malicious data. For example, Listing 3 shows a process that asks the user for an input and stores it on the stack. Figure 4a shows the

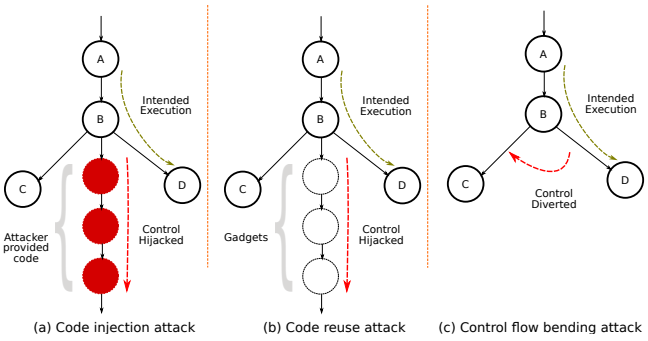


Figure 5. Code injection, code reuse and control flow bending attacks

application stack when a valid input (fits in the buffer) is provided; now compare it with Figure 4b when an invalid (size more than the buffer) input is given. In the latter case, the stack of the process can get overwritten with the extra data provided by the user. This generally leads to a process crash since overwriting the stack may result in *corrupting* the return address (stored on the stack). It is possible to meticulously craft the input such that the return address points to a location within the stack.

Now, the attacker can insert her own code in the stack (again via specially crafted inputs). Once the function returns, the control will jump to the return address. The code injected by the user will get executed. It can be used to leak sensitive data, change the application's control flow, or launch more attacks. These new instructions add new nodes (functions or basic-blocks) in the control flow of the application (see Figure 5a). Nowadays, these attacks are prevented by ensuring that a data page is either writable or executable, but never both. This protection mechanism is known as *data execution prevention* or DEP (also referred to as the $W \oplus X$ property) and is part of all modern operating systems.

2.2.2 Code reuse attacks

The protection provided by DEP ensures that an attacker cannot execute any malicious code by overwriting the stack of an application. However, it does not prevent overwriting the stack per se using the buffer overflow technique. An attacker can use the same method to divert the control flow of the application to the *attacker-intended* locations rather than the *developer-intended* locations. The basis of this attack is that an attacker has access to the binary or can probe the address space of the application to find code fragments that end in an indirect branch statement such as *ret*. Such code fragments are called *gadgets* and multiple gadgets can be chained together to execute a piece of malicious logic. This attack is known as the code reuse attack or CRA because it relies on code snippets that are already a part of the binary. Figure 5(b) shows a CRA attack where a chain of gadgets is executed.

A CRA attack can be mounted on the forward-edges (conditional branch statements) of a control flow graph or on the backward-edges (return statements). The former is known as a *jump oriented programming* attack or JOP attack, and the latter is called a *return oriented programming attack* or ROP attack. As of today, these are one of the most powerful set of attacks that can be mounted on the CFI of a binary. The sheer number of branch statements executed by an application makes it difficult to prevent a CRA attack without incurring unacceptable performance overheads. Furthermore, a CRA attack is hard to detect as it relies on the code fragments already present in a process's address space instead of requiring the insertion of new code. The standard approach here is to analyze the execution and check for an anomalous sequence of executed code (in the form of gadgets).

2.2.3 Control flow bending attacks

A CRA attack is powerful and hard to detect. However, it augments the control flow of an application. The malicious code executed by the attack shows up as entirely different pathways in the control flow of an application. Apart from this, the number of branch instructions executed (`ret`,`jne`) increases compared to the default execution. These properties are used by defense methods to detect these attacks by using "signature-based" schemes.

A *control flow bending attack* or CFB attack uses a single branch flip to break the CFI property of an application (see Figure 5c). Compared to a CRA attack, a CFB attack does not augment the original control flow graph – it does not add any new branches to the control flow, it just reuses the existing branches. It uses conditional branch instructions like `jne` for this purpose. A conditional branch instruction decides the path taken based on whether a condition is met or not. For example, in a license check module, the application may choose to execute or quit depending on whether a valid license is provided or not, respectively. This decision boils down to a single conditional branch. A CFB attack hijacks the control flow of a process by forcing it to take a fixed path in the control flow graph, irrespective of whether the branch condition is met or not [17, 57]. Since a CFB attack does not add new paths to the control flow graph, detecting it is very challenging [41].

Both, CRA and CFB attacks are Turing complete [17], that is, an attacker can use either of them to execute any arbitrary logic.

2.2.4 Data oriented programming attacks

Apart from the previously explained attack methods, a new class of attacks indirectly violates the control flow integrity of an application. This class of attacks is known as *data oriented programming*, or DOP attacks.

The address space of an application can be logically divided into two planes: the control and the data plane [50]. The control plane consists of pointers, instructions, etc., that

control the execution flow of the application. On the other hand, the data plane consists of data variables that are not a part of the control decisions in the application. The previously described attacks were based on the control plane, where an attacker attempts to hijack the control flow by influencing the process of jumping to a new location. A DOP attack modifies the variables that decide the control flow of an application, such as a variable being checked in an *if* statement.

Hu et al. [50] show that a Turing-complete attack can also be mounted using just the variables present in the data plane. These attacks are also known as non-control attacks [50]. We include them in the scope of the paper as an attacker can use DOP attacks to violate the CFI property of an application.

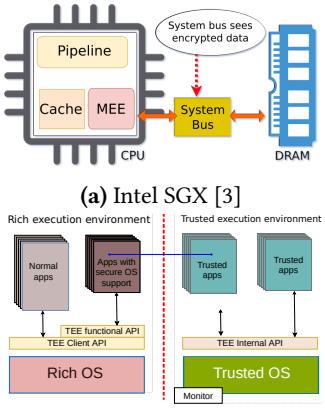
Apart from these classes of attacks, there is a new category of attacks where instead of directly attacking the binary, the attacker observes the effect of the execution of a binary on the system state, such as the branch predictors, shared caches, power usage, page faults, etc. This information is then used to leak sensitive information by the attacker. Here, an attacker violates the security guarantees by observing the state of these system components. The attacker can infer secrets within the protected region without root-level access. Over the past few years, this line of thinking has matured significantly, and many different variants have surfaced including but not limited to the prime+probe attack, flush+reload attack, and evict+time attack [55, 59, 72, 77, 105].

2.3 Hardware advancements

As explained before (see Section 1.1), we only focus on hardware-assisted defense mechanisms. Hence, in this section, we provide a brief overview of different hardware features that a defense method can use. This is not an exhaustive list, and we only focus on the hardware features prominently used for CFI protection. These features are typically generic in nature that different methods can use for different purposes. For example, the hardware PMU (*performance monitoring unit*) counters can be used while a binary is executing to either monitor it or collect it and validate it later.

2.3.1 Trusted execution environment or TEE

Modern CPUs have a provision for executing the instructions of a binary securely. They are even protected from privileged software such as the operating system and hypervisor. This secure environment is called a *trusted execution environment* or TEE. A TEE defines a secure region within the chip and guarantees *integrity, confidentiality, and freshness* of the data and code in it. The confidentiality property implies that the protected content will not be accessible to an untrusted malicious entity; the integrity property says that a malicious entity cannot change the protected content without being detected; the freshness property ensures that an attacker



(a) Intel SGX [3]

(b) ARM TrustZone [6].

Figure 6. Two trusted execution environments or TEE solutions from Intel and ARM, respectively.

cannot replay an older version of data without detection. A TEE also allows a user to verify that her application is executing without any tampering on a remote, untrusted machine. Here, we briefly discuss two of the most popular TEE solutions: Intel SGX [3] and ARM TrustZone [6] (see Figure 6).

Intel SGX: Intel Software Guard eXtension or SGX [26] is a TEE solution from Intel. It is a collection of a set of new instructions and hardware mechanisms that allow the application to execute securely inside a sandboxed region called an *enclave*. Intel SGX at boot time reserves a part of the memory called *processor reserved memory* (PRM) for its secure operations. As of now, the size of the PRM is mostly limited to 128 MB (256 MB in a few systems) and is managed by a hardware component called the *memory encryption engine* or MEE. The MEE transparently ensures the confidentiality, integrity, and freshness of the data stored in the PRM. However, this security comes at a performance cost. Since the operating system is not a part of the trusted code base, applications executing within SGX are not allowed to issue direct system calls. Furthermore, due to security reasons, applications within SGX are not allowed to share memory within the PRM. They can share memory in the untrusted region of the main memory. Costan and Devadas [27] present a detailed analysis of the working of Intel SGX in their paper [27].

ARM TrustZone: TrustZone is a TEE solution from ARM. Its implementation differs significantly from Intel SGX in spite of having similar features. TrustZone logically divides the CPU into two worlds: *secure* and *unsecure worlds*. The secure world comprises a secure OS, secure firmware, and secure I/O (which can only be accessed by the code running in the secure world). The isolation guarantee that separates the secure and non-secure worlds is ensured by the hardware.

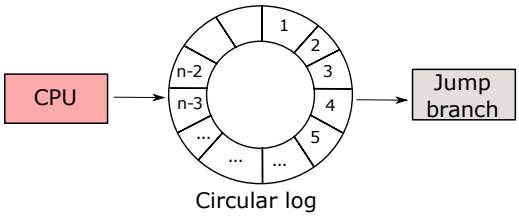


Figure 7. Last branch record or LBR

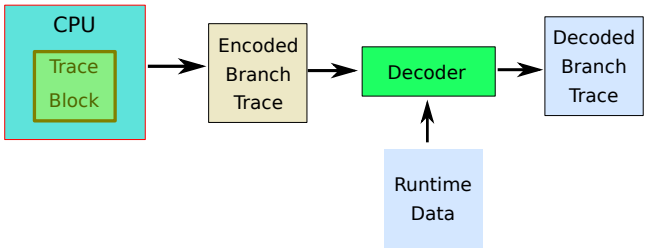


Figure 8. Collecting the branch trace on Intel processors. Adapted from [65]

The memory buses are also augmented to support TrustZone features.

2.3.2 Hardware-assisted instrumentation

Dynamic instrumentation of a process is a well-known technique to debug or optimize performance. Here, an application is profiled during execution to gain an insight into its working. The instrumentation can be done at the level of functions, basic blocks, or instructions. The instrumentation process can also be used to prevent an application from a CFI attack by validating the forward (indirect jumps) and the backward branches (return instructions). However, software-based instrumentation incurs unacceptably high performance overheads [94] due to the sheer number of events generated.

To remedy this situation, modern hardware provides different ways to instrument a process effectively.

The control flow of an application is determined by the branches taken by the application while executing. The information about the branches taken can be used to debug, optimize, and also validate the execution of the application. Modern Intel processors have a feature called *last branch records* or LBRs [56] that allow the logging of the branches taken by an application in special registers. As shown in Figure 7, the number of registers to log the branch information is limited and is organized as a ring buffer. This allows the system to analyze a branch in a "context" by observing the branches taken before or after it.

The LBR only captures the outcomes of the branch instructions taken by an application. However, sometimes we might need more information. *Intel processor trace* or IPT (see Figure 8) enables efficiently capturing the trace of a process. In order to reduce the overhead, IPT stores the trace in an encoded format and skips the information that can be generated from other recorded data. The encoded data can be decoded offline and processed as a regular trace. IPT incurs a 15% less performance overhead than the corresponding software-based methods [36, 42, 52, 65]. Specifically, IPT records branch information, system information, details of interrupts received, transaction entry/exits, and VM entry/exits.

2.3.3 Pointer integrity

All the attacks on the CFI of an application rely on "tricking" the system to use an attacker-crafted pointer instead of the original pointer. In an ROP attack, the attacker modifies the pointer containing the return address value. In a JOP attack, the attacker modifies the pointer containing the target address of an indirect branch. ARM introduced *pointer authentication* that ensures that only valid addresses can be used. Any attempt to use a modified pointer will result in an application crash. To achieve this, an encrypted signature is associated with every pointer. This signature is verified before using the pointer. The signature is generated using a key that is not accessible from user space, the pointer itself, and the current value of the stack pointer. As the attacker will not have access to the key, she cannot generate a valid signature. The signature is calculated using an instruction called `PUC` and is stored in the unused bits of the 64-bit pointer. The validation is done using an instruction called `AUT` [25, 93].

2.3.4 Memory accessibility

Apart from effective ways to instrument binaries, innovations in the memory system make it hard for attackers to leak information. Industry and academia have also come up with complete memory encryption solutions. The encryption and decryption operations are carried out in hardware to make them more efficient. Intel SGX [26] offers, for example, offers partial encryption of the physical memory space. In this case, a part of the main memory is encrypted and is used for an application running within an SGX enclave. The reserved part of the memory is called the PRM, and the usable part of it is referred to as the *enclave page cache* or EPC (see Figure 9). AMD recently launched *secure memory encryption* or the SME technology [54] that offers complete, hardware controlled, memory protection (primarily relies on encryption of data stored in memory).

Apart from encryption, Intel memory protection keys or MPK provides a mechanism to divide and separately manage the address space. Traditionally, bits in the page table entries are used to control the properties of the page (writable, executable, etc.). In Intel MPK, 4 unused bits in the page table entries are used to assign one of sixteen keys to each page. Apart from this, the processor contains a 2-bit register for each of the sixteen keys. A value of 0 in the register

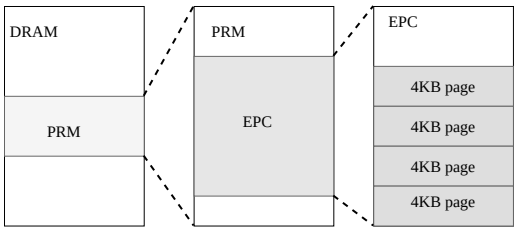


Figure 9. Overview of Intel SGX.

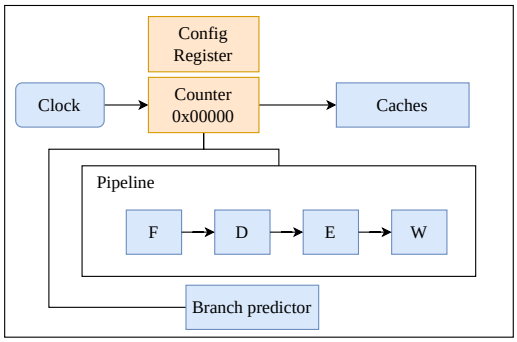


Figure 10. Use of dedicated registers in a traditional pipeline [12].

of a particular key will disable access to all of the pages that are assigned that key. This is used to divide the address space into different regions (up to sixteen) and manage them independently.

2.4 Hardware-accelerated instructions

Traditionally, a microprocessor was designed to optimize integer and floating-point operations. However, with the ever-evolving nature of the field, the workloads are becoming increasingly complex, and hence, the microprocessors are required to perform complex tasks. For example, common cryptographic operations such as encryption, decryption, and hashing are executed very frequently in these systems. A common approach to performing these operations is to break down these complex tasks into their basic mathematical operations. However, this causes a significant wastage of CPU cycles and a resulting slowdown in performance. Hence, there is a case to be made for hardware-accelerated cryptographic operations, where hardware developers provide dedicated instructions for standard cryptographic operations [35]. To this end, Intel has released dedicated instructions for encryption [1] and hashing [82].

Advantages: These instructions provided a performance improvement of up to 4× for encryption [47] and 4.6× for hash-based operations.

2.5 Hardware performance counters

Modern hardware has a mechanism to get access to low-level micro-architectural events such as the number of TLB hits and misses, L1 hits and misses, total cycles, total stall cycles, etc., in an application's execution [12]. This is achieved by additional programmable and dedicated registers that store the number of occurrences of specific events (see Figure 10). Though there are a fixed number of hardware registers, there is no limit on the number of events that can be sampled due to the multiplexing feature of the performance counter hardware [78]. While multiplexing, each event is sampled in a round-robin manner. This reduces the precision of counting, but increases the number of events that can be sampled at the same time. These registers are populated by the hardware during the execution of an application, incurring a minimal overhead while doing so. The values of these counters provide insights into the characteristics of a particular application's execution and can be used to debug, optimize, or sometimes even ensure a valid execution.

Advantages: In the absence of hardware performance counters, the burden of tracking events falls on the operating system or the hypervisor. There are two major drawbacks of doing so: ❶ Any software-level entity will not have access to the fine-grained micro-architectural events such as L1-hits and misses. ❷ Even if we somehow enable access to capture those events, the executing process has to be continuously interrupted; we need to transfer the context to the OS, capture the relevant metric, and return control to the process. This context switching will incur a huge performance overhead, as seen in software-only solutions. Hardware performance counters eliminate all these issues and enable seamless tracking of performance metrics of a process.

2.6 RISC-V

RISC-V is an open-source ISA that has gained popularity as it does not require a license to use it. Due to its open-source nature, it enables security researchers to do a more detailed analysis of it. However, at the same time, it also allows attackers to find vulnerabilities in it easily. Hence, developers are working towards providing a root-of-trust (a trusted entity) for RISC-V to prevent attacks on it. In this paper (see Section 6.5), we take a brief look at these techniques. For a more detailed analysis, readers should look at the work by Lu [67]. In Section 6.5, we discuss more proposals [11, 71, 101].

The hardware space is seeing a rapid development of features that address various challenges such as performance, ease of development, and security challenges that modern applications face (see Table 1). The defense community is increasingly adopting them to provide more efficient protection.

Table 1. Table showing a brief description of hardware features along with its use cases, overheads, and implementation details.

Feature	Purpose	Overheads	Implementation
TEE Trusted Execution Environment [3, 6]	Secure containers in clouds	Encryption and integrity checks. OS interactions	There is a dedicated memory encryption engine, which encrypts all the traffic between the CPU and memory. For ensuring data freshness, they use a Merkle tree, where the root of the tree is stored in the TCB.
HW Counters [102]	In-depth information about execution	Negligible	These are regular registers that store different performance-related data such as L1 misses, TLB misses, etc.
Memory Encryption [3, 54]	Encrypted memory (TME, AMD Epyc)	A few cycles for every memory access	Just encrypts the memory and does not perform any integrity checks.
HW-accelerated Instructions [89]	AES, SHA	Minimal	These are additional instructions in the ISA. They significantly accelerate the execution of encryption and hashing operations; the latency is at the most 10-20 cycles.
Last Branch Record [56]	Control flow generation	An additional circular buffer.	A simple circular queue that stores the details (PC, outcomes) of the last k branches.
Intel Processor Trace or IPT [52]	Trace	Encoding and Decoding	Extends the previous idea to have a larger, circular queue that stores branch, interrupt, and VM entry/exit information.
Pointer integrity [74]	Pointer authentication	Encryption tag with every pointer. Software changes.	A base and bound attached with every pointer (optionally a signature). This is checked by hardware periodically or upon the return of a function to validate the CFI.
Random Number Generation/ PUF [5]	Generate unique encryption keys	Small hardware units	TRNG generators based on physical phenomena or pseudo-random number generators. The random numbers are used to generate unique, per-session encryption keys for authentication.

Table 2. Related survey papers

Name	Year	Description
A survey of Hardware-based Control Flow Integrity. de Clercq and Verbauwhede [32]	2017	Pure hardware defense methods
A survey on security threats and defensive techniques of machine learning: A data driven view. Liu et al. [64]	2018	Defense methods with machine learning.
A comprehensive survey of hardware-assisted security: From the edge to the cloud. Coppolino et al. [24]	2019	List of hardware-assisted features
A Survey on IoT Security: Application Areas, Security Threats, and Solution Architectures. Hassija et al. [45]	2019	IoT related defense methods
Survey of Attack Projection, Prediction, and Forecasting in Cyber Security. Husak et al. [51]	2019	Cyber security forecasting
A Survey of Exploitation Techniques and Defenses for Program Data Attacks. Wang et al. [100]	2020a	Data attacks
Hardware-assisted CFI defense mechanisms: A Comprehensive Survey (this work)	2022	Hardware-assisted defense mechanisms.

3 Related work

As already mentioned, many attack and defense methods have been formulated in the past decade. Many survey papers have aimed to characterize and classify them. Table 2 shows the scope of other surveys in this field and compares them with this survey.

The work done by de Clercq and Verbauwhede [32] is the closest to our work. They focus on hardware-based defense methods to prevent attacks on the CFI of an application. They specifically focus on the prevention of code-reuse-attacks and discuss the work done by the hardware community in this direction. Examples include a hardware-based shadow stack, code-pointer integrity checks, and branch validations. However, they leave out a few critical hardware features such as secure execution mechanisms, specifically TEE (Intel SGX [3] or ARM TrustZone [6]), hardware tags [81], and heterogeneous ISAs [96]. A TEE explicitly violates one of the assumptions made by the authors in the paper for attackers' capabilities. The authors assume that an attacker can read the memory contents. However, in a TEE setting such as Intel SGX, a part of the memory (secure memory) is encrypted by the hardware. Although an untrusted entity can read the memory (by forcing a dump of either secure or unsecure memory), she cannot decrypt the data as it is encrypted by a key stored within the hardware. Intel SGX provides data confidentiality, integrity, and freshness guarantees for the data stored in the secure region of the memory – essentially thwarting attacks that rely on data being present in plaintext in the main memory.

Coppolino et al. [24] elaborately discussed the different hardware-assisted mechanisms available that developers can

Table 3. Differences between the work done by Coppolino et al. [24] and this work.

Features	Edge to Cloud [24]	*This Work*
Description of HW features	Yes	Yes
Use cases	No	Yes
Defense model using HW features	Unclear	CFI
Discussion of defense methods	No	Yes

use to improve the performance of existing defense methods or develop new defense methods. However, they do not discuss any work that potentially uses these features. Incorporating these hardware defense methods into a complex software ecosystem is non-trivial. In fact, the main aim of this survey is to elucidate the fact that most of the innovation lies in effectively using the features that are provided by hardware to ensure that the CFI can be verified efficiently (with a minimal degradation in performance). Merely listing the HW features is not enough. Showing how to use them is important and in our humble view, merits a full survey.

For example, Intel SGX provides a secure sandbox for processes to execute in isolation (without any fear of eavesdropping or tampering). However, it also imposes restrictions on the application due to security considerations [2, 44, 53]. It is also associated with large performance overheads [57, 63]. Hence, developers devise different methods to mitigate these overheads. A detailed discussion of these methods is important to completely understand the applicability of the hardware feature, which is missing from this work. On the other hand, our work covers all the popular hardware-assisted mechanisms (ignoring a few, such as Intel TDT, as there is insufficient public domain work). We focus on the applicability of hardware-assisted features and briefly describe their working in Section 2.3. The work by Coppolino et al. [24] can supplement our work for readers interested in more details. Table 3 lists the key differences between Coppolino et al. [24] and this work.

Wang et al. [100] categorize defense methods that aim to prevent data oriented programming or DOP attacks. They use three categories in their classification: the first category contains those methods that aim to fix a memory vulnerability, the second category captures those methods that aim to randomize the process layout, and the third category captures those defense methods that aim to dynamically prevent attacks by actively monitoring the execution of a process. However, they only focus on DOP attacks, leaving out other forms of attacks such as code reuse attacks and control flow bending attacks. Hassija et al. [45] capture defense methods in the field of IoT. They specifically focus on techniques that can be executed in a resource constrained environment and are suitable for IoT devices.

Liu et al. [64] discuss ML-inspired defense methods. Machine learning (ML) and deep learning (DL) algorithms have seen wide adoption in different fields in the past few years. Following this trend, many attempts have been made to devise "intelligent" defense methods powered by ML and DL algorithms. The authors focus on the attacks that target the training phase, inference phase, or data security in general.

Husak et al. [51] focus on the attacks and defense methods in cyber-physical (CP) systems with an emphasis on the methods that aim to forecast or predict an attack. They classify these methods as discrete, continuous, machine learning based, or based on pattern matching models.

4 Taxonomy

In this section, we present a novel taxonomy for hardware-assisted defense methods.

4.1 Challenges

There are a plethora of defense methods that aim to protect the CFI of an application by leveraging one or more hardware features.

One approach for classifying them can be based on which stage of the life cycle is targeted. There are different strategies for protecting the source code, binary, executing process, and the results generated by a process. Most of these defense methods provide online protection to an executing application by either monitoring the state of a memory region (e.g., stack) or by monitoring its impact on the system. Hence, a classification strategy using this approach will place most of the defense methods in the *executing application* category, making it difficult to gain insights. Furthermore, there are very few defense methods that focus on protecting or asserting the validity of post-execution results. This is because the defense methods in this category use an optimistic approach by allowing the execution to go through and then process the logs to detect attacks. This approach is not suitable for many real-time and interactive applications.

Another approach can be based on which hardware feature is used. However, this also does not lead to an elegant classification of the defense methods. As already mentioned, a hardware feature takes a long time to become available in commercial CPUs. It has to go through many years of development and testing. Hence, software developers try to leverage existing features to the hilt. As a result, a few features get disproportionately used and thus the treatment does not remain balanced.

4.2 Classification

To efficiently classify the defense methods, we propose to classify them on the basis – which system is being affected/-targeted. To this end, we propose a three-level of taxonomy: *binary-based*, *process-based*, and *system-based* classification (see Figure 11). These categories capture the impact of the defense methods on the binary, process, and system in terms of the modifications or observable effects, respectively. This classification approach presents a balanced and intuitive

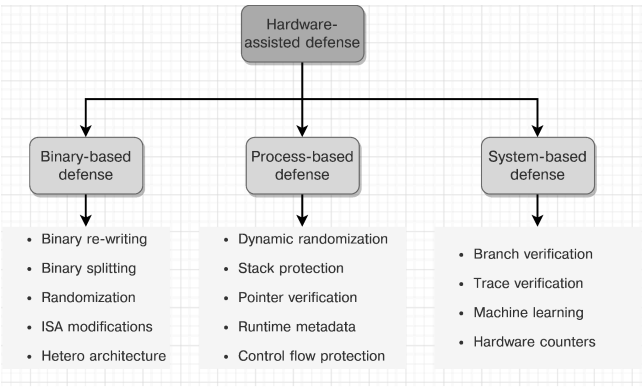


Figure 11. A novel three-level taxonomy

way of categorization. For example, a developer that has no control over how a binary will be executed in its target environment must look at the defense solutions in the *binary-based*. A security developer proposing generic solutions for applications has no control over binary creation and the environment where it will be executed must look at the *process-based* defense methods. A cloud provider has no control over binary creation or the execution process. However, she has full control of the environment where the application will execute. In such scenarios, the *system-based* defense methods make the most sense.

4.3 A *binary-based* classification

The defense methods in the this bucket [30, 34, 40, 63, 73, 83, 87, 99] work from the compilation of the application to its loading in the main memory for execution. The key idea here is to empower the binary to thwart off attacks during execution or raise the bar for the attacker to mount a successful attack. The set of defense methods consists of methods that rewrite the binary (addition or removal of certain instructions), split the binary into secure and unsecure regions and focus on the protection of the former and randomize the location of the binary's memory regions within the virtual address space.

Certain defense methods in this class work directly on the source code and propose changes to the compiler for including additional checks in the generated binary. Many approaches operate on the complete control flow graph and ensure that it remains valid during an execution.

4.4 A *process-based* classification

Although the methods in the *binary-based* category have access to the complete source code and other compiler-based tools, they are limited in terms of what can be done to secure a binary. The modifications to a binary have to take into account its size, compatibility, and performance overheads. Hence, the defense methods in the *process-based* level of classification [16, 18, 22, 28, 37, 49, 61, 62, 69, 80, 81, 85, 88, 106, 108] are based on what specific property of a binary's execution – stack, kernel data structures, memory pointers, and instructions – are protected.

Table 4. The pre-execution class of defense methods.

Defense method	Hardware feature	Application attribute	Perf.	Attack model	Method
Hafix [30]	CFI Inst. and CFI Memory	Backward-edge CFI	2%	ROP	Verification
OpaqueCFI [73]	Intel MPX	Inst.	4.7%	ROP	Randomization
Glamdring [63]	Intel SGX	Inst.	20% - 30%	CFI	Binary splitting
RISC-V [34]	HW key generation	Inst.	0.7%	ROP	Randomization & Encoding
SGXShield [83]	Intel SGX	Inst.	7.61%	ROP	Static Randomization, multi-stage Loader
PolyGlot [87]	ISR	Inst.	4.6%	DRA	Encrypted binaries
CFITrimming [40]	SHA instructions	Trace & Instructions	1.87%	CRA	Binary stripping
HetroISA [99]	Heterogenec ISA	Process behavior	15%	CFI	Moving target & Concurrent execution

4.5 A *system-based* classification

Finally, the third level of the taxonomy is based on the behavior of the application and its impact on other OS and hardware structures. A valid execution of a binary is associated with a set of micro-architectural events (TLB misses, cache misses, branches taken) and OS-level events (such as the number of system calls, interrupts, stack usage, and heap usage). This behavior changes under an attack. The defense methods in this category [10, 21, 29, 38, 46, 66, 68, 95, 97, 98, 104, 107] observe the execution of the binary, and based on the observed characteristics aim to detect an attack and take appropriate measures to neutralize it. These defense methods rely on the control flow, function calls, hardware counters, branches taken, and the trace of an application.

5 Binary-based defense methods

The first category of defense methods is based on *pro-active* defense. The defense methods in this category enable protection mechanisms in a binary before any attack on its CFI, i.e., before its execution. They secure a binary by employing different methods such as additional memory checks, adding new instructions, and randomization during binary loading. The defense mechanisms are based on a thorough analysis of the attack methods, including their working, dependencies, and limitations. Note that only those defense methods that either modify the binary or finish their operation prior to

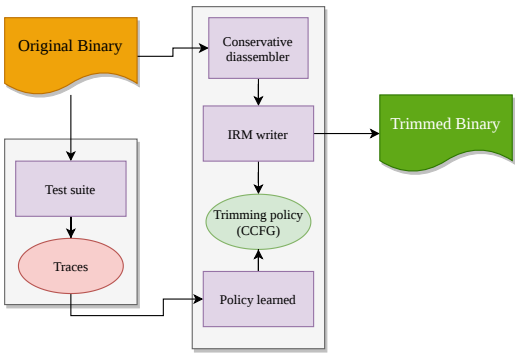


Figure 12. Method for trimming down a binary [40]. Adapted from [40]

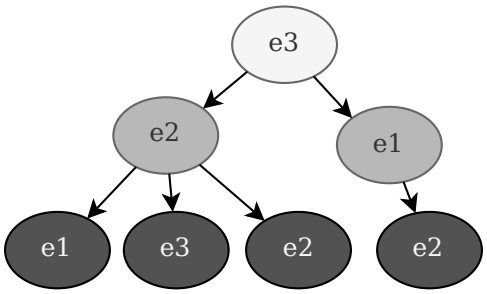


Figure 13. A sample decision tree used by [40] to form a contextual CFG. Adapted from [40]

the binary's execution (for example, layout randomization) are classified as *binary-based* defense methods (see Table 4).

5.1 Binary re-writing

We start with the methods that take an unmodified binary as an input and return a modified version that is more resilient to CFI attacks. The class of code reuse attacks (CRA attacks) rely on the presence of particular instructions in the binary to mount an attack (indirect branches). As explained earlier, they use offline processing to find code fragments that end in an indirect branch statement (*gadgets*). Finding gadgets is not hard in modern applications as they are complex in nature and have many complex code fragments (thus plenty of indirect branch statements).

However, these complex features are meant for a wider audience, and typically, a single user does not use all of them. Nevertheless, since these code fragments are a part of the binary, they can be maliciously executed. They can benefit a CRA attacker. The defense methods in this sub-category aim at removing or replacing such instructions from the binary, concomitantly ensuring that the binary's logic does not change. Doing so raises the bar for mounting a successful attack.

Following this idea, Ghaffarinia and Hamlen [40] aim to reduce the size of a stripped binary by removing such sets of features (code fragments) that are added by the developer but are not used by the end user. They even try to remove obscure features present in the binary that are no longer required and are also not tested thoroughly enough [90]. Figure 12 shows a high-level design of their idea.

They start by generating the traces of the binary execution on different unit tests provided by the user. The traces provide the information regarding what code fragments the user really needs, and also in what context. The authors explain this with an example. Assume that a required functionality F_1 executes code fragments c_1, c_2, c_3, c_4 in order. Another unwanted feature F_2 executes code fragments c_1, c_3, c_3, c_4 in order. Thinking naively, none of the code fragments can be removed. However, the authors point out that the logic that allows a branch between $((c_1, c_3)$ and $(c_3, c_3))$ can be removed. Furthermore, if another essential feature F_3 executes $c_2, c_3, c_3, c_1, c_3, c_4$ in order, then the transition from (c_3, c_3) can be allowed with the condition that it is immediately after (c_2, c_3), essentially providing *context* to a control path.

The authors create a decision tree for all such branch statements in the binary. Figure 13 shows a sample decision tree for two traces: one containing sub-sequences $[e_1, e_2, e_3]$, $[e_2, e_2, e_3]$, and $[e_3, e_2, e_3]$; and another containing sub-sequences $[e_2, e_1, e_3]$ and $[e_2, e_2, e_3]$. The paths from the leaf nodes to the root consists of all the valid paths to reach the root. The height of the tree is bounded, indicating that only a part of the history is used. The aim is to save on storage space. A flip side of this idea is that the decision trees may not capture all possible control paths. Hence, the authors have a provision for relaxing the pruning criteria based on the number of occurrences of a particular path considering all the traces together. A low number indicates low confidence.

Now, traversing a decision tree to validate all the branch statements will result in an unacceptable performance overhead. Hence, the authors encode this information as a hash and store it in a hash table. The authors leverage the hardware-accelerated hash instructions such as `sha1msg1` and `sha1msg2` to ensure a minimal performance overhead. Using a hash table, they show that the overhead of such a trimmed binary is merely 1.87%.

Trimming down a binary certainly reduces the attack surface for a CRA attack. However, modern applications generally do not have all the code built into them, and they rely on linking dynamic libraries during runtime to realize certain functionalities. These libraries are mapped into the address space of a process before they can be used, and hence, they bring back the original problem of having a large number of code fragments for a CRA attack to choose from. A classic example for this is the infamous *return-to-libc* attack [31]. The GNU C Library or *libc* provides the core libraries for different systems that use the Linux kernel. This includes access to some key functions such as `open`, `read`, `write`, `malloc`, `printf`, `dlopen`, `pthread_create`, and `exit`. The library provides a perfect place for an attacker to find gadgets

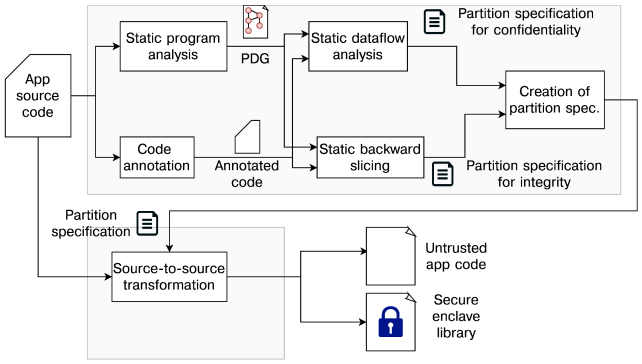


Figure 14. Working of Glamdring [63] using LLVM to split a process into secure and un-secure part. Adapted from [63]

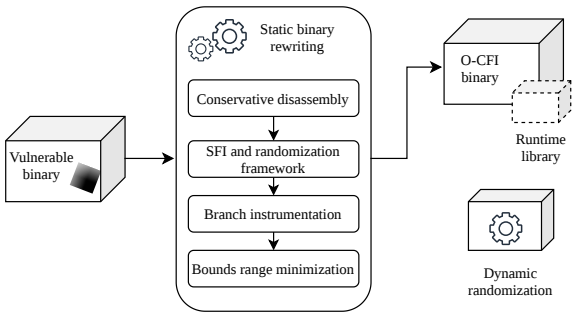


Figure 15. Working of OpaqueCFI [73]. Adapted from [73]

and mount an attack on the CFI [23, 75]. Agadakos et al. [8] aim to remove the unused code from the shared libraries using a software-based method.

5.2 Binary splitting: secure and unsecure

Modern applications incorporate many different functionalities that cater to the different needs of end users. These functionalities are generally independent of each other and are sometimes offered as "plugins" or "add-ons". Apart from these, the applications also contain ordinary features such as reading a file, opening a network connection, writing to the console, etc. These features, although important, do not require any security. Only the core logic of the binary or the modules that deal with sensitive data need to be protected [39, 57, 63].

Based on this observation, Lind et al. [63], in their work Glamdring, split a binary into secure and unsecure components. They leverage a trusted execution environment or TEE to execute the secure part of the binary where its security in terms of confidentiality, integrity, and freshness is guaranteed by the hardware. The rest of the binary executes in a traditional manner. However, splitting a binary into critical and non-critical components is not a trivial task. A module if wrongly placed in the unsecure region instead of the secure region will completely break the security of the application. Similarly, over-loading the secure region with unnecessary modules will cause the application to slow down. Hence, there has to be a balance between security and performance.

Glamdring relies on the developer to mark sensitive variables in the source code. After this, the idea is to perform a static data flow analysis to detect all the functions that have access to the sensitive variables and put those functions in the secure region of the binary. It then proceeds to perform static backward slicing to detect the functions that can write to the sensitive variables and also move those functions inside the secure region of the binary. Figure 14 shows the working of Glamdring [63]. First, the source code is annotated and is used to create a program dependency graph (PDG), where vertices are the statements, and the edges between two vertices indicate data and control dependencies. This graph is then used to perform the data flow analysis and the backward slicing creation. For the former, given sensitive information (S_F), the authors identify a sub-graph that has a dependency on S_F and move it to the secure region. For the latter, they find all the vertices in the PDG that can be used to reach S_F and move them to the secure region. They use the LLVM/Clang compiler tool chain to generate the final binary.

Another notable work in this sub-category is by Melara et al. [70]. They point out that *library operating systems* or LibOSes [9, 19] are gaining popularity in a TEE setting due to the fact that applications can be easily ported to it. However, this creates a security concern because of the large size of a LibOS that is susceptible to bugs. A security flaw in the LibOS negates the benefit of running a trusted code in a TEE setting. They propose to split the memory of a single enclave into different memory privilege regions using Intel MPK memory tagging technology. This will created an additional separation between the application and the LibOS.

5.3 Binary randomization

While executing a binary, the operating system (OS) first loads the binary file from the file system and populates the memory before it can be executed. Populating involves creating a fixed layout that has separate segments for code, data, stack, and the heap. This layout of a process is fixed and has not changed much in the past years (may vary across different architectures). It allows an attacker to make strong assumptions about the locations of different binary regions within the memory. This is crucial for a code-reuse attack, where the location of gadgets (blocks of instructions ending in a branch statement) has to be known in advance to make the attack feasible (since the attacker has to provide the addresses). The defense methods in this category make it challenging to locate a gadget within a loaded binary. They achieve this via randomizing the layout of the binary. However, this is a non-trivial task and needs support from both

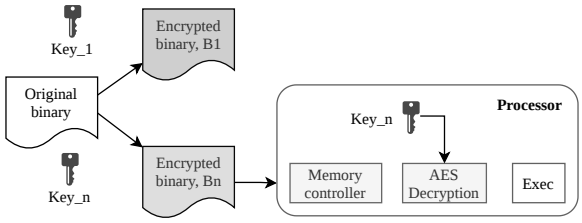


Figure 16. Working of PolyGlot [87]. Adapted from [87]

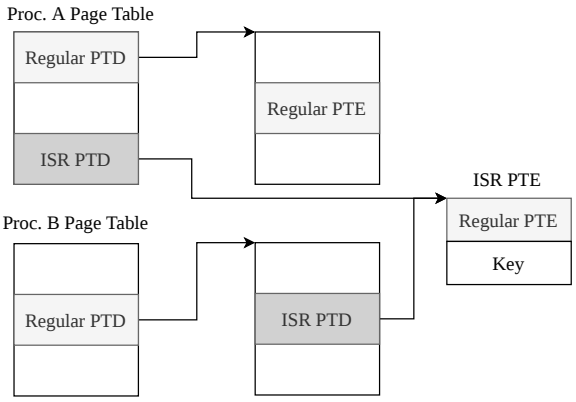


Figure 17. Storing of the keys in the page table in PolyGlot [87]. Adapted from [87]

the hardware and the OS. At a significantly coarser level, which is the default setting in the Linux kernel, the base address of the whole process is shifted by a fixed offset, resulting in a complete address space shift. The offset is changed for every execution. The point to note is that using a single offset shift allows for the easy location of objects in memory. This is called address space layout randomization or ASLR. Researchers have proposed randomization at a more sophisticated level; different segments are shuffled using different offsets to make the process of gadget discovery hard. However, a very fine-grained approach incurs more performance overheads due to the time spent in locating a memory word.

Mohan et al. [73] in their work opaque control-flow integrity (O-CFI), argue that a combination of fine and coarse-grained CFI can provide better security. They perform fine-grained randomization of the binary at load time. The authors propose to use the source code to create a destination set for each indirect branch. Then each set is reduced to its minimum and maximum value, essentially creating address-based bounds for any indirect jump instruction. They extend a prior approach to achieve fine-grained randomization of the basic blocks at load time. The objective here is to randomize the value of the bounds. This raises the bar for an attack as an attack that may be possible earlier is no longer allowed. Furthermore, they also employ an optimization method that keeps the difference between the maximum and minimum allowed jumps to a minimum (see Figure 15). The bounds are stored in a structure called the *bounds lookup table* or BLT. The BLT is stored in a hardware-protected memory and is not available to the attacker. These bounds are verified at runtime to determine if there is an attack. The logic that actually performs the checking leverages Intel MPK for its secure operation. The authors report a performance overhead of 4.7%.

Another approach for fine-grained randomization is at the instruction-level. The basic idea is that all the instructions in a binary are encrypted with a secret key. These instructions are then decrypted either in the fetch or the decode stage of the pipeline. As an attacker does not have access to the secret key, she cannot tamper with the instructions while the process is executing, thus preventing any code injection attacks. However, it cannot protect against code reuse attacks as it just needs the location of code fragments (gadgets) and not its content. This randomization technique is called *instruction set randomization* or ISR.

Sinha et al. [87] propose an instruction set randomization scheme that prevents code reuse attacks. The authors propose per-page code encryption of a binary using a random key. This key-to-page mapping is then packaged into the binary by encrypting it using an asymmetric encryption scheme, where we use the public key of a processor as its key (see Figure 16). The dynamic loader and the OS extract the key from the binary during execution and store it along with the page table entry (PTE) in the page table (see Figure 17). The decryption occurs when there is a page fault for a code page and the page is brought to the on-chip cache. The code page remains encrypted in the main memory. The data pages are not encrypted and can be accessed without any decryption. The authors point out that to prevent a code reuse attack, there are two sufficient conditions: the host binary should differ from the attacker binary and the loaded code should not be readable. The first condition is met by encrypting the binary with different keys. The second conditions is satisfied as the code pages are only decrypted when they are brought to the on-chip caches. They report an average performance drop of 4.6%.

Other notable works in this area are by Seo et al. [83] and Du et al. [34]. Seo et al. [83] address the challenge of randomization inside Intel SGX. Doing so is challenging inside SGX as the total amount of memory is low (128 MB). They implement a custom loader for the purpose. While loading the program in SGX, first, the loader is loaded into the code section and the program in the data section. In the second phase, the code in the enclave is loaded for execution. Its operation needs access to the source code and incurs a performance overhead of 2.35% to 7.6%. Du et al. [34] add new instructions to support dynamic key generation along with instructions to enable hardware-level randomization and instruction translation. Their implementation on an OpenRISC

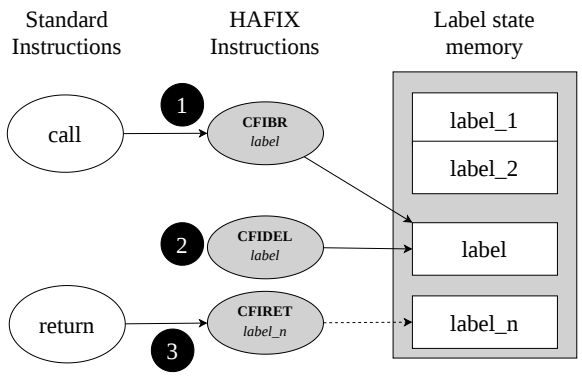


Figure 18. Architecture of HAFIX [30]. Adapted from [30])

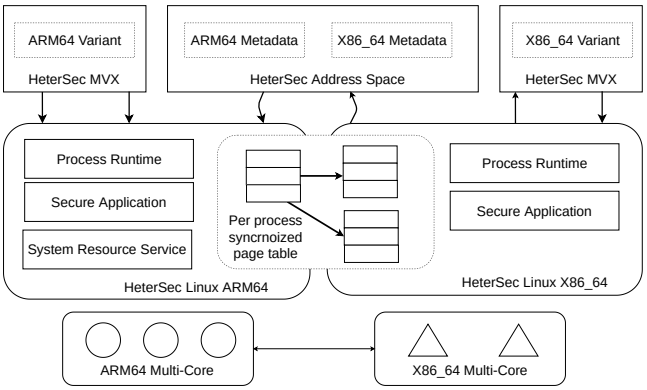


Figure 19. Design of an heterogeneous-ISA platform [99]. Adapted from [99]

processor running Linux leads to a performance overhead of less than 0.7%.

5.4 Control flow tracking of binaries

The instruction sets (ISAs) for different architectures were designed with only one objective in mind – performance. They did not have dedicated security-specific instructions. Hence, adding security and performing verification was the responsibility of the compiler; this had an adverse effect on the runtime performance of the binary. Using traditional instructions to do the verification results in a significant performance overhead. Researchers have proposed modifications to the ISA to make a security check a native operation.

Davi et al. [30], in their work HAFIX, propose a novel mechanism to prevent return oriented programming (ROP) attacks. The key idea here is to restrict the target of a return instruction to within the functions that called a *call* instruction. In doing so, HAFIX severely restricts the ability of a return statement to jump to a point anywhere in a process' address space. To achieve this, the authors propose a modified ISA that contains CFI instructions. Along with this, they also propose a new compiler that automatically adds these CFI instructions to the correct places in the binary.

A brief overview of HAFIX is shown in Figure 18. In order to track what function is currently executing, HAFIX modifies the compiler to assign unique labels to each function. Furthermore, it introduces three new instructions `CFIBR`, `CFIDEL`, and `CFIRET` to track functions during execution. HAFIX ensures that every function executes the `CFIBR` instruction as its first instruction. This instruction will load the unique-id of that function in a dedicated area of the memory indicating that the function is now executing (function activated). The `CFIDEL` instruction is executed just before a function returns. This will remove the id from the dedicated memory region, indicating that the function is done with its task. To ensure that a return instruction does not randomly jump to an address in the process's address space, a return is only allowed to a `CFIRET` instruction that has a pointer to the currently executing function.

To support recursive function calls, HAFIX adds a new instruction called `CFIREC` and a register `CFIREC_CNTR`. In case of a recursive function call, the compiler ensures that the first instruction called is `CFIREC` (instead of `CFIBR`). The instruction `CFIREC` activates the function only if the value of `CFIREC_CNTR` is zero. Otherwise the function is already activated, and the activation is skipped. Every time `CFIREC` is encountered, the value of register `CFIREC_CNTR` is incremented by one. Similarly, upon encountering the instruction `CFIDEL`, if the value of `CFIREC_CNTR` is more than one, then it is decremented by one. The authors report a performance overhead of 2% due to these new CFI instructions.

5.5 Heterogeneous architectures

The attacks on a binary that leverage the instructions in the binary assume a particular architecture. This assumption can be broken by compiling a source code to generate two different binaries targeted to different architectures. These binaries can then be executed in parallel, and their state can be compared to ensure that they match. A mismatch indicates that there is an attack. This is a non-trivial task, and doing so efficiently is a challenge.

Wang et al. [99] propose a framework to ensure the CFI of a binary by using heterogeneous ISA setups. They use real-world x86 and ARM machines, and the framework provides an illusion to the binary that it is executing on a multi-ISA chip multiprocessor (CMP). They evaluate two methods for protecting the CFI of a binary on a multicore chip: *moving target defense* (MDT) or *multi-variant execution* (MVX). In an MDT execution, the execution state of an application is probabilistically migrated from one ISA to another. This creates a challenge for a CRA attacker as she has to guess the timing of these migrations and chain the gadgets in such a way that they execute correctly across different executions on different ISAs. In an MVX execution, multiple copies of the executions are launched across different executions, and their state is monitored (such as the return value of system

calls, segfaults, etc.). Any divergence in the value of these states indicates a potential attack on the binary.

In order to support dynamic migrations across different ISAs, the authors propose a distributed kernel that helps to ensure a consistent state for a "protected application". The first challenge is that the memory state of the application has to be kept synchronized in order to ensure a valid state when migrating. For this, they propose a per-process on-demand synchronized page table (see Figure 19). The second challenge is the management of global resources such as open files, semaphores, and sockets. To support this, they introduce the concept of a *master node* and a *follower node*. Here, the application is provided the resources from the master node, and the follower node maintains a virtual descriptor table (VDT). When the application performs any operation on the follower node, the VDT is used to issue an RPC (remote procedural call) to the master node to service the request.

6 Process-based defense methods

The defense methods in the previous section (Section 5) aim to secure a binary by either putting CFI instructions directly in the binary or by removing instructions from it that are of no use to an user but can be potentially used by the attackers to mount an attack. These methods have the advantage of providing fine-tuned defense methods as per the application's requirement. However, most of them need access to the source code to support compiler-enabled defense methods. Apart from this, they are also limited in terms of what can be done to secure a binary. The modifications to a binary have to take into account various size, compatibility, portability and performance overheads.

The defense methods in this category do not modify the binary. Instead, they focus their attention on protecting the state of the application while it is being executed. The state of a process constitutes of the data contained in its address space – the stack and the heap, along with data structures that a operating system maintains to enable an efficient and fair execution in modern multi-core and multi-tenant systems.

Some of the defense methods create a dynamic form of the defense methods that are a part of the previous category (*binary-based*). For example, layout randomization is generally done at the time of loading, and then it mostly stays the same throughout the life cycle of a process [92]. Here, a dynamic component is added to it by enabling randomization of the layout throughout the life cycle [22, 37]. Apart from this, some of the defense methods require an initial setup for profiling the applications to enable these defenses. However, as most of the work is done while the application is executing, we put those defense methods in the *process-based* category instead of the *binary-based* category.

Table 5. Defense methods protecting key program components and structures.

Defense method	Hardware feature	Applicatio attribute	Perf.	Attack model	Method
HWTag [106]	HW Tag	Kernel memory	0% – 4%	Complete protection	Small TCB (Kernel)
MicroPolTag [49]	HW Tag and ISA changes	Reference monitor	-	Policy violations	Micro-policies
CCFI [69]	Registers & AES-NI inst.	Pointers	18% – 38%	ROP and JOP	MAC
StackCanary [28]	Register	Stack	3.2%	ROP	A parallel shadow stack
TaggW [81]	HW Tag	Stack	5.7%	ROP	Stack integrity
HCIC [108]	PUF	Return pointers	.95%	ROP	Encryption
HWAddrSan [85]	Tags	Stack	2×	Overflow	Validation of memory addresses
REST [88]	Novel HW	Stack and Heap	2%	Memory errors	Cache-line
LightCFI [80]	AES Inst.	Memory addresses	4%	ROP & JOP	Encryption of the addresses.
PACItUp [62]	ARM pointer protect.	Pointers	0.5% (code) & 19% (data)	ROP	MAC checks in pointers
HWCDI [61]	ISA modification	Control data	0.19%	Data flow attacks	Encoding
Morpheus [37]	HW Tags	Test	1%	ROP	Randomization
ShadowStack [16]	Intel MPK	Stack	2%	ROP and JOP	Shadow-stack
ShadowGuard [18]	IPT	Stack	2%	ROP	Selective shadow stack
ASIST [22]	Registers	Instructions	1.5%	ROP	Dynamic ISR
PACSafe [48]	ARM pointer protect	Pointers	30%	ROP	PAC chain
FineDIFT [20]	HW Tag	Data objects	5-6%	Integrity	Tags
Shakti-T [71]	HW enforced base and bound	Pointers	–	Spatial and Tempral protection	Branch and Bound
RetTag [101]	New instructions	Pointers	0.11%-7.69%	ROP	PAC

6.1 Dynamic process randomization

The defense methods in this sub-category aim to enable dynamic randomization of an application's variable-to-address mapping while the process is executing. The general idea here is to prevent an attacker from locating code snippets, also known as *gadgets*, in the address space of the application – a key requirement for CRA attacks.

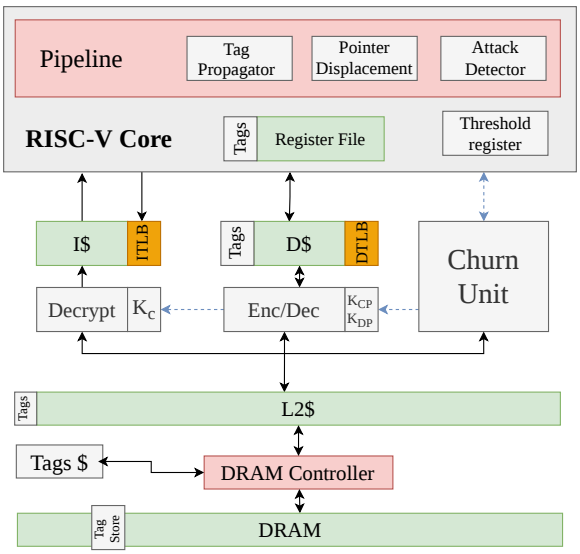


Figure 20. Architecture of Morpheus [37]. Adapted from [37]

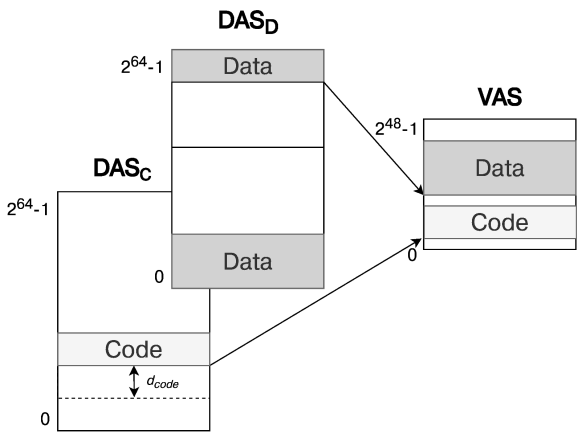


Figure 21. Pointer displacement defense in Morpheus [37]. Adapted from [37]

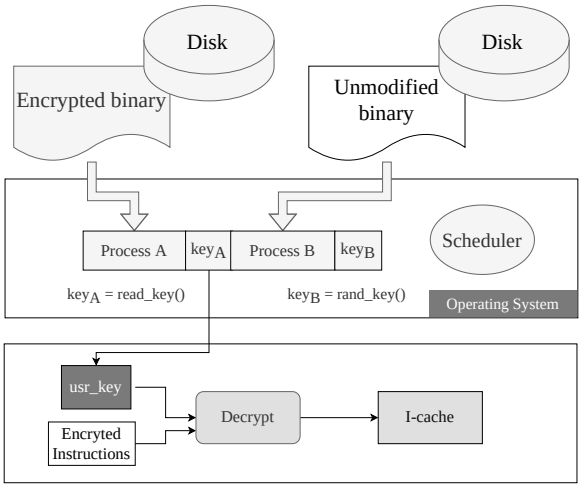


Figure 22. Architecture of ASIST [22]. Adapted from [22]

Gallagher et al. [37] in their work *Morpheus* argue that a single round of randomization of the contents of an application, irrespective of the granularity, is vulnerable to disclosure attacks from a patient attacker. The attacker can deploy a naive scanner that scans the complete address space to find the code fragments required to mount an attack. Since randomization is only done at load time, the locations of those fragments are fixed for the duration of the execution.

They propose a novel hardware-assisted, domain-wise, constant randomization (churning) of the application during execution. They point out that a valid execution of an application relies on the defined semantics of the language and, to a lesser extent, also on the undefined semantics. However, an attack relies almost entirely on undefined semantics such as out-of-bounds access, overwritten return addresses, and jumping to a random location in the code. Hence, they argue for the randomization of these undefined semantics. In order to do so, they divide the address space of an application into four categories: code (C), code pointers (CP), data pointers (DP), and data (D). They use a two-bit tag to encode this information with every memory object. Initially, these tags are assigned by the compiler (an LLVM extension). As shown in Figure 20, they modify the micro-architecture to propagate these 2-bit tags all the way in the pipeline of the processor. These bits are used to perform error-checks in the pipeline as follows:

Ops	Check Condition	Rule
Execute	Insn.tag != C	Only execute C
ANY	R1/R2.tag == C	No C in the pipeline
JAL(R)	R1.tag != CP	Jump target must be CP
LD/ST	R1.tag != DP	Address must be a DP

The second defense they employ is a *pointer displacement defense*. Every CFI attack relies on locating the memory object of interest within the process's address space. The authors add displacements to separate regions of the address spaces: code address space (DAS_C) and data address space (DAS_D). They are located at a fixed offset from the original virtual address space (VAS). Hence, a $VAS \rightarrow DAS$ translation, and vice-versa, requires a simple arithmetic operation (see Figure 21).

Finally, Morpheus introduces domain-based encryption. It essentially encrypts the content in different domains with a domain-specific key. The tag associated with the memory object is used to select the correct key for its encryption or decryption. They use the address of the object to ensure that the memory objects containing the same value do not output the same ciphertext. They report an average performance overhead of 0.84% and a worst-case overhead of 6.71%.

Christou et al. [22] in their work ASIST, propose hardware-assisted instruction set randomization (ISR) to protect the CFI of an application from random code injection attacks. They argue that a software-based ISR incurs high performance overheads: up to 290× in an emulator and 2.9× on bare-metal machines. Furthermore, they also point out that prior work statically encrypts a binary using a fixed key. This makes the encrypted binary vulnerable to information disclosure attacks where an attacker aims to leak the key used for encryption. This also restricts the dynamic linking of shared libraries (as they will not be encrypted). All the libraries that are to be used must be statically linked and encrypted before use.

Hence, the authors propose dynamic encryption of the instructions in the binary at the time of their usage. To enable this, they encrypt the code pages when a process faults while accessing them for the first time. In this case, a per-execution key k is generated, an anonymous page is allocated, and the content of the code page is copied to the anonymous page and then encrypted. The key k is stored in a register `usrkey`. This key is then used to encrypt the code pages on a fault and decrypt instructions by the CPU during the rest of the operation (see Figure 22). They use a separate key for the kernel, and that key is stored in a register called `oskey`. As shown in Figure 22, the instructions are decrypted before they are stored in the on-chip instruction cache.

Furthermore, to prevent ROP attacks, they propose to encrypt return addresses using the process's key just before a function call. If the attacker overwrites the return address, it will not decrypt to a valid value (as the key is secret), leading to a process crash. They incur a performance overhead of 1.5% and require 0.7% additional hardware.

6.2 Protecting the stack

During execution, an application makes many functions calls and uses a lot of temporary variables to finish its task. In order to enable this, an application is associated with a memory region known as the *stack*. The stack of an application is used to store temporary variables, return addresses, function call arguments, and spilled register values. It uses a LIFO (last-in, first-out) policy. This allows it to place the most recent data on top of the stack so that it can be accessed quickly. The addresses stored on the stack determine the control flow of the application after a function returns. Hence, the stack of a binary is a popular target for attackers to violate the CFI of an application. By manipulating the values stored on the stack, the attacker can divert the control flow any way she likes. Hence, multiple defense methods aim to protect the stack of a process against manipulation while ensuring a minimal performance and storage overhead.

The idea of using a *shadow-stack* to ensure the consistency of the "main-stack" has attracted the attention of many security researchers. A shadow stack is a secure region in the memory that stores either a complete or a partial copy of the main stack. The values in the main stack are compared with the values stored in the shadow stack prior to using them, and an error is flagged in case of a mismatch. In a completely protected binary, all the values on the stacks are verified.

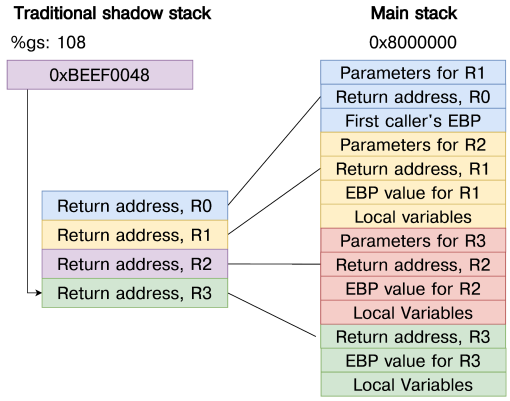


Figure 23. Traditional approach for maintaining a shadow stack [28]. Adapted from [28]

Dang et al. [28] discuss the traditional approach of maintaining a shadow stack. They argue that the traditional approach is inefficient in terms of looking up values in a shadow stack; moreover, while handling certain instructions such as *longjmp* and *setjmp* it may unwind the stack (pop return addresses) multiple times. Figure 23 shows the working of a traditional shadow stack. Here, the shadow stack is located at a fixed memory address in the address space of the process. The return addresses that are stored on the main stack are also pushed into the shadow stack. A *shadow pointer* is maintained that points to the top of the shadow stack. However, in the case of a *longjump* it may be the case that the main stack is unwound multiple times. In that case, the value at the top of the shadow stack will not match that on the main stack. A possible solution here is to keep popping values from the shadow stack till a match is found. However, this is not a complete solution (what if multiple functions are called multiple times).

To this end, the authors propose a design for a parallel shadow stack (see Figure 24). Here the shadow return pointers are maintained at a constant offset from the values stored on the main stack. This simple idea solves many issues. Now, there is no need to maintain a *shadow stack* pointer as the shadow pointers can be easily accessed by a simple arithmetic operation on the addresses. Furthermore, this design naturally handles the issues raised while using *longjmp* instructions. Irrespective of the addresses used, the corresponding shadow address can be easily computed. The authors also note the issue of *time of check to time of use* or TOCTOU attacks. They discuss two scenarios: prologues where an attacker modifies the value of the return address before it can be placed on the stack and *epilogues* where the return address value is modified after it has been verified against a value

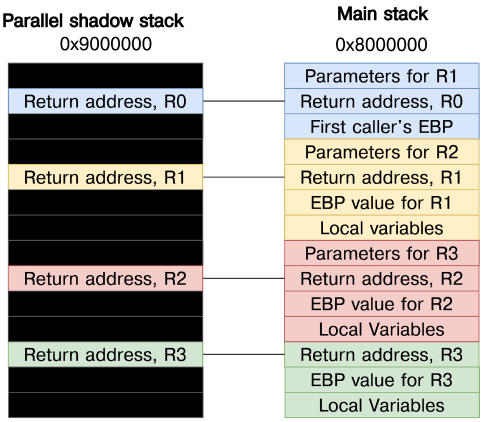


Figure 24. Working of a parallel shadow stack [28]. Adapted from [28]

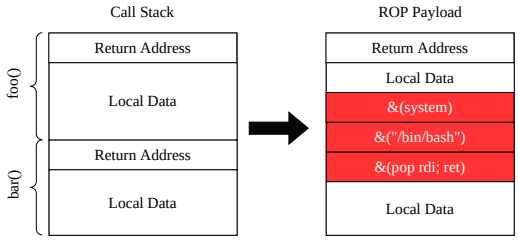


Figure 25. Return oriented programming or ROP attack on a stack [16]. Adapted from [16]

stored on the shadow stack. However, as noted in the later work, doing so requires a precise idea of the time required for different memory accesses [16]. Using their design of a parallel stack, they were able to bring down the overhead of the shadow stack from 10% to 3.5%.

Burow et al. [16] thoroughly analyze the different mechanisms of implementing a shadow stack and different ways to compare the addresses stored on them with values on the main stack. They focus on preventing return-oriented programming or ROP attacks on a process. As explained earlier, in an ROP attack, the backward edges of the control flow, i.e., the return addresses stored on the stack of an application, are overwritten with attacker-provided data (see Figure 25). Earlier, these were used to return the control to attacker-provided code (present in the overwritten values). However, with the DEP protection in place where stack pages are non-executable, the return is to a code-fragment that is already a part of the code (gadgets). Many gadgets can be chained together to execute malicious logic. The authors propose two methods to implement a shadow stack: direct shadow stack and compact shadow stack. These were referred to as parallel stack and traditional stack in the prior work [28].

They point out that a directly mapped shadow stack provides an efficient way to compute the addresses. However, it increases the memory overhead by 2× since we need to maintain a copy of the main stack. Furthermore, as it needs to be a fixed offset from the main stack, it interferes with the usage of the address space by other threads in the application. They propose to use a dedicated register to store the offset and use a per-thread offset that is dynamically determined. This solves the issue of address-space hogging by the shadow stack.

For a compact shadow stack, the system needs to maintain a dedicated shadow stack pointer. The authors argue that if the pointer is stored in the memory, then it is accessed twice for a single function call: during a call to store the return address and during a return to validate the return address. Hence, they propose to use a dedicated register to store the stack pointer as well. This reduces the number of memory accesses and dramatically improves performance.

For comparing the return addresses, the authors propose two mechanisms: either compare the return addresses from the main stack and the shadow stack and proceed only if they match or just use the address from the shadow stack. The former approach allows for the detection of an attack and subsequent recovery steps, whereas the latter approach silently stops an attack. They report a performance overhead of 5.78% and 5.33% for both direct and compact stacks. The latter scheme uses a dedicated register to store the stack pointer. As the performance overhead in both is the same, the authors argue for using compact shadow stacks due to their low memory overhead.

Chamith et al. [18] argue that shadow stacks have not gained popularity due to the high performance-overhead incurred by them. They state that the prior work has focused on an efficient implementation of the shadow stack by either providing a better lookup scheme or limiting the amount of data that is stored on the shadow stack. They focus on the policy aspect of a shadow stack by arguing that not all the functions in a binary or all the paths of a binary need to be checked during an execution. The possibility of a stack overwrite is only from a control path that actually writes on the stack. The authors perform a static analysis on the binary at a functional and basic block level. A function is tagged as a safe function if it does not write anything on the stack above the local stack frame and does not call any other function that is tagged as a non-safe function. The authors use graph analysis to mark all the functions as either safe or unsafe starting from the leaf level. Now, it may be possible that a function is marked as unsafe due to a statement that is conditional, i.e., it does not appear in all of the control paths of the function. Hence, the authors repeat the above procedure at a basic block level, and all the control paths that are marked as safe are skipped from a shadow stack check. Doing so reduces the overhead from 8% to 2%.

6.3 Pointer verification

A shadow stack offers protection to the stack at the expense of performance and storage overheads. Ensuring the consistency of every object on the stack impacts the performance and increases the storage cost to maintain the metadata related to the objects on the main stack. Researchers have identified that it is not required to protect the complete stack, and we can get away with ensuring the consistency of some objects. The methods in this sub-category aim to follow this idea and improve the performance and the storage cost.

6.3.1 Trip-based error detection

A dynamic memory pointer is used to determine the control flow of a process. We can protect a pointer in two ways [109]: *spatial protection*, the pointer should not access any memory outside the region allotted to it, and *temporal protection*, which restricts the "use-after-free" type situations. As explained earlier (refer to Section 2.1), unsafe languages do not have explicit memory checks. The defense methods in this sub-category aim to add memory checks intelligently to ensure a minimal performance overhead.

Serebryany et al. [84] proposed a tool called *AddressSanitizer* to reduce the number of memory errors. It provides spatial and temporal protection to memory pointers by using a shadow memory to store encoded forms of valid memory addresses. This information is then used to ensure that all memory accesses are valid.

To design the shadow memory, AddressSanitizer uses the fact that an 8-byte memory can be in one of 9 states: either all of the memory is safe to access or only k bytes, $0 \leq k < 8$ are accessible. If $k == 0$, then all the bytes are accessible. They use one byte to encode this information and store it in the shadow memory. This encoding mechanism also allows an efficient lookup: given an address `Addr`, the address in the shadow memory is `Addr >> 3 + Offset`. The `Offset` is used to control the location in the shadow memory such that it does not interfere with other memory regions of the application. The validation logic is also quite simple.

```
ShadowAddr = (Addr >> 3) + Offset;
k = *ShadowAddr; // Valid Addresses
if (k != 0 && ((Addr & 7) + AccessSize > k))
//AccessSize can be: 1B, 2B, 4B, or an 8B.
ReportAndCrash(Addr);
```

Listing 4. Logic to detect an invalid access in AddressSanitizer [84]

Furthermore, AddressSanitizer provides spatial protection by modifying the `malloc()` call to allocate extra memory around the originally allocated region. The authors call this region the *red zone*. Any attempt to access pages in this red zone will signal either an underflow or overflow error. Also, to provide protection against temporal memory errors such as *use-after-free*, AddressSanitizer modifies the `free()` call to put the freed region into a state of *quarantine*. This region will not be allocated by the `malloc()` call. The basic idea here is to delay the allocation of the freed memory region so that any immediate access to it after it has been freed can be detected. Naturally, the size of this quarantine region needs to be limited, otherwise it will result in significant memory pressure.

Based on this, they were able to detect many previously undetected errors. However, this resulted in an overhead of 2×, 3×, and 3× for CPU performance, memory usage, and binary size, respectively. The CPU performance worsened due to the additional instructions for shadow memory checks.

To remedy this situation, Serebryany et al. [85] proposed a hardware-assisted shadow memory check scheme that leverages the memory tagging feature on modern hardware. Memory tagging allows appending *TS* bits (tag size) to every *TG* bytes (tagging granularity) of the memory. The authors used these bits to store the shadow encoding, thus resulting in a significant drop in the total memory usage (4% from 3×) [85]. This also simplifies the validation logic and brings the overhead in the binary size down to 50% from 3×. However, the CPU overhead remains the same (2×). This is primarily because of the overheads of tagging the heap/stack objects during their allocation and deallocation.

A similar tool called KASAN [4] exists to find memory-related bugs in the Linux kernel. It can be enabled by setting `CONFIG_KASAN=y`. If enabled, the compiler will insert function calls (`asan_load*(addr)`, `asan_store*(addr)`) before every memory access. The logic to translate a kernel address to its shadow memory address remains the same as before [84]:

```
static inline void *kasan_mem_to_shadow(
    const void *addr)
{
    return ((unsigned long)addr >>
        KASAN_SHADOW_SCALE_SHIFT)
            + KASAN_SHADOW_OFFSET;
}
```

`KASAN_SHADOW_SCALE_SHIFT` and `KASAN_SHADOW_OFFSET` are configurable variables.

The hardware optimizations help us get around some of the limitations of purely software-based approaches such as AddressSanitizer. However, the overall performance overhead is still pretty high, and hence, limits the usage of AddressSanitizer in production binaries. Nevertheless, AddressSanitizer has detected previously unknown bugs that were able to bypass rigorous testing phases.

Hence, to further reduce the performance overhead, Sinha and Sethumadhavan [88] propose another hardware-based trip-based memory corruption detection mechanism called Random Embedded Secret Token (REST), which is based on

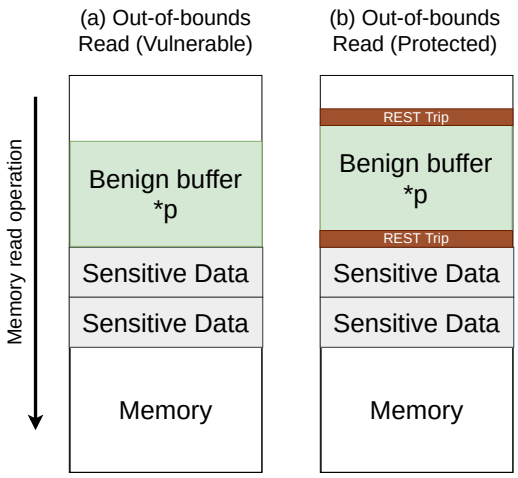


Figure 26. Spatial pointer corruption detected by using REST [88]. Adapted from [88]

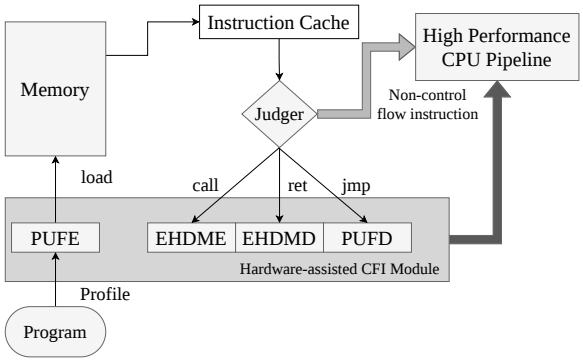


Figure 27. Architecture of HCIC [108]. Adapted from [108]

large random values. These REST tokens are stored on the stack of a process. A pointer object is surrounded with a red zone memory filled with REST tokens as shown in Figure 26. Any access to a REST token present in the red zones will cause privileged memory safety exceptions.

6.3.2 Encryption-based error detection

Apart from such trip-based techniques, another promising approach is to encrypt the addresses and decrypt them before using them. An invalid address indicates an attack on the control flow of the binary.

Zhang et al. [108] in their work HCIC, propose a mechanism to validate the return addresses and indirect branch addresses before the corresponding branch/jump is executed. These instructions are used by return-oriented programming (ROP) and jump-oriented programming (JOP) attacks, respectively. By ensuring that every such instruction is validated, they thwart both of these attacks. In case of an ROP attack, the attacker uses the buffer overflow vulnerability to overwrite the return address stored on the stack to a new address that points to a chain of *gadgets*. To prevent that, the authors calculate an encrypted hamming distance (EHD) between the return address and a key generated using the PUF (Physical Unclonable Function) [5, 13] before the address is stored in the stack. The EHD is encrypted using the same key and is stored on the stack. When the function call returns and the return address is popped from the stack, the EHD is calculated again and is expected to match the one calculated earlier. If it does not, a fault is raised. Any tampering with this process can be detected; this will fail the test prior to executing the return statement (see Figure 27).

To prevent JOP attacks, the authors scan the loaded application and encrypt the first instruction at the target address. Thus, during an attack, forcing the jump to an unencrypted instruction will lead to a system error. The key used for encryption (and decryption) is generated dynamically when the application is loaded. Furthermore, they propose a hardware extension to store the keys into dedicated registers and a module to dynamically generate the keys. Figure 27 shows a high-level design of HCIC [108]. The authors report a negligible performance overhead of 0.95%.

Other notable works in this category are by Qiu et al. [80] and Mashtizadeh et al. [69]. Identifying the need for an efficient encryption and decryption method, Qiu et al. [80] provide a unique way to prevent ROP (return oriented programming) and JOP (jump oriented programming) attacks. Their proposal adds an LEA-AES module to the CPU architecture. This is responsible for the encryption and decryption of return addresses during the execution of the binary. ROP and JOP rely on modification of these addresses to successfully mount an attack. It proposes encrypting the return address when the *call* statement is executed, and decrypting it when *ret* is called. If the return address is modified in between, the decrypted address will be erroneous, which will result in a code exception. *CCFI* (Cryptographically enforced control flow integrity) by Mashtizadeh et al. [69] uses a message authentication code, or MAC, to protect the pointers stored in the memory. The MAC function uses the value of the pointer and its location to generate the final hash value. This prevents the swapping of two-pointers in memory. The key used to generate these MACs is stored in a dedicated register to prevent their leaking. They report a performance overhead of 18-38%. This method, however, cannot prevent a replay attack if the attacker knows the value, location, and MAC of an old pointer.

Liljestrand et al. [62] use the ARM ISA's pointer authentication (PA) mechanism to ensure the security of data pointers during execution. As already mentioned before, the PA mechanism generates MAC signatures (called *pointer authentication code* or PAC) that are associated with every memory pointer and are verified before using the address. The signature is generated using a secret key, address, and the current stack pointer. Since an attacker does not have access to the key, she cannot generate a return address with a valid PAC.

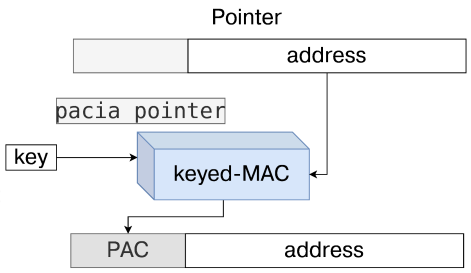


Figure 28. Working of ARM's pointer authentication code. The instruction `pacia` is used to generate the signature of the pointer and store it in the unused bits. The size of the signature depends on the system's configuration [62]. Adapted from [62]

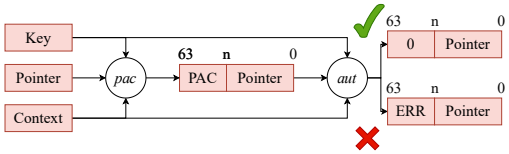


Figure 29. Working of PACSafe [48]. Adapted from [48]

Figure 28 shows the generation of the PAC using the instruction `pacia` that uses the instruction code key for signature generation. PA provides 5 different keys: two for code, two for data, and one generic [25, 62]. The instruction `autia` is used to validate the pointer that contains the PAC generated using the `pacia` instruction.

The authors start by listing the limitations of the current PA implementation. First, the current setup is vulnerable to replay attacks. An attacker can reply an old address, and that will be assumed to be valid as the MAC will match (since the same key is used). To prevent this, the authors propose to add additional checks during the validation of an address. They encode additional information about the data type a particular pointer is pointing to. They use the LLVM compiler's *ElementType* feature for this. Furthermore, to prevent the overwriting of return addresses, a return address is encoded with a compile-time nonce (*function-id*) to ensure that it cannot be reused later.

Hohentanner et al. [48] also leverage the ARM pointer authentication feature in their work PACSafe to ensure memory safety in applications developed using C/C++. PACSafe provides complete spatial and temporal protection for an application, including its stack, heap, and global variables.

Figure 29 shows the working of PACSafe. The main idea is to use a shadow copy of a pointer. Every object is associated with a unique ID, which is used to calculate the PAC for a pointer during its creation. Upon de-referencing, the PAC is checked again. If it was manipulated to point to a different object, it would get a different ID, and the PAC check would fail. When the object is deallocated, its ID is removed from the system. This will ensure that an attacker can no longer use any pointer pointing to it – ensuring temporal protection.

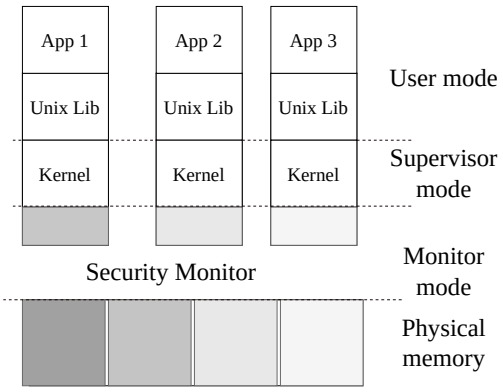


Figure 30. Architecture of Loki [106]. Adapted from [106]

6.4 Runtime metadata

A compiler uses the information ("metadata") available in source code to optimize the corresponding generated binary. This metadata is subsequently discarded and is not a part of the generated binary. Researchers have observed that this information, either whole or in part, can be used to validate the execution, and in turn, the CFI of a binary. However, augmenting a binary with this information is not trivial, since it negatively affects the size and the performance of the binary. Furthermore, adding a lot of memory checks defeats the purpose of using unsafe languages in the first place.

Leveraging hardware support to store the metadata and propagating it through the pipeline during the execution reduces the performance overheads vis-a-vis similar software-based methods [60, 76, 79]. This metadata can be used to implement access control or security of the data. Upon detecting an invalid access, the hardware generates and exception and runs a custom handler.

Zeldovich et al. [106] argue that due to the semantic gap between the software and hardware layers, existing hardware support is seldom adequate. Due to this, developers rely on software solutions embedded in the application to ensure the security of the data. Hence, the trusted computing base, the components which the user must assume to be secure, consists of the OS and the software. Even if the software is leveraging a hardware feature to provide security, the final decision is made by the software. Any bug in the trusted code base or TCB (the part of the software that is assumed to be trusted) can be exploited by an attacker to violate the CFI, and given the size of the TCB, the attacker can easily find memory vulnerabilities providing an opening point for an attack. Hence, the authors argue that the security of the data must be handled by feature-rich hardware with support from a small kernel code that is optimized for security.

The authors propose to use a tagged memory architecture to ensure hardware-enforced application security policies. They implement their prototype on an OS that is designed to have a small trusted kernel. They use the HiStar OS and modify it to support tagging. They term the new OS as *LoStar*. Furthermore, they introduce a trusted software component that runs underneath the kernel in a special privileged mode called the *monitor mode* (see Figure 30). They associate a 32-bit tag with every 32-bit memory location. The monitor software enforces the data policies and handles any faults resulting from the tags. Other traditional interrupts, such as the timer interrupts, cache misses, page faults, divide-by-zero, are handled by the OS.

Working on similar lines, Hritcu [49] propose to use the hardware tags to enforce micro-policies for memory safety, control-flow integrity (CFI), compartment isolation, taint tracking, information-flow control (IFC), and dynamic sealing. They propose minimal changes to the ISA of a RISC architecture to implement their solution. They tag every word in the memory with a tag. This tag is then propagated in the pipeline to enforce policies. To propagate tags efficiently, they propose a hardware structure called *rule cache*, which is responsible for ensuring whether the current operation is valid or not, and if it is, decide on the tag of the result.

Roessler and DeHon [81] also propose an implementation of software-defined policies to protect objects on the stack by leveraging hardware-based tags. They propose that every object be associated with a tag. These tags are propagated by the hardware in the pipeline while maintaining the policies defined by the user. They propose two different modes of working. First is the protection of only the return address pointers, and the second is protecting every stack object. They report a performance overhead of 1.2% and 5.7%, respectively.

Chen et al. [20] use dynamic information flow tracking, or DIFT, to track the integrity of data items in an executing process. This process is used to generate a robust data flow graph, or DFG, which is typically not possible using a static binary. The main aim is to ensure that the application follows the developer's intentions. Any deviation from it indicates a violation of the integrity of data structures, which can eventually be used to divert the control flow of an application (DOP attacks).

Figure 31 shows the workings of FineDIFT. The authors developed a co-processor responsible for implementing policies to protect the integrity of the data. To do so, they leverage tagging data elements to track them across execution cycles. While processing an instruction such as a `store` instruction, the co-processor uses related information such as the target address and the store size to fetch the tag and the policies associated with the data object. The policy and the tag are then used to determine whether the corresponding `store`

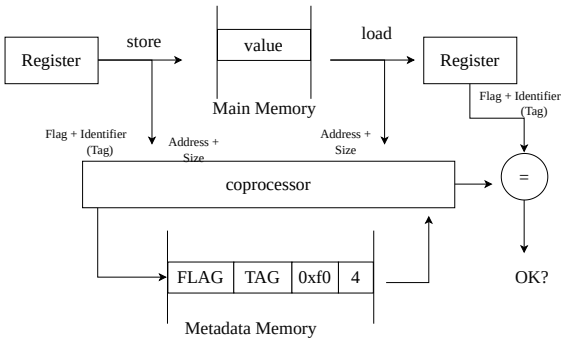


Figure 31. Working of FineDIFT [20]. Adapted from [20]

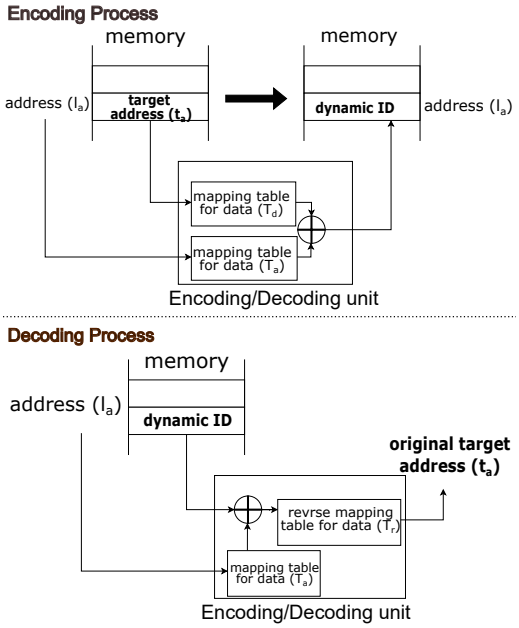


Figure 32. Encoding and decoding in HW-CDI [61]. Adapted from [61]

operation is allowed or not. The metadata is stored in a CAM (content addressable memory).

6.5 Protecting control flow data

Lee and Lee [61] propose to protect the control flow data instead of the control flow information. They argue that the control flow of a program is dependent on the value of the control variables, and this is what is targeted by the attackers to hijack the control flow. In the work HW-CDI [61], the authors propose to encode the control data information at runtime and decode it prior to use (see Figure 32). They propose a hardware extension to generate the key for encoding and decoding data variables dynamically, which is not only based on the value of the control data but also its address in the address space. They propose two new hardware instructions *emov, dmov* similar to the already existing `mov`

instruction, to move the data to and from memory. `emov` or encoded-mov is used to store the control data in memory, and `dmov` or decode-mov is used to load the control data from memory. They claim a performance overhead of less than 1% (compare this with the reported 21% performance overhead of the original CFI proposal [7]).

Menon et al. [71] discussed the issue of spatial memory attacks (accessing data beyond its size) and temporal memory attacks (using invalid pointers) in RISC V-based systems. They start with a discussion on different attacks such as buffer-overflow attacks, return-to-libc attacks, and ROP attacks. They also discuss different ways to prevent these attacks such as encryption of the code pointer so that it cannot be modified by an attacker, using a shadow stack, randomization, and control flow integrity checks. The authors pay special attention to defense mechanisms using “fat-pointers" to prevent spatial attacks on the pointers. Here, every pointer is associated with a base and bound value that defines the region the pointer can access. Anything beyond this, results in an error. However, this presents a challenge in terms of pointer storage as every pointer has to be appended with additional information. The overhead of storing them can be non-trivial. Prior work has proposed storing an embedding of the base and bound information in the unused bits of a 64-bit pointer. Although this solves the storage issue, the embedding causes memory fragmentation as the memory can now be allocated only in fixed sizes.

To solve the challenges associated with the “fat-pointers" and memory fragmentation, the authors propose Shakti-T [71], a set of ISA extensions to RISC-V processors that provide performance-efficient security solutions that protect against spatial and temporal memory attacks. To achieve this, they propose Pointers Limit Memory, or PLM, which is used to store the base and bound information of the pointers. Every pointer is associated with its own `ptr_id` that acts as an offset into the PLM. To ensure quick access to the PLM, the authors introduce a new register called the Pointer Limits Base Register (PLBR) that will contain the base address of the PLBR. Here, every aliased pointer (pointers pointing to the same object) will point to the same entry in the PLM. For example, assume there are n aliased pointers. Using the earlier approach, the total storage overhead will be $2n$: one base and one bound value for every pointer. However, in Shakti-T, we only need $n + 2$ storage units, one `ptr_id` for every pointer, and a common base and bound value. Using this approach, they were also able to reduce the storage overhead by up to 4× by using a common storage area for pointers pointing to the same object.

This design also takes care of temporal attacks. Any data freed by any aliased pointers will modify the common base and bound values, which will be checked upon an access by any other pointer. However, if all the pointers point to different objects, the proposed solution has an additional overhead. In the previous example, let’s assume we have

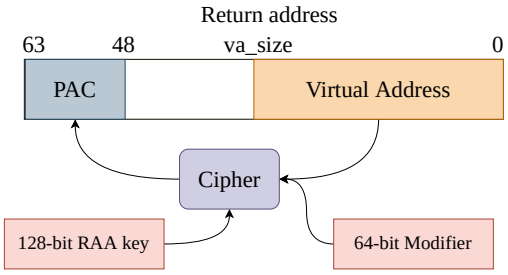


Figure 33. Generation and storage of PACs [101]. Adapted from [101]

n unique pointers. Then, prior work will have a storage overhead of $2n$, whereas, in Shakti-T, the overhead will be $3n$, one `ptr_id`, one base, and one bound value for every pointer.

Wang et al. [101] in their work RetTag propose using a *Pointer Authentication Code*, or PAC, to prevent ROP attacks. Inspired by the message authentication code or MAC, a PAC is a hashed and encrypted chain of return addresses (encrypted hash values of return addresses in a nested function call). The idea is as follows: the PAC of a return address is calculated upon a function call. Once the function is done, the PAC of the actual return address (where the control flow is being diverted) is calculated and matched with the previously calculated PAC.

Figure 33 shows the generation of a PAC for a return address and its corresponding storage location. The PAC is generated using the virtual address, a 128-bit key (called the RAA or return address authentication key), and a 64-bit modifier to protect against replay attacks. The RAA key is fixed and is stored in a secure area on the chip. The authors augmented the RISC-V ISA with dedicated instructions to generate and validate PACs to make the whole process efficient. Specifically, they added two new instructions: `pac`, which calculates the PAC upon a function call, and `aut`, which validates the PAC when the function returns. Figure 34 shows the working of RetTag [101]. The authors use the intermediate representation, or IR, of the source code to plugin the new instructions and then execute them on an ISA-extended processor.

Similar work in this area was done by Austin et al. [11] in developing Morpheus-II, a RISC-V based processor that has additional instructions to support always-on encryption for code and pointers.

7 System-based defense methods

In the last section, we discussed different ways to protect the runtime structures of an application using hardware-assisted defense mechanisms. Those methods rely on either direct or indirect access to either the source code or the binary of the application. The defense mechanisms are then added to the application (either to the source code or to the binary).

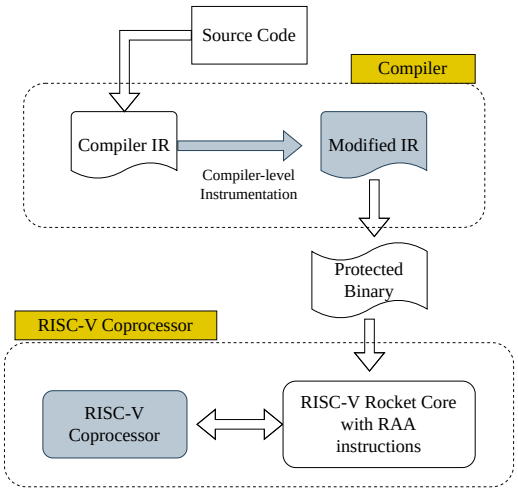


Figure 34. Overview of RetTag [101]. Adapted from [101]

However, it might be the case that a defense method cannot directly modify or monitor the state of the internal structures of an application. A typical scenario can be in a cloud-based setting where the user wants to use a stock version of an operating system or when the application is to be executed in a TEE environment.

An attacker analyzes every aspect of a binary's execution, and hence defense methods need to do the same. In situations where the defense methods cannot modify the source code or the binary, they rely on the behavior of the application rather than the state of its internal data structures and memory regions. An application performs many operations during its execution life cycle that can be quantifiably measured using instruction traces, dTLB events, iTLB events, cache events, branches taken, execution cycles, stall cycles, etc. However, collecting these statistics using software-based collection methods will incur a high performance overhead, due to constant interrupts, context switches, and cache pollution. Modern hardware provides an efficient way to access this information without incurring a significant performance overhead by using mechanisms such as Intel processor trace, indirect-branch tracking, and hardware performance counters.

7.1 Control flow verification

The CFI property of an application implies that it should follow the *developer-intended* control path during execution. Any deviation from this path is deemed as a violation of the CFI property. Now, the deviation may also occur due to unforeseen bugs in the code. In those cases, the application will just crash without leaking any data. However, if an attacker forces the binary to take a completely different control path (that was not present in the developer-intended paths) with a malicious intent of leaking sensitive information, then this action gets classified as a CFI attack.

Table 6. The *system-based* class defense methods.

Defense method	Hardware feature	Application attribute	Perf.	Attack model	Method
NumChecker [98]	PMU	Program behavior	2.8%	Rootkit	Behavior comparison
ROPDetect [68]	PMU	Program behavior	–	ROP	Machine learning
PathArmor [95]	LBR	Branches	8.5%	CRA	Branch Validation
LockInsec [10]	Novel HW	Branch instructions	0.5%	Data-attacks	Edge-caching
Remote [97]	PMU	Program behavior	4.42% – 5.92%	CFI	Compression and remote analysis
BBCFI [29]	Novel HW	Instructions	< 1%	CFI	Basic blocks
Griffin [38]	IPT	Pointers	11%	ROP and JOP	CFI enforcement
NoJumpBB [46]	LBR with Novel HW	Basic blocks	0.13%	ROP and JOP	Basic blocks
FlowGuard [66]	IPT	Trace	4%	CFI	CFG
HeNet [21]	IPT	Trace	–	Malware detection	Deep learning
DeepCheck [107]	IPT	Trace	–	CRA	CFG using deep learning

The control paths in an application can be represented using a control flow graph or CFG (see Figure 1). In a CFG, the vertices are the functions in the application, and a directed edge between two vertices indicates a function call from one function to another. A CFG can also be constructed at an even finer granularity by using basic blocks as nodes instead of functions [29]. We use a function-based CFG here for the sake of discussion. A complete CFG of an application will have all the developer-intended paths in it. However, creating a complete CFG is a non-trivial process [29]. This is because of the dynamic nature of the applications. The direct branches in the binary of an application can be factored in easily while creating the CFG; however, the targets of indirect branches are resolved at runtime, and there is no easy way to resolve them statically. Hence, researchers have used traces of applications to create the CFG [29]. However, this also does not cover every possible path as that will require the execution of the application with every possible input in every possible environment – a difficult task [10]. Hence, defense mechanisms rely on an approximated version of the CFG to implement defense mechanisms.

Arthur et al. [10] aim to solve the core issues that enable CFI attacks on an application: indirect branches. An attacker overwrites the return addresses of the target in memory and then misdirects the control flow to the overwritten addresses instead of the developer-intended targets. The authors first use the source code to generate a CFG and then replace all the indirect branches (including return instructions) with a set of

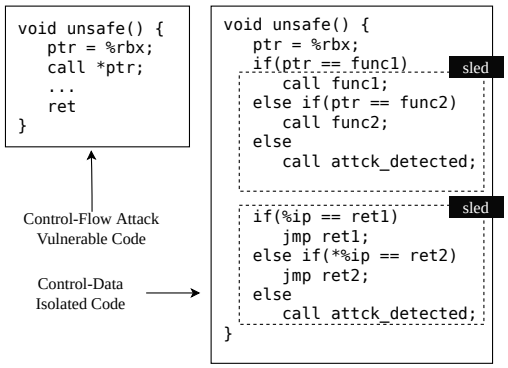


Figure 35. Replacing indirect branches with direct branches [10]. Adapted from [10]

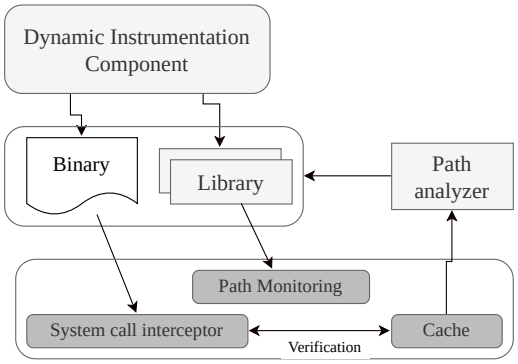


Figure 36. Architecture of PathArmor [95]. Adapted from [95]

compares and direct jumps (see Figure 35). Subsequently, the authors argue that doing this for every indirect instruction in the binary will impact the performance considerably. Hence, they introduce a hardware structure called the *edge cache* that stores the program counter and the target of all the valid indirect branches. While executing an indirect branch, if the edge cache contains the mapping, the jump is deemed valid and allowed. If it is not in the cache, the jump is validated against all the possible valid targets using the CFG. If it is valid, an entry is created in the edge cache, and the jump is allowed.

van der Veen et al. [95], in their work *PathArmor*, propose a context-sensitive CFI where they not only check if a control flow (forward or backward) is valid or not but also check whether it is valid in the current *context*. A similar approach was used to trim down the binary in [40]. The authors point out three challenges in dynamically validating the branches of an application: how to track the branches taken, what is required to validate those branches, and when to validate those branches? Figure 36 shows the architecture of PathArmor. For the first challenge, they rely on Intel's Last Branch Record or LBR registers [56] to track the branches taken by

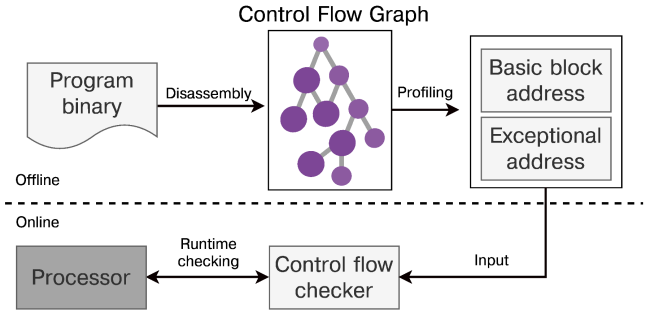


Figure 37. Architecture of BB-CFI [29]. Adapted from [29]

an application. They write a kernel module that interacts with the application and provides access to the values stored in the LBR registers (these registers can only be accessed in Ring 0). For the second challenge, the authors acknowledge the fact that calculating a CFG of an entire application can lead to *path explosion*; hence, it is not practical. They propose an on-demand, constraint-driven, context-sensitive static analysis over a normalized CFG representation. The analysis is on-demand because the validation is only done when the application wants to take a particular branch, and it is constraint-driven because it only checks for branching within an application. It does not consider branches within the dynamically-linked libraries. It is context-sensitive because it uses the LBR to look at the immediate history of the branches taken to validate the current branching sequence. Finally, a normalized CFG is used because it merges multiple paths from one function to another into a single path. PathArmor also keeps a cache of some of the recently validated paths in order to improve the performance. The cache is maintained as a hash table.

Das et al. [29] argue that a CFG constructed using functions is not granular enough to enforce the control flow policies of an application. Hence, they propose to use basic blocks and call the scheme basic block CFI or BB-CFI. The authors argue that basic blocks are a natural point that can be used to enforce control policies. Note that no jump is allowed from within a basic block. Execution flow enters a basic block from the first instruction and leaves at the last instruction. Figure 37 provides an overview of the working of BB-CFI. They disassemble a binary to obtain a CFG. However, this is incomplete as the addresses of the indirect branches are missing. In order to fill this gap, the authors profile the binary using different inputs and generate a list of a valid set of target addresses for indirect branches. This is an offline setup and is assumed to be free of attacks.

Once a set of valid addresses is generated, it is used during the execution of the application to validate indirect branches: ❶ a *call* instruction must target the first instruction of a function. ❷ For a *ret* instruction, the authors maintain a *return address stack* or RAS. During a *call* instruction, the address

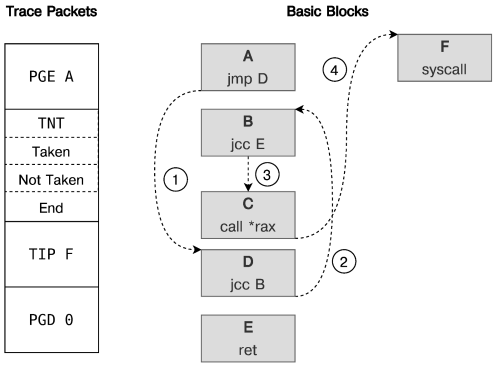


Figure 38. Tracing control flow in Griffin [38]. Adapted from [38]

of the next instruction (which is the first instruction in the immediately succeeding basic block that contained the call instruction) is pushed. While returning, the *ret* instruction's target address is compared with the popped value from the RAS stack. This is also known as *call-ret* pairing.

❸ A *jmp* instruction allows a jump to the first instruction of any basic block in the application. The authors also discuss few exceptional cases such as when a *call-ret* pair is replaced by a *call-jmp* pair (for example in *longjmp()*. Also, exception handlers need to be properly handled (they can jump to an arbitrary location). In such cases also, the *call-ret* paradigm will not be followed. This will result in false positives as the RAS will not contain invalid return addresses. The authors provide certain mechanisms in order to remove these false positives. This is done by allowing *ret* instructions to target addresses in the RAS and *jmp* instructions to target exception handling addresses. They claim that this technique has an accuracy of 100%. The implementation on an FPGA board incurs a performance overhead of less than 1%.

7.2 Application execution trace

The trace of a process's execution is an ordered set of executed instructions. It contains every detail of what the process did (or is doing) during the execution. It can be analyzed, either online or offline, to detect any CFI violations by comparing the trace with another trace that is known to be from a valid execution. However, as a process may execute millions of instructions, this set can be huge, and processing it online is not a trivial task because it requires instrumenting every instruction of the application and then processing it. It will cause a significant performance overhead if done in software. Hardware support such as Intel processor trace or IPT ensures that the trace collection can be done with minimal overhead.

Ge et al. [38], in their work *Griffin*, use IPT for online enforcement of CFI policies on the forward edges (function calls) and a shadow-stack on the backward edges (for return pointers). The authors use the trace data generated by the IPT to ensure CFI policies on the fly. However, doing so is not trivial. The data generated by IPT is meant for debugging purposes, and hence, is encoded to save space and processing time. Furthermore, it also does not record information that can be generated from other data (e.g., the source address of an indirect branch). There are no issues if the processing is done offline; however, online processing is difficult. IPT can capture user-level and kernel-level traces; however, the authors focus their attention on the user-level instruction trace. The table below lists some of the *trace packets* generated by the IPT, its usage, and its size.

Packet	Usage	Size (B)
PGE	Packet Generation Enable packets provide the PC at which the tracing begins	≤ 8
PGD	Packet Generation Disable packets mark the end of tracing	≤ 8
TNT	Taken/Not-Taken packets indicate the direction of conditional branches	1
TIP	Target IP packets provide the target for some control-flow transfers	≤ 8

Griffin supports two kinds of CFI policies: coarse-grained and fine-grained. The coarse-grained policy only checks if the target address of an indirect branch is a valid address or not. For this purpose, it maintains a page at a constant offset with respect to every code page; 1 represents a valid jump address and 0 means an invalid jump address. In the fine-grained policy, Griffin checks whether the source and target address pair is valid. For this, it maintains a bit-wise matrix with rows representing the source address and columns representing the target address. This matrix is dynamically grown when a shared library is loaded. Here, Griffin needs to decode the trace packets since they do not contain the source address. To summarize, Griffin requires the trace packets and a disassembled binary to reconstruct the CFG and validate the jump.

Figure 38 shows a sample IPT trace and the corresponding execution flow. The first PGE packet indicates that the execution begins at basic block *A*. Then it takes a direct jump to block *D* (no trace packets for a direct jump), then the TNE packet indicates whether the conditional jump in block *D* is taken or not. It is taken, so the control passes to block *B*. The unconditional branch in block *B* is not taken, and the control falls to block *C*. Then, the control is transferred to block *F* with a *call* instruction as recorded by the IPT packet. The trace ends with a PGD packet. The authors propose to use idle cores on a modern system to reduce the overhead. They report a performance overhead of $\approx 11\%$.

Liu et al. [66] improve the runtime overhead involved in using traces from IPT in their work *FlowGuard*. IPT generates traces in an encoded format, which needs to be decoded later to get the complete trace. FlowGuard addresses this challenge by compressing and generating the control flow graph in the same format as the encoded traces generated by IPT. This makes the comparison of traces feasible. During the process

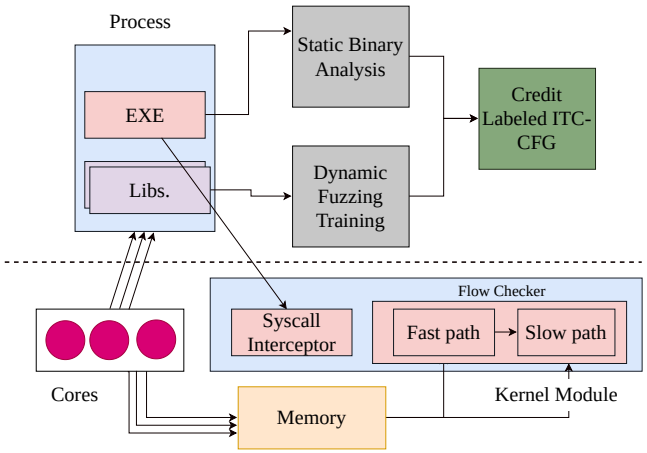


Figure 39. Architecture of FlowGuard [66]. Adapted from [66]

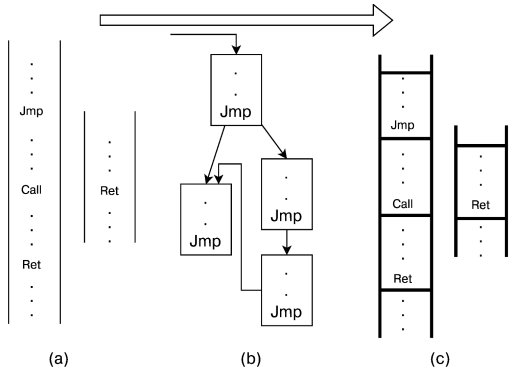


Figure 40. Basic block boundaries [46]. Adapted from [46]

of offline generation, it automatically generates inputs for the application and marks the edges in the graph with high or low credits (see Figure 39). A *high-credit* branch can be verified quickly while checking, while a *low-credit* branch is checked more thoroughly – as these are rare. They report a runtime overhead of 4%.

He et al. [46] leverage the trace from IPT to mark an indirect jump as either valid or invalid. Note that entries and exits can only happen at the boundaries of basic blocks. There are no entry or exit points in the middle of a basic block. They aspire to detect an entry into the middle of a basic block, which causes CFI checks to fail (see Figure 40).

7.3 Trusted services

Hardware features can also be used to implement trusted services in the operating system; they can be used by other trusted or untrusted applications. For example, Intel SGX allows for the secure execution of an application in a trusted sandbox on the CPU. However, the application cannot access

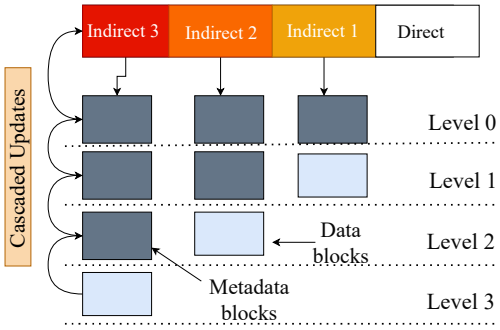


Figure 41. Cascaded updates in an inode-based design [58]. Adapted from [58]

the file system as that falls under the purview of the operating system – an untrusted entity from the point of view of SGX.

Kumar et al. [58] in their work, SecureFS, propose a secure file system that can be used with Intel SGX without modifying the application. The authors start by pointing out that a secure file system must encrypt the file blocks with a different key and also check their integrity before sending them to the underlying, untrusted file system. As the data is encrypted, an attack cannot read the content and also cannot modify the content since it will fail the integrity checks. The key used to encrypt the file block is generated anew for every encryption. Doing so prevents a replay attack on the encrypted file blocks, where an attacker can replay an old encrypted block and can force an application to either leak sensitive information or break internal security (like license checks). The key is typically stored with the metadata of the file system ([33, 58]). The metadata state is maintained in the secure memory provided by SGX. Prior work in this area had used an i-node-based file system to store the metadata. This is the standard format used in modern file systems; it is prevalent because of its flexibility and ability to support large file systems. However, a secure application generally does not require such a large file system. Furthermore, the tree-like structure of an inode-based file system creates a cascading effect of key updates upon an encryption (encrypting a child will force an update of its parent to store the key, which in turn needs to be re-encrypted, and so on till the root of the tree). To ameliorate the situation the authors proposed a novel solution based on FAT tables.

Gregor et al. [43] used SGX to implement Palaemon, which acts as a secret provisioning system in an untrusted environment. The first and foremost question is, how do you validate an application's request for a secret? Palaemon relies on the local attestation process of Intel SGX. Using cryptographic primitives, SGX guarantees that the requesting application is running on the same machine and has not been tampered with.

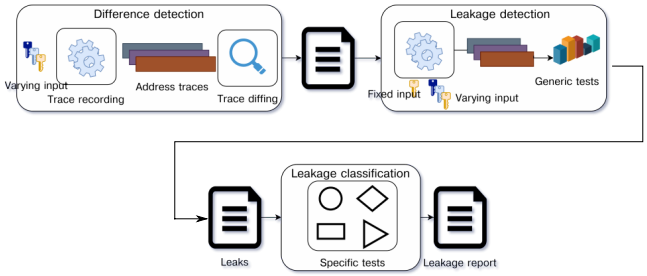


Figure 42. Working of DATA [103]. Adapted from [103]

7.4 Machine/Deep learning approaches

Online methods to detect a CFI violation using an application trace prevent an attack from doing any significant damage. However, the overhead of processing the additional information is still there. Another approach is to follow an optimistic approach by allowing the execution to go through and just log all the generated traces. This allows the defense methods to find CFI violations using sophisticated methods, such as machine learning, deep learning, and formal verification; these would have led to a significant performance overhead in an online mode.

Biondi et al. [15] argue that the trace generated for a process is huge. They propose to first shrink its size by selecting security-relevant parameters and then apply Markov chain based analysis to assess if the application's behavior diverges from its ideal or safe execution path. They perform the analysis on a tool called Quail [14], which can analyze a process for its security properties using the compressed form of the collected trace.

Weiser et al. [103] propose to detect address leaks in a binary by analyzing its traces. They propose a three-stage detection mechanism (see Figure 42). First, trace data is collected by running a process multiple times. In the second phase, different kinds of leaks are detected. Finally, in the last stage, all the leaks are identified and categorized based on the type of information they can leak.

Chen et al. [21] propose to use deep learning methods on the trace collected via IPT. Deep learning methods are ideal for processing large amounts of data, such as the trace of a process. Using their method, the collected trace is converted into an image. The image is then segmented to find anomalies in the execution. It has two levels of operation: the lower model is a per-application behavioral model, which is trained via transfer learning on a time-series of images generated from the control flow trace of an execution, and a higher-level ensemble model, which collects all the low-level model details to detect an attack. Figure 43 shows the working of HeNet. To train a neural network, the authors use a transfer learning approach, and transfer learned data from state of the art deep neural networks such as Inception [91] and VGG [86].

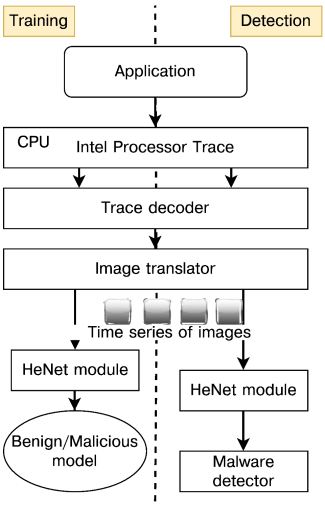


Figure 43. Overview of HeNet [21]. Adapted from [21]

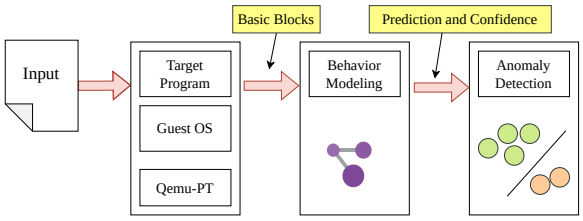


Figure 44. Overview of Barnum [104]. Adapted from [104]

Working on similar lines, Yagemann et al. [104] argue that the size of the traces generated during an application execution is very large, and all of it is not useful in determining whether there is an attack on the control flow integrity of the application. The authors argue for an offline anomaly detection method over a classification-based system since any divergence from the normal control flow can be termed an attack on the CFI, irrespective of the class of attack. The authors propose to use the trace of a process since it requires no modification to the application or any type of additional instrumentation support. The authors leverage Intel Processor Trace, or IPT, to efficiently collect the traces of an application with a minimal performance overhead.

As shown in Figure 44, the proposed model called Barnum [104] has three layers: ❶ trace collection, ❷ control flow modeling, and ❸ anomaly detection. The authors use IPT for the first step. To generate a model, the authors use a Long Short Term Memory or LSTM network, which consists of an embedding layer, three LSTM layers, and a final dense neural network (see Figure 45). Their model has an accuracy of 98.1%, a precision of 100%, and a recall of 97.6%.

Zhang et al. [107] in their work DeepCheck, propose using a deep learning method to detect code reuse attacks on an application. They create a model for the control flow of a process based on a coarse-grained control flow graph of

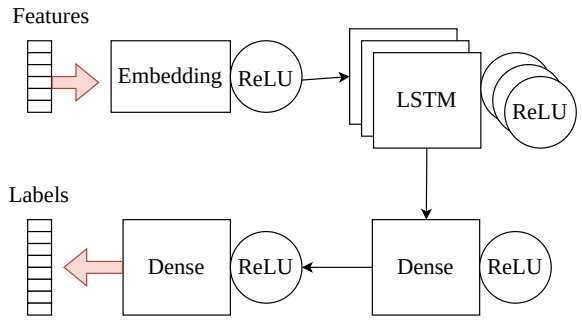


Figure 45. Overview of the Barun LSTM model [104]. Adapted from [104]

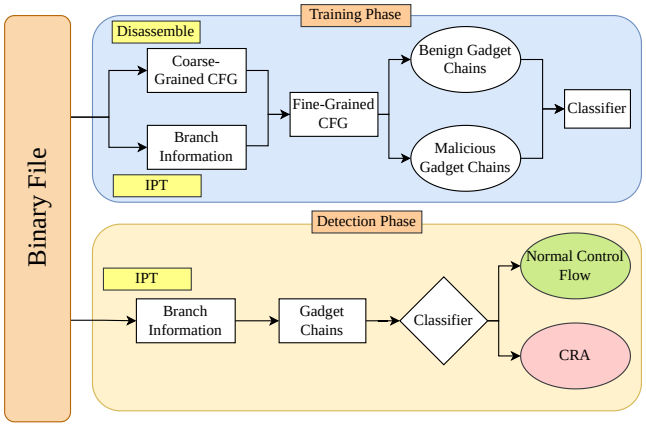


Figure 46. Working of DeepCheck [107]. Adapted from [107]

the application using the disassembled instructions from the binary. The nodes in the generated CFG represent the gadgets in the binary, and an edge between two gadgets indicates a path in the control flow graph. As shown in Figure 46, this coarse-grained CFG is combined with the dynamic trace information (using IPT) to obtain a fine-grained CFG. .

This fine-grained model is sent as an input to a six-layer deep neural network (DNN): one input layer, four hidden layers, and one output layer. The hidden layers consist of 1024, 512, 128, and 32 nodes, respectively. A rectified linear unit (ReLU) is used as the activation function. This model is then used on the traces collected from other executions to classify them either as a normal execution flow or as a *code reuse attack*. The authors report an accuracy of up to 99.4% for a set of real-world workloads such as Adobe Flash, Nginx, and Firefox.

7.5 Hardware performance counters

Modern systems have hardware performance counters built into them. These counters capture fine-grained statistics of either a process's execution or for the complete system (see Section 2.3 for more details).

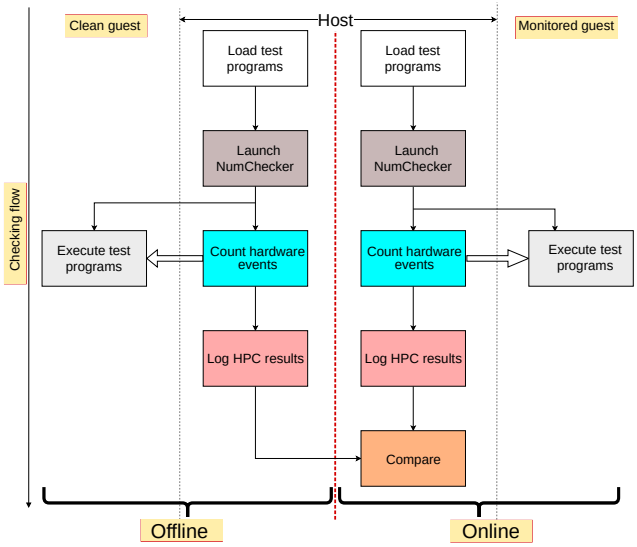


Figure 47. Working of NumChecker [98]. Adapted from [98]

Wang and Karri [98] propose to use these hardware counters to detect a kernel rootkit. A kernel rootkit attacks the kernel of an operating system by either modifying the non-control data in the kernel's data structures or hijacking the control flow of the kernel. The authors focus on the problem of control flow. A popular example of an attack is where a kernel rootkit replaces the system call table and diverts the system calls to its malicious code. To detect such a kind of attack, the authors propose to validate a system call by monitoring the hardware counters during their execution. They argue that each valid system call is associated with a certain number of hardware events such as total instructions, branches, returns, floating-point operations, etc. A malicious implementation of the same system call will have a different signature.

As shown in Figure 47, in the offline stage, the authors use valid system calls and create "clean" profiles (number of hardware events) for those calls. The input parameters are varied to capture different paths taken by the OS to service a particular system call. Later, to authenticate an OS, a system call is made using known input parameters and the hardware counters are monitored. This collected counter data is compared with the original profile. The execution is marked valid if the counters are within a threshold of the "clean-copy" data (online-phase).

Working on a similar line, Lu and Hansen [68] use the insight that during a return oriented programming attack, the number of `ret` instructions executed will be high than when compared to the normal execution of an application. This is because in an ROP attack, the attacker overwrites the target address of a `ret` instruction. This is done more than once to form a chain of gadgets to execute a piece of malicious logic, which results in a high number of `ret` instructions vis-a-vis

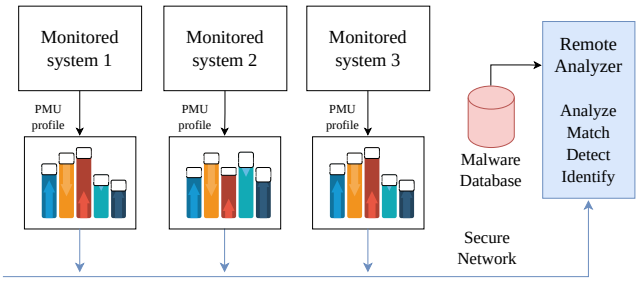


Figure 48. Monitor systems in an HPC setting and validating it remotely [97]. Adapted from [97]

normal execution. Apart from this, during the execution of an application, the retired *ret* instructions are generally associated with a corresponding*call* instruction. This `call-ret` pair is typically missing in an ROP attack. The authors use these facts to propose unsupervised machine learning based techniques to detect ROP attacks in an application.

Wang et al. [97] argue that in an HPC setting, the low-level counters are readily accessible and can be used to profile or validate a process. However, after collecting the data, it is not necessary to analyze the data on the same system, which may degrade the performance of the original application. They compress the data and send it to a remote machine for processing (see Figure 48). This ensures that vital information required to detect the presence of malware in the system is not lost during compression. To ensure this, they use a technique called "compressive sensing", which is a standard technique for efficiently acquiring, compressing, transmitting, and reconstructing a signal. The technique leverages the sparsity in the data to recover it efficiently.

8 Research directions and future work

In this section, we discuss possible research directions for the defense methods based on the material presented in the paper.

8.1 Bounding the problem

The key to maintaining the CFI is that the binary should not do what it is not supposed to. A modern application contains millions of instructions and there are billions of possible control paths. Statically generating all these control paths is not feasible as some of those paths are dynamic in nature. Furthermore, facilities such as shared libraries and JIT code make the situation even worse. Arguing for a dynamic approach to generate all the possible control paths is even harder as it requires feeding the application all possible inputs and varying the system state to take a large number of values. Doing so is infeasible, and hence, modern defense methods rely on an approximation of the space of control paths and work on probabilistic defense mechanisms.

The key issue here is that defense methods are trying to be too generic and cater to every possible input combination. Real-world benchmarks have distinct signatures in terms of their execution patterns. Defense methods can leverage this behavior to limit the control paths an application can take while executing. Developing a defense method for a particular application or for a set of applications that show similar execution characteristics has multiple benefits: as stated before, it bounds the sample space of the control paths, the defense method knows the sensitive parts of the application, all the attack methods targeting it can be thoroughly investigated (as it is also bounded now), and the defense method can be configured to be highly optimized for the set of applications.

8.2 Configuring defense methods

Modern applications are complicated, and the environment they are executed on (a bare metal, a container, a VM, on the edge) is even more so. Furthermore, it is rarely the case that an application has complete access to all the hardware resources. It generally shares these resources with other applications over which it has no control whatsoever. All it can do is trust the OS (and any other entity between it and the hardware) to do a fair allocation of the resources.

While enabling this, an application has to have many interactions with the OS, the hardware, and any other entity managing the resources. These interactions are not trivial and often form the basis for an attack on the CFI and also other forms of attacks (side-channel attacks). The users of an application do not possess the technical expertise or patience to go through all these interactions and then tune the defense mechanisms. The ideal scenario is that a user lists down the security requirement, preferably in simple and clear terms, and then the OS (or some other entity) ensures that those security requirements hold during the execution. Relying on users to first correctly pick a defense mechanism, then correctly tune it as per their requirements may either lead to very strict rules (severely affecting the performance) or very relaxed rules (severely affecting the security of the application). Hence, there is a need for defense solutions that are aware of users' requirements and can also auto-tune themselves.

8.3 What about the operating system?

In recent times, computing trends have seen a significant shift from local machines/servers to commercial data centers, colloquially known as "cloud computing". This has enabled small/medium sized users to scale up their applications to millions of customers without having to worry about managing and maintaining the costly servers. This is a highly cost-effective way as the users do not have to worry about the storage space for the servers, hire people to manage the servers, ensure a consistent power supply, cooling unit, and network connections.

However, the security of the data and the code on a remote, untrusted server is also a big concern for users. A malicious server administrator can access all the code and data stored on the remote servers. Hence, there is a need to provide secure computing in a cloud setting. Attacks can emanate not only from a malicious server administrator but from any other malicious privileged entity such as the operating system or the hypervisor.

A TEE or a trusted execution environment is a promising solution in such cases. A TEE ensures the security of an executing application in terms of its confidentiality, integrity, and freshness. These security properties are guaranteed by the hardware and cannot be tampered with even by privileged entities such as the OS and hypervisors. Intel SGX is a TEE solution from Intel that is present in most of the commercially available CPUs. However, as of now, Intel SGX is severely restricted. For example, it only offers a limited amount of secure memory of 128 MB (extendable to 256 MB on some processors), does not allow direct system calls or memory sharing and requires a large amount of porting effort by the developers. Researchers in industry and academia have proposed many solutions that either alleviate some of these challenges or increase the performance of applications executing with SGX. Also, there have been efforts to execute unmodified binaries on SGX. Graphene-SGX [19] is a library operating system that acts as a shim layer for an application; it executes system calls on an application's behalf. However, it incurs significant performance overheads. Hence, there is a need to make TEE solutions more practical and transparent to use.

8.4 Future of computing and security needs

In the past few years, the computing world has seen a paradigm shift where computing has moved from desktops and servers to big data centers. This service is known as IaaS, or infrastructure-as-a-service. Here, the users pay for the infrastructure hosted at a remote location. It is a game-changing move for small industries and developers, who could not have found it affordable to buy and maintain servers and networking equipment. Large industries eventually adopted this new model, which led to the birth of another paradigm, SaaS (software as a service). Microsoft Office 365 is one of the prime examples of this, where developers moved the traditional software products to a browser, and the complete service moved to a subscription-based model instead of a one-time payment. Another shift is happening in the data center, where micro-services-based computing is becoming more popular than monolithic designs. This facilitates easy maintenance of the complete stack, better load-balancing, and seamless integration of different components that have possibly been coded using different programming paradigms.

However, along with the ease of development, the security of these services is also important because, in this case, breaching the security of one service may affect the entire network's security. Hence, there is a dire need to ensure the CFI of these applications or micro-services executing on remote, untrusted machines. Along with the usual challenges of enforcing security policies, these services bring another set of challenges along with them. These microservices are sensitive to latency, and a performance bottleneck in any one of the services can cascade to the whole stack (other dependent services). Hence, there will be a requirement for high-speed solutions for such services in the near term. A low network overhead is preferred as the network traffic is already too high because of communication between the connected components. Any increased traffic here will impact the performance of the entire network.

All these requirements point towards hardware-assisted defense mechanisms due to their enhanced security guarantees (as they are directly embedded in the hardware) and their performance is superior as compared to software-based methods.

9 Conclusion

The number of attacks on the control flow integrity or CFI of an application has significantly gone up in the past few years. Every aspect of an application has been explored and exploited by the attacker to violate the CFI property of an application. Similarly, researchers have also explored a large number of avenues to prevent these attacks. Lately, they have adopted a hybrid approach towards providing defense against such attacks by leveraging the security and speed of the hardware and the flexibility of the software.

In this paper, we present a novel taxonomy for hardware-assisted defense mechanisms based on their impact. We believe this presents a clear depiction of the existing defense methods and also provides clear guidance to select a particular category of defense methods based on the user's constraints and requirements (*binary-based* if we have access to the source code, *process-based* if we have access to the operating system's internals, and *system-based* if we have access to the system). Furthermore, our taxonomy can also aid a security researcher by providing different ways to protect a feature of an application. For example, the stack of an application can be prevented by either adding additional memory checks or CFI-related instructions directly in the binary (*binary-based*), by using a shadow stack (*process-based*), or by monitoring the character of operating system and micro-architectural events (*system-based*).

References

[1] [n.d.]. Advanced Vector Extensions - Wikipedia. https://en.wikipedia.org/wiki/Advanced_Vector_Extensions. (Accessed on 03/25/2022).

[2] [n.d.]. hiie-report-s16-17.pdf. https://courses.cs.ut.ee/MTAT.07.022/2017_spring/uploads/Main/hiie-report-s16-17.pdf. (Accessed on 10/23/2019).

[3] [n.d.]. Intel® Software Guard Extensions | Intel® Software. https://software.intel.com/en-us/sgx. (Accessed on 12/14/2019).

[4] [n.d.]. The Kernel Address Sanitizer (KASAN) — The Linux Kernel documentation. https://www.kernel.org/doc/html/v4.12/dev-tools/kasan.html. (Accessed on 12/04/2021).

[5] [n.d.]. Physical unclonable function - Wikipedia. https://en.wikipedia.org/wiki/Physical_unclonable_function. (Accessed on 12/02/2019).

[6] [n.d.]. TrustZone – Arm Developer. https://developer.arm.com/ip-products/security-ip/trustzone. (Accessed on 12/14/2019).

[7] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (Alexandria, VA, USA) *(CCS '05)*. Association for Computing Machinery, New York, NY, USA, 340–353. https://doi.org/10.1145/1102120.1102165

[8] Ioannis Agadakos, Nicholas Demarinis, Di Jin, Kent Williams-King, Jearson Alfajardo, Benjamin Shteinfeld, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2020. Large-Scale Debloating of Binary Shared Libraries. *Digital Threats: Research and Practice* 1, 4, Article 19 (dec 2020), 28 pages. https://doi.org/10.1145/3414997

[9] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Mark L Stillwell, David Goltzsche, David Eyers, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. *Osdi*, 689–704.

[10] William Arthur, Sahil Madeka, Reetuparna Das, and Todd Austin. 2015. Locking Down Insecure Indirection with Hardware-based Control-data Isolation. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. ACM, New York, NY, USA, 115–127. https://doi.org/10.1145/2830772.2830801

[11] Todd Austin, Austin Harris, Tarunesh Verma, Shijia Wei, Alex Kisil, Misiker Aga, Valeria Bertacco, Baris Kasikci, and Mohit Tiwari. 2021. Morpheus II: A RISC-V Security Extension for Protecting Vulnerable Software and Hardware. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–18. https://doi.org/10.1109/HCS52781.2021.9567000

[12] Denis Bakhvalov. 2018. PMU counters and profiling basics. | Easyperf. https://easyperf.net/blog/2018/06/01/PMU-counters-and-profiling-basics. (Accessed on 12/18/2021).

[13] Mario Barbareschi, Pierpaolo Bagnasco, and Antonino Mazzeo. 2015. Authenticating IoT Devices with Physically Unclonable Functions Models. *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)* (2015), 563–567.

[14] Fabrizio Biondi, Axel Legay, Louis-Marie Traonouez, and Andrzej Wasowski. 2013. QUAIL: A Quantitative Security Analyzer for Imperative Code, Vol. 8044. 702–707. https://doi.org/10.1007/978-3-642-39799-8_49

[15] F. Biondi, Jean Quilbeuf, and A. Legay. 2014. Information Leakage by Trace Analysis in QUAIL.

[16] Nathan Burow, X. Zhang, and M. Payer. 2019. SoK: Shining Light on Shadow Stacks. *2019 IEEE Symposium on Security and Privacy (SP)* (2019), 985–999.

[17] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 161–176. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini

[18] Buddhika Chamith, Xiaozhu Meng, and Ryan Newton. 2020. ShadowGuard : Optimizing the Policy and Mechanism of Shadow Stack Instrumentation using Binary Static Analysis. arXiv:2002.07748 [cs.CR]

[19] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference*.

[20] Kejun Chen, Orlando Arias, Qingxu Deng, Daniela Oliveira, Xiaolong Guo, and Yier Jin. 2022. FineDIFT: Fine-Grained Dynamic Information Flow Tracking for Data-Flow Integrity Using Coprocessor. *IEEE Transactions on Information Forensics and Security* 17 (2022), 559–573.

[21] Li Chen, Salmin Sultana, and Ravi Sahita. 2018. HeNet: A Deep Learning Approach on Intel® Processor Trace for Effective Exploit Detection. *2018 IEEE Security and Privacy Workshops (SPW)* (2018), 109–115.

[22] George Christou, Giorgos Vasiliadis, Vassilis Papaefstathiou, Antonis Papadogiannakis, and Sotiris Ioannidis. 2020. On Architectural Support for Instruction Set Randomization. *ACM Trans. Archit. Code Optim.* 17, 4, Article 36 (nov 2020), 26 pages. https://doi.org/10.1145/3419841

[23] Context. [n.d.]. Microsoft Word - Return-to-libc.txt. https://css.csail.mit.edu/6.858/2017/readings/return-to-libc.pdf. (Accessed on 12/05/2021).

[24] Luigi Coppolino, Salvatore D'Antonio, Giovanni Mazzeo, and Luigi Romano. 2019. A comprehensive survey of hardware-assisted security: From the edge to the cloud. *Internet Things* 6 (2019). https://doi.org/10.1016/j.iot.2019.100055

[25] Jonathan Corbet. 2017. ARM pointer authentication [LWN.net]. https://lwn.net/Articles/718888/. (Accessed on 12/17/2021).

[26] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86.

[27] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86.

[28] Thurston H. Y. Dang, Petros Maniatis, and David A. Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (2015).

[29] Sanjeev Das, Wei Zhang, and Yang Liu. 2016. A Fine-Grained Control Flow Integrity Approach Against Runtime Memory Attacks for Embedded Systems. *IEEE Trans. Very Large Scale Integr. Syst.* 24, 11 (Nov. 2016), 3193–3207. https://doi.org/10.1109/TVLSI.2016.2548561

[30] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. 2015. HAFIX: Hardware-assisted Flow Integrity Extension. In *Proceedings of the 52Nd Annual Design Automation Conference* (San Francisco, California) *(DAC '15)*. ACM, New York, NY, USA, Article 74, 6 pages. https://doi.org/10.1145/2744769.2744847

[31] David J. Day, Zhengxu Zhao, and Minhua Ma. 2010. Detecting Return-to-libc Buffer Overflow Attacks Using Network Intrusion Detection Systems. *2010 Fourth International Conference on Digital Society* (2010), 172–177.

[32] Ruan de Clercq and Ingrid Verbauwhede. 2017. A survey of Hardware-based Control Flow Integrity (CFI). *CoRR* abs/1706.07257 (2017). arXiv:1706.07257 http://arxiv.org/abs/1706.07257

[33] Judicael B. Djoko, Jack Lange, and Adam J. Lee. 2019. NeXUS: Practical and Secure Access Control on Untrusted Storage Platforms using Client-Side SGX. *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2019), 401–413.

[34] S. Du, Hui Shu, F. Kang, Xiaobing Xiong, and Zheng Wang. 2017. Hardware-based instruction set randomization against code injection attacks. *2017 3rd IEEE International Conference on Computer and Communications (ICCC)* (2017), 1426–1433.

[35] Abbas A. Fairouz, Monther Abusultan, Viacheslav V. Fedorov, and Sunil P. Khatri. 2021. Hardware Acceleration of Hash Operations in Modern Microprocessors. *IEEE Trans. Comput.* 70, 9 (2021), 1412–1426. https://doi.org/10.1109/TC.2020.3010855

[36] Daichi Fukui, Mamoru Shimaoka, Hiroki Mikami, Dominic Hillenbrand, Hideo Yamamoto, Keiji Kimura, and Hironori Kasahara. 2015. Annotatable Systrace: An Extended Linux Ftrace for Tracing a Parallelized Program. In *Proceedings of the 2nd International Workshop on Software Engineering for Parallel Systems* (Pittsburgh, PA, USA) *(SEPS 2015)*. Association for Computing Machinery, New York, NY, USA, 21–25. https://doi.org/10.1145/2837476.2837479

[37] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, M. T. Aga, Austin Harris, Zhixing Xu, Baris Kasikci, V. Bertacco, S. Malik, Mohit Tiwari, and T. Austin. 2019. Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn. *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019).

[38] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *ASPLOS*.

[39] Dimitris Geneiatakis, Georgios Portokalidis, Vasileios P. Kemerlis, and Angelos D. Keromytis. 2012. Adaptive Defenses for Commodity Software through Virtual Application Partitioning *(CCS '12)*. Association for Computing Machinery. https://doi.org/10.1145/2382196.2382214

[40] Masoud Ghaffarinia and Kevin W Hamlen. 2019. Binary Control-Flow Trimming. In *CCS '19*.

[41] Enes Goktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *2014 IEEE Symposium on Security and Privacy*. IEEE. https://doi.org/10.1109/sp.2014.43

[42] Brendan Gregg and Jim Mauro. 2011. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD* (1st ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.

[43] F. Gregor, W. Ozga, S. Vaucher, R. Pires, D. Le Quoc, S. Arnautov, A. Martin, V. Schiavoni, P. Felber, and C. Fetzer. 2020. Trust Management as a Service: Enabling Trusted Execution in the Face of Byzantine Stakeholders. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 502–514.

[44] Aisha Hasan, Ryan Riley, and Dmitry Ponomarev. 2020. Port or Shim? Stress Testing Application Performance on Intel SGX. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 123–133. https://doi.org/10.1109/IISWC50251.2020.00021

[45] Vikas Hassija, V. Chamola, V. Saxena, D. Jain, Pranav Goyal, and B. Sikdar. 2019. A Survey on IoT Security: Application Areas, Security Threats, and Solution Architectures. *IEEE Access* 7 (2019), 82721–82743.

[46] Wenjian He, Sanjeev Das, Wei Zhang, and Yang Liu. 2017. No-jump-into-basic-block: Enforce basic block CFI on the fly for real-world binaries. *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)* (2017), 1–6.

[47] Adrian Hoban. [n.d.]. Using Intel AES-NI to Significantly Improve IPSec Performance on Linux. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/aes-ipsec-performance-linux-paper.pdf. (Accessed on 03/25/2022).

[48] Konrad Hohentanner, Philipp Zieris, and Julian Horsch. 2022. PACSafe: Leveraging ARM Pointer Authentication for Memory Safety in C/C++. *ArXiv* abs/2202.08669 (2022).

[49] Catalin Hritcu. 2015. Micro-Policies: Formally Verified, Tag-Based Security Monitors. In *PLAS@ECOOP*.

[50] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. *2016 IEEE Symposium on Security and Privacy (SP)* (2016), 969–986.

[51] M. Husak, J. Komarkova, E. BouHarb, and Pavel Celeda. 2019. Survey of Attack Projection, Prediction, and Forecasting in Cyber Security. *IEEE Communications Surveys & Tutorials* 21 (2019), 640–660.

[52] Intel. 2017. Intel Processor Trace Tools | Intel® Software. https://software.intel.com/en-us/node/721535.

[53] Intel. 2019. SDK Intel Software Guard Extensions. https://software.intel.com/en-us/sgx/sdk. (Accessed on 10/25/2019).

[54] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* (2016).

[55] Deokjin Kim, Daehee Jang, Minjoon Park, Yunjong Jeong, Jonghwan Kim, Seokjin Choi, and Brent Byunghoon Kang. 2019. SGX-LEGO: Fine-grained SGX controlled-channel attack and its countermeasure. *Comput. Secur.* 82 (2019), 118–139.

[56] Andi Kleen. 2016. An introduction to last branch records [LWN.net]. https://lwn.net/Articles/680985/. (Accessed on 11/17/2018).

[57] S Kumar, D Moolchandani, T Ono, and S R Sarangi. 2019. F-LaaS: A Control-Flow-Attack Immune License-as-a-Service Model. In *2019 IEEE International Conference on Services Computing (SCC)*. 80–89. https://doi.org/10.1109/SCC.2019.00025

[58] Sandeep Kumar and Smruti R. Sarangi. 2021. SecureFS: A Secure File System for Intel SGX *(RAID '21)*. Association for Computing Machinery, New York, NY, USA, 91–102. https://doi.org/10.1145/3471621.3471840

[59] Nate Lawson. 2009. Side-Channel Attacks on Cryptographic Software. *IEEE Security and Privacy* 7, 6 (Nov. 2009), 65–68. https://doi.org/10.1109/MSP.2009.165

[60] Jinyong Lee, Ingoo Heo, Yongje Lee, and Yunheung Paek. 2015. Efficient Dynamic Information Flow Tracking on a Processor with Core Debug Interface. In *Proceedings of the 52nd Annual Design Automation Conference* (San Francisco, California) *(DAC '15)*. Association for Computing Machinery, New York, NY, USA, Article 79, 6 pages. https://doi.org/10.1145/2744769.2744830

[61] Yongsuk Lee and Gyungho Lee. 2019. HW-CDI: Hard-Wired Control Data Integrity. *IEEE Access* 7 (2019), 10811–10822.

[62] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. 2019. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 177–194. https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand

[63] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter R. Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *USENIX Annual Technical Conference*.

[64] Qiang Liu, Pan Li, Wentao Zhao, Wei Cai, Shui Yu, and Victor CM Leung. 2018. A survey on security threats and defensive techniques of machine learning: A data driven view. *IEEE access* 6 (2018), 12103–12117.

[65] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. Transparent and Efficient CFI Enforcement with Intel Processor Trace. *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2017), 529–540.

[66] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In *Proceedings - International Symposium on High-Performance Computer Architecture*. https://doi.org/10.1109/HPCA.2017.18

[67] Tao Lu. 2021. A Survey on RISC-V Security: Hardware and Architecture. *CoRR* abs/2107.04175 (2021). arXiv:2107.04175 https://arxiv.org/abs/2107.04175

[68] Yifan Lu and Christopher Hansen. 2015. ROPDetect : Detection of Code Reuse Attacks.

[69] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. ACM, New York, NY, USA, 941–951. https://doi.org/10.1145/2810103.2813676

[70] Marcela S. Melara, M. Freedman, and M. Bowman. 2019. EnclaveDom: Privilege Separation for Large-TCB Applications in Trusted Execution Environments. *ArXiv* abs/1907.13245 (2019).

[71] Arjun Menon, Subadra Murugan, Chester Rebeiro, Neel Gala, and Kamakoti Veezhinathan. 2017. Shakti-T: A RISC-V Processor with Light Weight Security Extensions. In *Proceedings of the Hardware and Architectural Support for Security and Privacy* (Toronto, ON, Canada) *(HASP '17)*. Association for Computing Machinery, New York, NY, USA, Article 2, 8 pages. https://doi.org/10.1145/3092627.3092629

[72] Ahmad Moghimi. 2017. Side-Channel Attacks on Intel SGX: How SGX Amplifies The Power of Cache Attack.

[73] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. 2015. Opaque Control-Flow Integrity. In *NDSS*.

[74] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2015. Everything You Want to Know About Pointer-Based Checking. In *SNAPL*.

[75] Nergal. 2001. The advanced return-into-lib(c) exploits. http://phrack.org/issues/58/4.html. (Accessed on 12/05/2021).

[76] J. Newsome and D. Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *NDSS*.

[77] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology* (San Jose, CA) *(CT-RSA'06)*. Springer-Verlag, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/11605805_1

[78] Perf. 2021. Tutorial Perf Wiki. https://perf.wiki.kernel.org/index.php/Tutorial. (Accessed on 12/17/2021).

[79] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. 2006. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 135–148. https://doi.org/10.1109/MICRO.2006.29

[80] Pengfei Qiu, Yongqiang Lyu, Jiliang Zhang, Dongsheng Wang, and Gang Qu. 2018. Control Flow Integrity Based on Lightweight Encryption Architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 7 (jul 2018), 1358–1369. https://doi.org/10.1109/TCAD.2017.2748000

[81] N. Roessler and A. DeHon. 2018. Protecting the Stack with Metadata Policies and Tagged Hardware. In *2018 IEEE Symposium on Security and Privacy (SP)*. 478–495. https://doi.org/10.1109/SP.2018.00066

[82] Kirk Yap Wajdi Feghali Jim Guilford Sean Gulley, Vinodh Gopal and Gil Wolrich. [n.d.]. Intel SHA Extensions. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sha-extensions.html. (Accessed on 03/25/2022).

[83] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *NDSS*.

[84] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[85] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and how it improves C/C++ memory safety. *CoRR* abs/1802.09517 (2018). arXiv:1802.09517 http://arxiv.org/abs/1802.09517

[86] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs.CV]

[87] Kanad Sinha, Vasileios P. Kemerlis, and Simha Sethumadhavan. 2017. Reviving instruction set randomization. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 21–28. https://doi.org/10.1109/HST.2017.7951732

[88] Kanad Sinha and Simha Sethumadhavan. 2018. Practical Memory Safety with REST. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 600–611. https://doi.org/10.1109/ISCA.2018.00056

[89] Nicolas Sklavos. 2012. Cryptographic hardware & embedded systems for communications. In *2012 IEEE First AESS European Conference on Satellite Telecommunications (ESTEL)*. IEEE, 1–6.

[90] Mark Stone. [n.d.]. Shellshock In-Depth: Why This Old Vulnerability Won't Go Away. https://securityintelligence.com/articles/shellshock-vulnerability-in-depth/. (Accessed on 12/13/2021).

[91] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. 2017. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 4278–4284.

[92] PaX Team. [n.d.]. https://pax.grsecurity.net/docs/aslr.txt. https://pax.grsecurity.net/docs/aslr.txt. (Accessed on 11/01/2020).

[93] Qualcomm Technologies. 2017. Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf. (Accessed on 12/17/2021).

[94] Ryung Uh, Robert S. Cohn, Bharadwaj Yadavalli, Ramesh V. Peri, and Ravi Ayyagari. 2006. Analyzing Dynamic Binary Instrumentation Overhead Gang -.

[95] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 927–940. https://doi.org/10.1145/2810103.2813673

[96] A. Venkat, Sriskanda Shamasunder, H. Shacham, and Dean M. Tullsen. 2016. HIPStR: Heterogeneous-ISA Program State Relocation. In *ASPLOS '16*.

[97] Xueyang Wang, Sek Chai, Michael Isnardi, Sehoon Lim, and Ramesh Karri. 2016. Hardware Performance Counter-Based Malware Identification and Detection with Adaptive Compressive Sensing. *ACM Trans. Archit. Code Optim.* 13, 1, Article 3 (March 2016), 23 pages. https://doi.org/10.1145/2857055

[98] Xueyang Wang and Ramesh Karri. 2013. NumChecker: Detecting Kernel Control-Flow Modifying Rootkits by Using Hardware Performance Counters. In *Proceedings of the 50th Annual Design Automation Conference* (Austin, Texas) *(DAC '13)*. Association for Computing Machinery, New York, NY, USA, Article 79, 7 pages. https://doi.org/10.1145/2463209.2488831

[99] Xiaoguang Wang, SengMing Yeoh, Robert Lyerly, Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. 2020. A Framework for Software Diversification with ISA Heterogeneity. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 427–442. https://www.usenix.org/conference/raid2020/presentation/wang-xiaoguang

[100] Ye Wang, Qingbao Li, Zhifeng Chen, Ping Zhang, and Guimin Zhang. 2020. A Survey of Exploitation Techniques and Defenses for Program Data Attacks. *Journal of Network and Computer Applications* 154 (03 2020), 102534. https://doi.org/10.1016/j.jnca.2020.102534

[101] Yu Wang, Jinting Wu, Tai Yue, Zhenyu Ning, and Fengwei Zhang. 2022. RetTag: Hardware-assisted Return Address Integrity on RISC-V. (2022).

[102] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. 2018. sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves. In *Middleware*.

[103] S. Weiser, A. Zankl, Raphael Spreitzer, K. Miller, S. Mangard, and G. Sigl. 2018. DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *USENIX Security*

Symposium.

[104] Carter Yagemann, S. Sultana, Li Chen, and W. Lee. 2019. Barnum: Detecting Document Malware via Control Flow Anomalies in Hardware Traces. In *ISC*.

[105] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[106] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. 2008. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 225–240.

[107] Jiliang Zhang, Wuqiao Chen, and Y. Niu. 2019. DeepCheck: A Non-intrusive Control-flow Integrity Checking based on Deep Learning. *ArXiv* abs/1905.01858 (2019).

[108] Jiliang Zhang, Binhang Qi, and Gang Qu. 2018. HCIC: Hardware-assisted Control-flow Integrity Checking. *CoRR* abs/1801.07397 (2018).

[109] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2019. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS* (2019), 631–644. https://doi.org/10.1145/3297858.3304017