Received 28 October 2023, accepted 23 November 2023, date of publication 27 November 2023, date of current version 30 November 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3337043

SURVEY

Hardware-Based Software Control Flow Integrity: Review on the State-of-the-Art Implementation Technology

SENYANG LI, WEIKE WANG, (Member, IEEE), WENXIN LI, (Graduate Student Member, IEEE), AND DEXUE ZHANG
College of Electronic and Information Engineering, Shandong University of Science and Technology, Qingdao 266590, China

Corresponding authors: Weike Wang (wangweike@sdust.edu.cn) and Dexue Zhang (dexuezhang@sdust.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 62201325, in part by the Shandong Provincial Natural Science Foundation under Grant ZR2020QF027, and in part by the Qingdao West Coast New Area Science and Technology Project under Grant 2021-25.

ABSTRACT Code Reuse Attacks (CRA) represent a type of control flow hijacking that attackers exploit to manipulate the standard program execution path, resulting in abnormal processor behaviors. In response to the security concern, proposals for Control Flow Integrity (CFI) verification have emerged. The CFI scheme diligently monitors program jumps during execution, effectively restraining abnormal program execution and robustly safeguarding against CRA. This paper provides a comprehensive analysis and synthesis of the current state of hardware-based CFI implementations. In this survey, we initially discuss common attack methods and variations of predominant CRA, elucidating the general procedural steps intrinsic to such attacks. We delve into the protective capacities inherent in contemporary hardware-based CFI implementations. By conducting a thorough examination and organization of diverse research endeavors on hardware-based CFI, we systematically classify CFI based on implementation methodologies, including label verification, instruction encryption, stack edge detection, instruction tracing, sensitive data isolation, and basic block validation. We provide comprehensive explanations and critical evaluations for each category followed by comparative analyses while offering personal insights on the evolution of hardware-based CFI.

INDEX TERMS Code reuse attacks, control flow integrity, hardware-based CFI implementations.

ABBREVIATIONS

CFI	Control-Flow Integrity.
CRA	Code reuse attacks.
CFH	Control Flow Hijacking.
ROP	Return-Oriented Programming.
JOP	Jump-Oriented Programming.
FG-CFI	Fine-Grained Control Flow Integrity.
CG-CFI	Coarse-Grained Control Flow Integrity.
TE	Trace Encoder.
DEP	Data Execution Prevention.
ASLR	Address Space Layout Randomization.
CFG	Control Flow Graph.
FPGA	Field-programmable gate array.
AES	Active Encryption Standard.
RAB	Return Address Buffer.
PTM	Program Trace Macrocell.
PT	Processor Trace.
TPIU	Trace Port Interface Unit.
MPU	Micro Processor Unit.
IPT	Intel Processor Trace.
MU	Match Units.
TLB	Translation Lookaside Buffer.
FF	Flip-Flop.
LUT	Look UP Table.
BBB	Basic Block Boundar.
RAS	Random Access Storage.
MAC	Massage Authentication Code.
PUF	Physically Unclonable Function.
RoCC	RocketChip with custom coprocessor.

The associate editor coordinating the review of this manuscript and approving it for publication was Byoung Wook Choi.

I. INTRODUCTION

A. THE CURRENT CHALLENGES IN EMBEDDED SYSTEM SECURITY

Recently, the rapid advancements in both the integrated circuit industry and the Internet have led to the seamless integration of embedded devices into various aspects of everyday life and industrial processes. These devices find extensive applications in various domains including healthcare, autonomous driving, industrial robotics, smart homes, and smart cities, permeating every aspect of contemporary existence. The widespread adoption of embedded devices has significantly enhanced productivity and quality of life. However, this rapid acceptance has also given rise to a multitude of cybersecurity challenges.

In the past five years, data from CVEdetails.com has revealed a significant increase in reported vulnerabilities, reaching tens of thousands annually. Notably, the year 2022 recorded an alarming 25,082 reported vulnerabilities. The aforementioned statistics highlight the significant challenges encountered in ensuring the security of embedded devices. Additionally, data obtained from IEEE Spectrum reveals that a majority of embedded programs are written in the C language, which unfortunately lacks the support for CFI mechanisms. The programming languages used provide an opportunity for attackers to exploit security vulnerabilities, such as stack overflow [1], integer overflow [2], and format string overflow [3], thereby enabling the manipulation of program control flow data. Consequently, attackers possess the capability to manipulate the execution sequence of programs on processors, initiating malicious assaults on critical devices and systems that pose significant risks to human life, property, and information security. This type of attack, which manipulates program execution through stack overflow, is referred to as a Control-Flow Hijacking (CFH) [4]. Due to its aggressive nature and ability to bypass existing safeguards in embedded devices effortlessly, CFH attacks remain a paramount and pivotal threat within the realm of embedded device security.

The CFH attack represents a prominent method that exploits memory buffer overflows to manipulate either the return address of a program or the destination of a jump instruction. As a result, it gains control over the program's flow and facilitates the execution of malicious attacks against the targeted system. The CFH attacks can be categorized into two types: code injection attacks [5] and code reuse attacks [6]. In code injection attacks, attackers exploit the vulnerability present in the program's storage region, which lacks boundary checking mechanisms. They identify the location of the return address or branch instruction and then manipulate memory through buffer overflow. This manipulation redirects the modified return address to the location where malicious code was introduced by the attacker. During program execution, the processor executes the injected malicious code block, thereby achieving the objectives of the attacker, which may include compromising the target computer system, exfiltrating sensitive information, or causing harm to system integrity. Both code reuse attacks and code injection attacks exploit buffer overflows to hijack the control flow. However, code reuse attacks eliminate the need for explicit injection of malicious code. Instead, attackers skillfully construct concise code fragments known as "gadgets" using compiled assembly instructions from the program. Each instruction within these gadgets serves a specific purpose, with attackers adeptly interconnecting them to achieve equivalent functionality as the injected malicious code. As a result, code reuse attacks exhibit enhanced concealment and pose greater challenges for defensive measures. Common manifestations of code reuse attacks include Return-Oriented Programming (ROP), Jump-Oriented Programming (JOP), and return-into-libc attacks.

ROP attacks [7], [8], [9], which were initially introduced in 2007 and subsequently recognized as Turing complete, revolve around the manipulation of return instructions. These attacks exploit stack overflow vulnerabilities to manipulate the return address of a function, thereby coercing the processor into executing specific instructions designated by malicious attackers. In contrast, JOP attacks [10], [11] primarily rely on both direct and indirect jump instructions to tamper with the program's execution flow by modifying target addresses associated with function calls and jumps. The return-into-libc [12] attack represents an initial manifestation of the ROP attack strategy, where the attacker manipulates the return address to facilitate a jump and subsequent execution of a targeted function within the libc library, evading defense mechanisms to achieve their objectives. However, this attack's effectiveness is limited due to its reliance on only a restricted subset of functions from shared libraries. This limitation poses challenges for the successful execution of broader attacks. To expand the attack scope, Shacham et al. [10] extended the concept of return-into-libc by linking short instructions in a specific logical order that concludes with a ret instruction, thereby implementing program logic known as the ROP attack. Therefore, ROP and JOP attacks are considered prominent mainstream methodologies within code CRA.

B. THE CURRENT STATE OF DEFENSE AGAINST CODE REUSE ATTACKS

In response to the continuous evolution of attack techniques, security researchers have developed and implemented diverse defense technologies to provide a certain level of protection for embedded devices. Nevertheless, CRA has demonstrated the ability to circumvent specific prevailing defense mechanisms, primarily owing to their distinctive trait of circumventing the need for injecting malicious code into the target system. An example of this is Data Execution Protection (DEP) [13], which modifies the execution permissions of memory's code regions by designating data segments as non-executable and code segments as non-modifiable. When a shellcode is executed through a buffer overflow, DEP triggers an error signal to effectively protect against code injection attacks. However, DEP remains vulnerable in the context of code reuse attacks, as established attack methodologies have demonstrated the ability to bypass its protective measures [14].

Address Space Layout Randomization (ASLR) [15], different from DEP, introduces randomization to the stored return addresses to prevent attackers from determining the location of return instructions within the stack. Even if attacks are successful in overflowing the stack, computing the position of the return address presents a significant challenge. As a result, modifying critical data to initiate powerful code reuse attacks becomes more difficult. However, vulnerabilities still exist in both coarse-grained and fine-grained ASLR implementations, and malicious actors can bypass this defensive mechanism [8]. Although DEP and ALSR increase the difficulty of implementing code reuse attacks, they both need the support of an operating system and are difficult to implement for resource-constrained embedded systems.

The landscape of defense strategies has been disrupted by code reuse attacks, challenging the erstwhile attack pattern centered on the assumption that attackers were restricted to triggering malevolent code injections solely from external sources. To address the imminent threat posed by code reuse attacks, Abadi et al. [16] introduced a pioneering methodology for implementing CFI, which combines software and hardware techniques. At the core of CFI lies the concept of imposing stringent constraints on the program's control flow, ensuring its execution aligns with a predetermined transfer flow graph. By enforcing this constraint, the program's transitions are confined within a secure scope, thereby endowing CFI with remarkable efficacy in mitigating CFH attacks and providing an efficient defense against CRA. The Control Flow Graph (CFG) is an analysis of the sequence in which instructions are executed during program execution, and constructing a program jump relationship graph based on these jumps is crucial for implementing CFI. The accuracy of CFG is an important metric of CFI, and based on the accuracy of CFG CFI can be classified as fine-grained CFI [17], [18] and coarse-grained CFI [19], [20], [21]. Both schemes have their advantages and disadvantages, coarse-grained compared to fine-grained, coarse-grained scheme security performance will be much lower and can be bypassed by complex code reuse attacks [22], [23], [24], [25], [26], [27], [28]. However, this approach reduces the frequency of legitimate verification of program jumps during program execution. Therefore, the implementation cost of this solution is relatively low, which meets the application requirements of practical embedded systems.

Currently, CFI in a processor can be achieved in both Software [29], [30], [31], [32], [33], [34], [35] and hardware. BCI-CFI [34], an implementation of software-based fine-grained CFI on the Linux kernel, exhibited an average execution overhead of 19.67%, with the peak overhead attaining 31.2% in test programs. This elevated execution overhead bears the potential to induce diminished processor performance, heightened power consumption, squandered resources, and compromised system stability. Consequently, the feasibility of applying software-based fine-grained CFI to real-world applications becomes questionable. Consequently, the implementation of various software-based CFI schemes on physical processors often faces significant challenges primarily due to the notable increase in execution overheads.

In response to the challenges arising from software-based CFI, researchers have introduced hardware-based CFI techniques [35], [36], [37], [38], [39], [40], [41], [42], [43], [44], [45], [46] In terms of hardware, security experts design hardware circuits for processors or integrate additional storage capacity and instruction set extensions to implement CFI. Hardware-based CFI significantly reduces processor execution overhead. For instance, HCIC [42] successfully implemented CFI by developing hardware circuits within existing processors, resulting in a minimal average execution overhead of 0.95%. This demonstrates the effectiveness of hardware-based CFI in mitigating the limitations arising from high software execution overheads. However, incorporating external hardware circuits often introduces increased complexity, and implementing hardware-based CFI may require compiler modifications for software instrumentation, thereby amplifying the associated implementation challenges. Nonetheless, designing an external hardware circuit is frequently intricate, and modifying it afterward becomes inflexible; additionally, integrating additional hardware circuits will also expand the processor chip's area.

To enhance the adaptability of hardware-based CFI and eliminate the need for compiler support, researchers propose utilizing the processor's existing hardware modules to actively monitor program instruction execution in real-time and trace the program's control flow. Notable among these hardware modules is Intel's PT module [47], which effectively collects control flow transition information from the CPU. Conversely, Program Trace Macrocell (PTM) [37] tracks the historical data related to instructions executed by ARM cores and subsequently compresses this instructive information. Subsequently, researchers analyze the decompressed data to gain insights into the program's control flow during execution, a crucial process for CFI validation. TE [48] is an open-source RISC-V processor instruction tracking compression module that compresses executed instructions into data packets. These packets are analyzed by researchers to obtain the program's control flow during execution, providing insights into its behavior. Alternative hardware-based CFI solutions involve modifying the processor framework to support novel extended instructions designed for CFI validation, such as TrustFlow-X [40], HAFIX [46], and HCFI [49]. Coprocessors also emerge as a viable avenue for implementing CFI. For instance, Nile [50] assumes the role of monitoring various events during program execution to optimize and scrutinize processor behavior. By leveraging this module, security experts can execute CFI and conduct validation without requiring compiler modifications or additional instructions, thereby significantly simplifying the deployment complexities of CFI. Furthermore, utilizing an autonomous coprocessor for CFI validation enhances the inherent adaptability of hardware-based CFI implementation. However, this approach has certain limitations as it solely detects whether the processor has been attacked and cannot proactively defend against malicious attacks.

C. MAIN CONTRIBUTIONS

This present paper summarizes and analyzes recent research on implementing control flow integrity for mitigating against CRAs. In contrast to Sayeed et al. [51] review of software and hardware for control flow integrity, we focus on an overview of the work on implementing CFI in hardware, focusing on the characteristics of the hardware. Compared with the similar Kumar et al. [52] who proposed a new classification method for the review of the control flow integrity of the hardware-assisted mechanism, we carried out a classification review of the current hardware implementation of CFI from the perspective of the implementation method of the hardware control flow integrity.

This article's primary contributions encompass.

- We provide a comprehensive analysis of CRA methods and present a concise summary of the general steps involved in executing such attacks.
- From a novel perspective, we categorize existing hardware-based CFI implementations into six distinct categories based on their implementation methods, elucidate diverse hardware features employed, and provide critical evaluations.
- We conducted a comprehensive comparative analysis of diverse methods for implementing CFI, considering their respective advantages, disadvantages, security implications, execution overheads, and resource utilization. Based on these findings, we provide insights into the future development trends of hardware-based CFI.

In the subsequent sections of this article, we embark on a comprehensive exploration. Section II meticulously elaborates on CRAs, encompassing an elucidation of distinct features, including those beyond ROP and JOP. Moving to the third segment, we provide a methodically categorized summary of capabilities achievable through prevailing control flow integrity defenses. In the following part, we categorically introduce existing efforts in the field of hardware-based CFI implementations. Transitioning to the fifth division, we engage in a thorough comparative analysis that succinctly dissects various implementation approaches while highlighting their characteristics and hardware attributes. Overall, this article systematically and comprehensively delves into hardware-based CFI implementations by unveiling their strengths and limitations while concurrently paving the way for future innovations in this domain.

II. CODE REUSE ATTACKS

Both code injection attacks and code reuse attacks exploit stack overflow to manipulate stored return addresses and associated parameters on the stack. However, their distinction lies in the attack code they employ. Code injection attacks require malicious code to be injected from external sources into the application's memory space. In contrast, code reuse attacks leverage pre-existing instructions within the program to construct a series of program chains known as 'gadgets.' The gadgets used in ROP attacks consist of instruction sequences culminating in a 'ret' instruction.

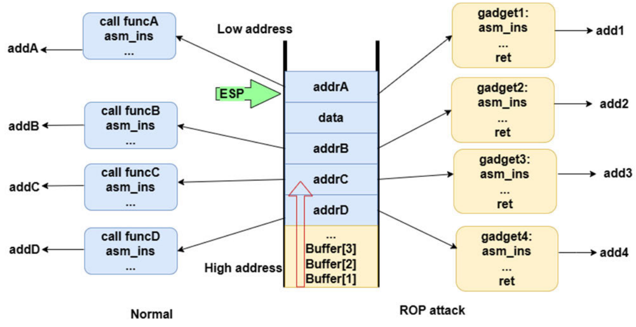


FIGURE 1. The comparison between the process of ROP attack and the process of normal program execution.

The DEP mechanism acts as a defensive measure against code injection attacks by modifying memory attributes to make data writable but non-executable, effectively thwarting code injection attacks [53]. However, the predominant threat in contemporary times is the code reuse attack, which can be further classified into two archetypal variants based on the terminal instruction of the repurposed code segment: ROP and JOP.

ROP attacks involve the chaining of sequences of gadgets that culminate in "ret" instructions, while JOP attacks rely on sequences ending with "jmp" instructions. These forms of CRAs have emerged as significant threats to modern systems, necessitating robust countermeasures for their mitigation. The ROP attack relies on indirect function calls and return instructions. When a function call instruction "call" is executed, the processor performs a context-preserving operation through the stack to preserve the address of the subsequent instruction following the function call. Subsequently, the processor executes the instructions within the invoked function zone. Concluding this series, the final instruction within this region is a return instruction "ret." By executing this instruction, the return address is retrieved from the stack and assigned to the processor's program counter (PC), facilitating a context-restoration operation.

In a typical processor execution (illustrated on the left side of Fig. 1), the stack serves as a crucial component for managing function calls and returns. It stores the return addresses of various functions, allowing the program to progress smoothly from one function to another. However, this orderly process is disrupted when a ROP attack is initiated (As depicted on the right side of Fig. 1). During an ROP attack, the attacker carefully identifies specific instruction locations within the target program to construct a sequence of gadgets. These gadgets are short snippets of code that end with a "ret"

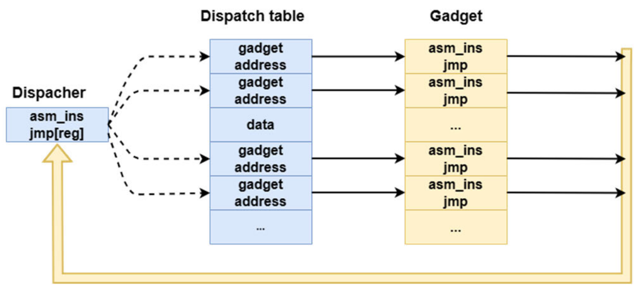


FIGURE 2. JOP attack process.

instruction, which causes the processor to jump to the next gadget in line. To carry out such an attack, the attacker takes advantage of a stack overflow vulnerability. By overflowing certain buffers or variables on the stack, they can overwrite the original return addresses with addresses pointing to their chosen gadgets instead. When executing these manipulated code segments embedded with malicious logic, known as payloads or exploits, allows attackers to achieve their objectives. The processor no longer follows its regular flow from function A to D but rather diverges into executing each gadget in succession – gadget1 followed by gadget2 and so on. Each gadget performs specific actions desired by the attacker and may include instructions like loading values into registers or modifying memory contents. By skillfully chaining together these gadgets through careful selection and arrangement of their respective return addresses on the stack, attackers can effectively bypass security measures and gain unauthorized control over targeted systems. The consequences of successful ROP attacks can be severe. Attackers could exploit system vulnerabilities for purposes such as gaining unauthorized access or privilege escalation within an organization's network infrastructure. Therefore, understanding how ROP attacks work is essential for developing effective countermeasures against them and ensuring a robust system.

The JOP attack, like ROP, exploits the program's existing code to manipulate the processor and control the program's execution flow. However, there are distinctions between these two approaches. JOP attacks manipulate the program's control flow by utilizing the indirect jump instruction 'jmp' and the indirect call instruction 'call'. The target address for the indirect jump is obtained from a specified register, which can be modified by attackers by altering the parameter values stored in the stack. Consequently, these modifications enable attackers to exploit JOP attacks on the processor using both indirect call and jump instructions.

As illustrated in Fig. 2, the JOP attack model involves the attacker's search for target gadgets and subsequent incorporation of their addresses into a data region known as the dispatch table. This dispatch table includes a dispatch gadget pointer that refers to the table itself. This pointer interacts with a specific register, triggering a jump to the address stored in that register, corresponding to a gadget's address. Upon execution of the gadget, control is redirected back to the dispatch gadget pointer, which then indicates the address of the next gadget within the dispatch table. This iterative process traverses through the addresses in the dispatch table, thereby facilitating a successful JOP attack. The comparison between ROP and JOP attacks reveals distinct methods for modifying target addresses. ROP exploits stack overflow vulnerabilities to manipulate return addresses, while JOP alters the value of the register containing indirect jump instructions through stack overflow. This divergence in attack techniques renders existing ROP defense mechanisms ineffective against JOP attacks.

Beyond ROP and JOP attacks, a range of other code reuse attack variants exists, including GOT Overwriting [54], [55], Speculate execution [56], [57], [58], Sigreturn [2], Return-to-libc [59], and ROP's variant Call-Preceded [27]. An inclusive overview of significant attack techniques is delineated in Table 1, encapsulating attack methods, techniques, core concepts, and defining traits for each attack.

The analysis of the provided Table 1 reveals that prevailing CRA techniques exploit vulnerabilities specific to processors, such as stack overflow, branch prediction errors, and side-channel attacks. These vulnerabilities enable unauthorized access to the processor, facilitating manipulation of the program's control flow or acquisition of sensitive data. The diverse objectives and methods employed in these attacks highlight their wide-ranging nature. Focusing on code reuse methods discussed earlier, the essence of these attacks lies in an attacker's ability to manipulate the program's execution sequence. Therefore, when implementing CFI strategies within a processor context, it is crucial to first obtain the program's CFG, which represents its standard execution sequence.

By analyzing the prevalent code reuse attacks, we present a comprehensive outline of the implementation steps involved in executing code reuse attacks as shown in Fig. 3. In the initial phase, the attacker acquires the program's source code and employs relevant gadget search tools to identify available gadgets within the program. The locations of these gadgets are recorded in preparation for the subsequent step of stack overflow, where in modifications are made to both the return address and register value. During this phase, defenders can employ ASLR to introduce variability in code locations upon each program execution, thereby disrupting previously obtained gadget addresses and significantly enhancing attack complexity. However, it is important to note that ASLR protection can still be circumvented through information leakage attacks [60], [61]. The second step involves executing a stack overflow attack, wherein the attacker utilizes functions like strcpy(), strcat(), sprintf(), and others to inject payloads into the stack to manipulate the value of the return address stored on the stack as well as registers associated with indirect jump addresses. To mitigate against modifications to these values caused by stack overflow, De Asmit et al. [39] proposed PUFCanary, which entails storing a section of secret bytes at the boundary of the stack. When a stack buffer overflows, these secret bytes are altered. By verifying these secret bytes for detecting potential stack overflow

TABLE 1. Main code reuse attack techniques at present.

	Method of attack	Means of attack	Attack core idea	Attack characteristics
GOT Overwriting [54]	Stack overflow vulnerability.	Build malicious programs using existing program code.	The dynamic linking mechanism in the program is used to modify the target address of the function in the GOT.	The knowledge of the target program's memory layout is essential for locating and overwriting function pointers in the GOT, enabling arbitrary code execution.
Return-to-libc [58]	Stack overflow vulnerability.	Existing functions of the system library.	Bypass control of execution locations.	The attacker must possess knowledge of the memory layout and library function addresses of the target program.
Sigreturn [2]	Stack overflow vulnerability.	Call the Sigreturn () function.	The sigreturn function is called to modify the processor state.	Bypassing the call-return mechanism of a function allows direct control over the processor state.
Speculate execution [59]	Branch prediction error vulnerability.	The result of exploiting the wrong predicted branch.	Attackers exploit the speculative execution nature of processors to execute sensitive code or obtain confidential information without authorization.	Spectre attacks span different processor architectures and have a wide range of impacts. This attack technique is difficult to detect and repair.
Call-Precede [27]	Using timing attacks or side-channel attacks.	Using the processor's execution cache.	A special sequence of precall instructions is inserted before the target instruction.	The program's execution flow can be controlled by inserting special instruction sequences and utilizing the target instructions in the cache, leveraging the processor's execution cache and out-of-order execution characteristics for further attacks.

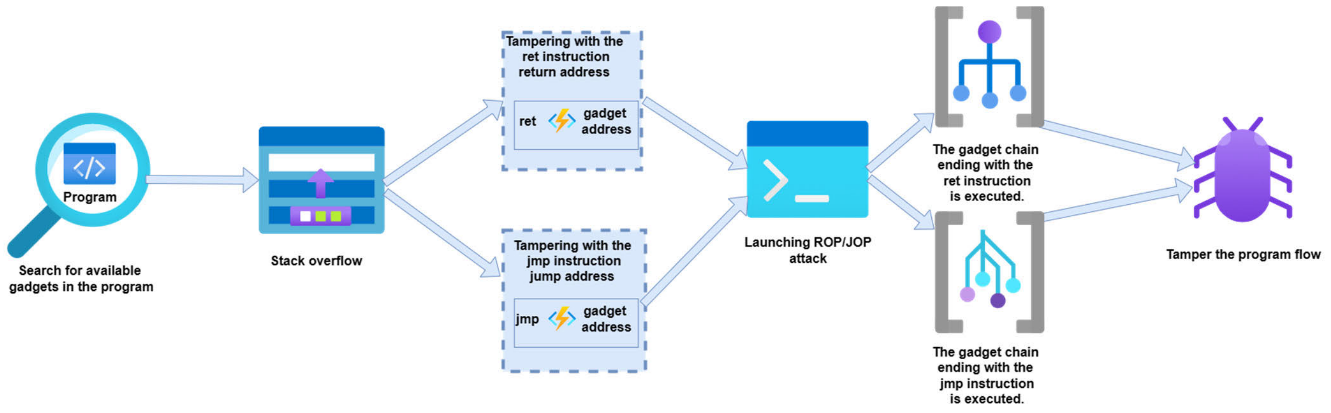

Program
Search for available gadgets in the program
Stack overflow
Tampering with the ret instruction return address
ret
gadget address
Tampering with the jmp instruction jump address
jmp
gadget address
Launching ROP/JOP attack
The gadget chain ending with the ret instruction is executed.
The gadget chain ending with the jmp instruction is executed.
Tamper the program flow

FIGURE 3. The general procedure for conducting CRA attacks.

occurrences, this approach effectively addresses issues related to buffer overflow attacks on stacks. However, it is important to note that this method is currently only implemented in simulation and requires further validation before deployment within an actual system. The third step of the attack involves the execution of a meticulously crafted series of gadgets by the attacker, following a specific logic, ultimately leading to program control flow hijacking. In this step, related research endeavors aim to narrow down their focus by utilizing these available gadgets, with BBB-CFI [41], dividing the program into blocks, and employing an instruction tracking module that enforces jump instructions to target only entry addresses of basic blocks. This approach effectively reduces 90% of potential gadget exploitation. It greatly increases the difficulty of code reuse attack implementation.

III. THE FUNCTIONS IMPLEMENTED BY CFI

The CFI technology is employed to detect unauthorized transfers within a program. During program execution, CFI verifies the target addresses associated with indirect jump, indirect call, and return instructions to ensure legitimate redirection of the program. This capability effectively defends against various code reuse attack techniques. Obtaining a comprehensive understanding of the authorized target addresses of the program requires acquiring the CFG, which encompasses all valid execution pathways and target addresses during routine program operation.

The CFG encompasses comprehensive details regarding all authorized transfer destinations throughout the program's execution. Security professionals extract and process the jump-related data of the program, with some referring to the

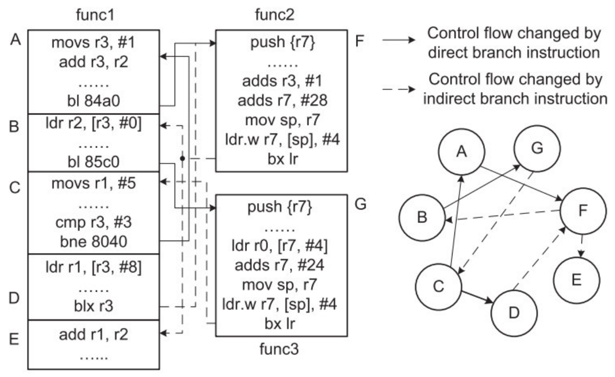


FIGURE 4. CFG transformation based on basic block Information in FastCFI [37].

extracted content containing jump-related details as ''metadata'' [62]. This ''metadata'' constitutes crucial information that needs to be extracted before deploying CFI. During program execution, the processor utilizes this metadata information to authenticate target addresses for each jump instruction including conditional branch, unconditional jump instructions, and system call instructions.

The CFG plays a crucial role in the implementation of CFI. Currently, there are various techniques available for extracting CFG, including static, dynamic, and hybrid methods. Prominent tools used for extracting program control flow include the LLVM compiler, Python scripts, and GDB debugging tools. Before extraction, the executable file of the target program is obtained and then disassembled using the corresponding processor's toolchain to generate the corresponding assembly file. By analyzing and parsing the instructions in the assembly file, the program can be divided into distinct basic blocks. Each basic block consists of a jump instruction followed by a sequence of consecutive instructions that ensure no self-looping within the block occurs. Basic blocks can be classified into three types based on their contained jump instructions: indirect jump blocks, indirect function call blocks, and function return blocks.

Subsequently, the CFG extraction tool establishes inter-block jump relationships by considering the presence of jump instructions within each basic block, encompassing both conditional branches and unconditional jumps. By computing the target addresses of these jump instructions, the tool identifies all potential target basic blocks, thereby constructing a CFG that visually represents the program's transfer connections. The specific composition of this CFG is depicted in Fig. 4. Following the construction of the program's CFG, CFI technology ensures that the target program functions as intended based on the delineated pathways within the CFG.

Taking Fig. 4 as an illustrative example, each segmented basic block corresponds to a node, with directed arrows denoting paths between nodes that signify valid transitions. func1 is partitioned into four fundamental blocks: A, B, C, D, and E, with directed arrows indicating connections to their authorized target blocks. Thus, the CFG can be symbolized as a set G = (N, L), wherein N signifies the collection of nodes within the CFG, and L signifies the assortment of valid pathways for each node. The legitimate transfer destination for node A is F, and any transition from node A to B during program execution would constitute an unauthorized transfer. However, there are certain limitations associated with the prevailing approach of constructing CFGs. Extracting CFGs for large-scale programs using existing tools can present challenges. Furthermore, static methods of CFG extraction cannot encompass dynamic target nodes generated based on input parameters during program execution [37]. The precision of the extracted CFG directly correlates with the security level of CFI, where greater precision in CFG construction results in heightened CFI security. Nevertheless, current extraction tools often experience inaccuracies that impact to some extent both the precision of CFI and the scope of CFI detection. During program execution, CFI assesses if the program's transfer paths align with the CFG. Based on techniques applied to authenticate the validity of target addresses during program execution, existing approaches to CFI can be categorized into coarse-grained CFI [19], [20], [21], [63], [64] and fine-grained CFI [17], [18], [32], [65]. Moreover, depending on whether historical branch information is integrated into target address validation, it can be classified as stateful CFI [66], [67], [68], [69], or stateless CFI [39], [44], [62]. Additionally, considering whether CFI is tailored to thwart ROP attacks or JOP attacks allows segmentation into backward-oriented CFI [70], [71] and forward-oriented CFI [72]. Subsequent sections will provide more comprehensive insights into these functionalities.

A. COARSE-GRAINED CFI AND FINE-GRAINED CFI

Coarse-Grained Control Flow Integrity (CG-CFI) represents a CFI technique that compromises security for reduced overhead, aiming to enhance the practicality of CFI schemes. Several studies [20], [21], [63], [64] have implemented CG-CFI with a shared characteristic: centralized management of target addresses for indirect jumps without assigning distinct identifiers to each valid target address. Evaluation results demonstrate that the average execution overhead remains below 5%. However, this reduction in overhead comes at the expense of security, making CG-CFI vulnerable to bypass attacks. Fig. 5 illustrates a model showcasing the bypass of CG-CFI. func1 directly jumps to func2 and func3 through function direct call instructions, resulting in both functions having the same ID (ID1). func2 indirectly jumps to func4, and func3 indirectly jumps to func4 and func5. Since both func2 and func3 jump to the same function (func4), the entry labels of the target functions are the same.

Consequently, even though func2 would not normally jump to func5, an attacker exploiting stack overflow to modify the target address of an indirect function call could lead to an illegal path where func2 jumps to func5. In such a case, CFI would not recognize this target address as illegal. During normal program execution, the 'ret' instruction's valid target is the subsequent instruction after the indirect call.

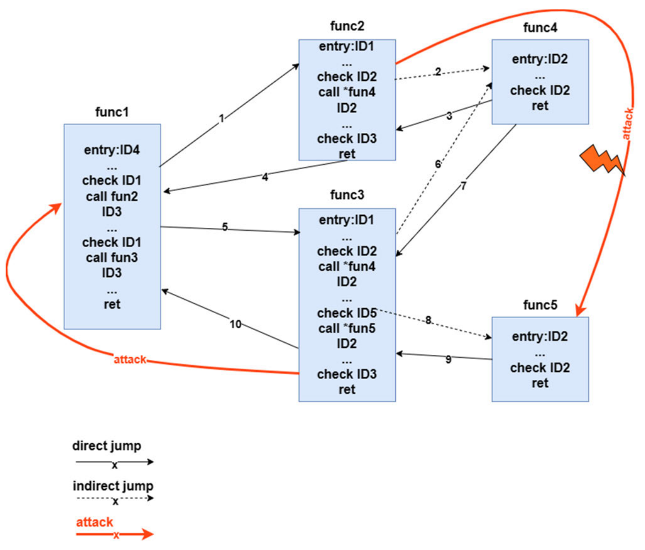


FIGURE 5. The defense and bypass model of coarse-grained CFI.

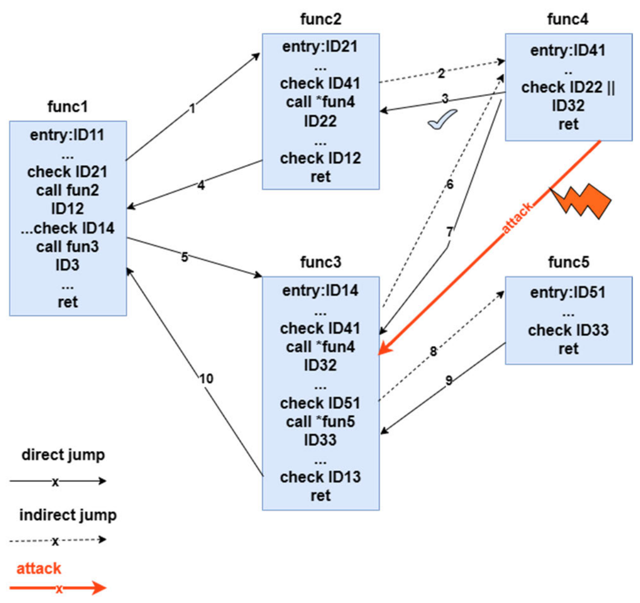


FIGURE 6. The defense and bypass model of fine-grained CFI.

To restrict runtime program transfer paths, a code instrumentation approach incorporates labels at the entry addresses of function blocks. Before executing the function call instruction, a verification process ensures coherence between the function label and the entry ID of the destination function block, thereby validating the legitimacy of the target address. Similarly, upon completion and return from a function block, authentication is performed on the label associated with the return address for its legitimacy.

The Fine-Grained Control Flow Integrity (FG-CFI) approach distinguishes itself from Coarse-Grained CFI by individualizing each valid target address through the assignment of unique verification labels. However, it will incur high execution overhead and increase the capacity of code storage. While effectively addressing the limitation of low granularity in CG-CFI, the increased number of labels necessitates additional processor executing time for verifying target addresses of branch instruction during program execution, resulting in escalated processor overhead and performance decline.

Moreover, while FG-CFI effectively restricts the target addresses of function call instructions, its protection mechanism for "ret" instructions is comparatively weaker, making it susceptible to be bypassed by ROP. As illustrated in Fig. 6, an ROP bypass attack on FG-CFI is demonstrated. In Fig. 6 func2 and func3 both have func4 as their target function. Therefore, during the return of func4, the legitimate labels include ID22 and ID32. As a result, when func4 returns, it can return to both func2 and func3 as valid targets. Attackers can exploit this scenario to bypass FG-CFI by returning to func3 in step 3 without FG-CFI's validation. In Fig. 6, each function is assigned a unique entry ID to prevent attackers from freely navigating through legitimate targets using indirect function calls. However, since multiple functions can redirect to a single function, multiple valid target addresses may arise when a function performs a return operation. Consequently, FG-CFI still faces the risk of being circumvented by attackers.

To enhance the security of FG-CFI and mitigate bypass attacks, certain approaches leverage the Shadow Stack technique. This technique leverages the program's inherent execution rules: upon a function call event, the processor stores the address of the subsequent instruction on the stack. Subsequently, during the execution of the "ret" instruction, the processor retrieves and transitions to that designated address from the stack.

The Shadow Stack establishes a secure storage realm within the processor, secluded from attackers' purview, thus deterring their attempts to manipulate stack contents. During a function call, the processor records the return address both on the conventional stack and the Shadow Stack. When the "ret" instruction is executed, the processor compares the return address from the stack against the trusted address contained in the Shadow Stack. A match signifies the validity of the return address, whereas a mismatch deems it invalid.

However, the conventional implementation of a stack introduces an approximate execution overhead of 10% [73]. Furthermore, in the traditional shadow stack approach, return addresses are typically dispersed rather than densely packed together, resulting in each return address occupying a separate cache line. This incurs costs depending on the calling pattern of the program. Moreover, as the synchronization between the shadow stack pointer and execution exists, adversaries can manipulate the stack pointer to point to any desired return address and even modify it to reference expired return addresses based on entries from previous shadow stack frames.

B. STATEFUL CFI AND STATELESS CFI

During the process of transforming program control flow, the validity of these changes can depend on previously executed transformations. As illustrated in Fig. 7, both transitions from

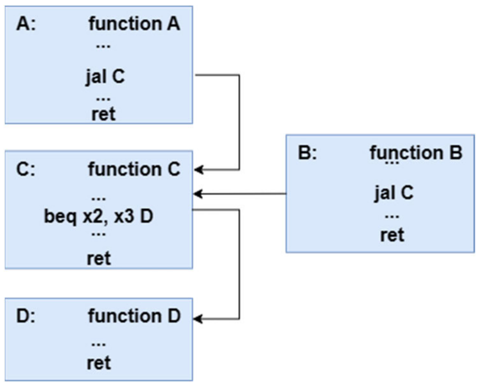


FIGURE 7. An illustrative program fragment that performs conditional jumps based on the program context.

A to C and from B to C are considered valid, as well as the transformation from C to D. However, only the sequential transformation from A to C to D remains valid. Conversely, if the sequence of transformations is B to C to D, it is deemed invalid. This ability to determine the accuracy of branch transitions based on contextual insights characterizes stateful CFI or dynamic CFI [66], [67], [68], [69]. Studies such as those in [66], and [67] utilize Intel processors' Performance Monitoring Unit (PT) to record execution context paths during program execution. Additionally, this approach necessitates protection against attackers attempting malicious write maneuvers on the defense data.

During the validation of control flow transformations, this technique examines runtime contextual data to evaluate the appropriateness of program transition targets. In contrast, static CFI solely evaluates whether runtime jump relationships adhere to statically established CFG transformation relationships. If these transformations remain coherent, static CFI maintains their legitimacy.

Consequently, dynamic CFI provides enhanced security compared to static CFI. However, hardware assistance is required for dynamic CFI to obtain runtime contextual insights.

C. FORWARD CFI AND BACKWARD CFI

JOP attacks manipulate the target addresses within registers of indirect jump instructions through stack overflow, while ROP attacks hijack the program's control flow by tampering with return addresses via stack overflow. These distinct attacks employ different gadgets, with JOP attacks relying on gadgets culminating in "jmp" instructions, whereas ROP attacks leverage gadgets culminating in "ret" instructions. Consequently, CFI techniques tailored to safeguard against JOP attacks may not effectively defend against ROP attacks. Therefore, CFI can be divided into forward CFI and backward CFI according to whether it defends against a JOP attack or ROP attack. Forward CFI is a technique that validates the legitimate target addresses of indirect jump instructions, while Backward CFI focuses on protecting the target addresses of "ret" instructions.

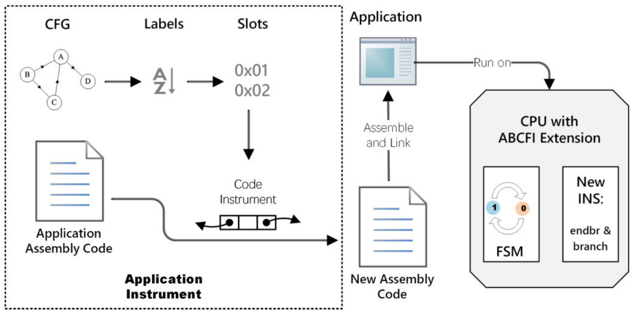


FIGURE 8. Application preparation workflow with ABCFI: Compilation, Code Instrumentation, Assembly/Linking, and execution [72].

IV. HARDWARE IMPLEMENTATION OF CFI TECHNIQUES

The current research focus revolves around achieving robust security and minimizing execution overhead in the implementation of CFI. This section aims to categorize hardware-based CFI techniques based on their implementation methods, encompassing six aspects: Label Verification, Instruction Encryption, Stack Edge Checking, Instruction Tracking, Sensitive Data Isolation, and Basic Block Validation. We will present relevant works in hardware-based CFI, analyze their advantages and disadvantages, and summarize the implementation approaches within each category.

A. LABEL VERIFICATION

Label verification is a commonly employed approach for implementing CFI. Notable studies such as [45], [46], [49], and [51] proactively acquire the legitimate CFG of the program by inserting instructions to establish labels at the call points of the target execution program based on CFG analysis. The label validation instruction, positioned at the call point, serves as a unique identifier for the target address. During program execution, a comparison between the label of the call point and the label of the called point determines whether the program's target address adheres to its CFG.

The ABCFI [72] workflow depicted in Fig. 8 presents a generic process for implementing label-based CFI, consisting of an offline stage and an execution stage. During the offline stage, the program's assembly code is inserted with labels, and the processed assembly code is linked to generate executable files for program execution. Subsequently, the CFI-protected processor executes the modified program code to protect the safety of the program jump through the verification of the label.

HCFI [49] implemented a fine-grained static CFI scheme on SPARC SoC processors, enhancing information security by extending a secure shadow stack and label registers within the processor core. To prevent unauthorized modification of the shadow stack and label register values, dedicated instructions for accessing these elements were introduced through an expanded instruction set, thereby strengthening CFI security. In the offline phase, a Python script (200LoCs) was used by the instruction instrumentation tool to insert SetPCLabel and SetPC instructions in the delay slot after indirect call, direct call, and return instructions. Additionally, the CheckPC instructions were then inserted before the first instruction of each function to effectively mark and differentiate valid target addresses for branch instructions. During execution, direct and indirect call instructions initially execute extension instructions that store the target address label in the label register and save the function's return address in the shadow stack. Subsequently, upon executing the first instruction of the target function, it retrieves the stored label from the label register to validate against its corresponding address. After function execution completes, a comparison is made between stored return addresses in the shadow stack with the current return address before executing any return instruction. In the event of any discrepancies, the program will utilize the return address from the shadow stack for continued execution and error logging.

In comparison with HAFIX [46] by forcing the program to return to the active target program, HCFI [49] extends CFI defense further by introducing dedicated instructions and a shadow storage array to address security concerns related to stack expansion and active tags triggered by setjmp and longjmp. Additionally, a 128*1-bit bitmap is incorporated to store function recursive index flags, enabling support for recursive calls. The proposed solution modifies the Lenon3 softcore and deploys it on the Xilinx m1605-rev. e Field-Programmable Gate Array (FPGA). Test results demonstrate that the HCFI [49] incurs an execution overhead of less than 1%, registers' hardware overhead increases by 2.52%, and Look-UP Table (LUT) increases by 2.55%. This scheme successfully achieves fine-grained CFI with JOP and ROP protection at a cost of less than 1%. Furthermore, it supports recursive function calls, setjmp, and longjmp while maintaining low CFI overhead for enhanced security. However, this approach requires software instrumentation along with complex instruction set expansion and processor architecture modification, contributing to its intricacy. Moreover, deploying shadow stack and label registers on the chip leads to increased processor area overhead as well as constraints in shadow stack capacity.

HAFIX [46] implemented fine-grained static backward CFI on Intel Siskiyou and SPARC processors. This approach involved expanding the instruction set and incorporating a dedicated label memory to indicate the activity status of functions. Additionally, it enforced that return instructions within a function only returned to the active function. During the offline phase, each function was assigned a unique label by the compiler, which was stored in the label memory to represent its state during program execution. By exclusively constraining the target address of return instructions to reside within the active function region, this implementation effectively reduces available attacker gadgets and overcomes capacity limitations associated with shadow stacks found in HCIC [42]. Notably, no operating system support is required for deploying this scheme on the Siskiyou processor and LEON3, demonstrating its scalability. Furthermore, the test results demonstrated an overall execution overhead of 2%. Resource evaluations conducted using Xilinx PAR indicated a register increase ranging from over 2% to under 3% for both schemes, with a negligible increase in LUTs of less than 1%. However, it is crucial to note that this scheme exclusively ensures backward CFI protection while remaining vulnerable to forward-edge attacks. Moreover, its implementation necessitates modifications in processor architecture and compiler support for precise label insertion at designated locations. In scenarios involving multiple function calls, the presence of multiple active functions may compromise strict program return to the nearest caller location, thereby posing a potential security risk.

Similar to HAFIX [46], the work by Davi, Lucas et al. [45] also implements CFI based on function state, with a notable distinction. The proposed scheme by Davi, Lucas et al. achieve both forward and backward CFI technology, providing comprehensive defense against code reuse attacks. To accomplish this, the scheme introduces two novel label instructions, CFIBR and CFIRET, for regulating return addresses. Each target function is marked with CFIBR at its entry instruction to assign a unique label. Additionally, the subsequent instruction following a function call is associated with CFIRET. During program execution, the CFIBR instruction stores the label of the current function in the label memory to indicate its active status. Upon executing a call instruction, it enforces that the following instruction must be a CFIBR instruction and stores the label of the called target function in the label memory as well. When a ret instruction executes, it verifies whether both the target instruction is a CFIRET instruction and if its corresponding tag is active to ensure validity of return addresses. However, it should be noted that this scheme falls under the coarse-grained CFI category which makes it susceptible to bypass risks.

ABCFI [72] presents an optimized implementation of fine-grained static CFI using labels, offering a significant implementation over conventional label-based CFI approaches. Unlike the label-based CFI techniques employed in previous works such as CFI [45], [46], [49], which require unique labels before the call and target points to achieve FG-CFI, ABCFI [72] addresses the challenges associated with these operations. Conventional methods involve the processor performing two crucial tasks during instruction jumps: storing the pre-jump label and validating the post-jump label. This results in substantial area overhead and execution costs. In response, ABCFI introduces the concept of a "Slot" with a bit width of n, representing the lowest n bits of an instruction address. This approach utilizes the branch instruction slot to differentiate between branch instructions and target functions, eliminating the need for labels before branch instructions. Consequently, it reduces the number of inserted labels into the program, alleviating label storage issues during execution. Additionally, this scheme introduces a new instruction called "endbr," placed before the target instruction. Subsequently, it enforces that only "endbr" is executed after a branch instruction. To further distinguish valid target addresses, it replaces the lowest n bits of endbr's

TABLE 2. **Summary and comparison of the work of tag verification to achieve CFI.**

	LUTs	Flip Flops	Code Size	Performance Overhead	Hardware components	ISA Support	Whether CFG is needed	Whether code instrumentation is required
HCFI [49]	2.55%	2.52%	6.60%	<1%	FSM Label Register	Y	Y	Y
HAFIX [46]	0.33%	2.97%	N/A	2%	Label Memory	Y	Y	Y
Davi [45]			N/A		FSM Label State Table	Y	Y	Y
Dean Sullivan [74]	1.78% area		13.5%	1.75%	LSR	Y	Y	Y
ABCFI [72]	0.93%	0.72%	8.16%	0.55%	FSM	Y	Y	Y

address with corresponding bits from the branch instruction at the call point. This refinement enhances target address distinction and overall CFI security.

The unique approach of ABCFI [72] avoids the necessity of instrumenting call points, thereby eliminating the need for registers to store labels for verification. Consequently, it effectively reduces resource consumption and execution overhead without compromising security. In comparison, HCFI [49] and FIXER [39] exhibit area overheads of 2.5% LUTs and 2.9% area overhead respectively. Deploying ABCFI on Xilinx Zynq VC707 demonstrates that the scheme only requires 10 LUTs without label register support and incurs an execution overhead of less than 0.55%. However, as the scheme is based on code instrumentation, it results in a final code overhead of 8.16%. Additionally, while it achieves forward CFI, it remains susceptible to ROP attacks.

As shown in Table 2, we have summarized the CFI execution overhead and resource consumption of various existing label verification implementations. It is evident that CFI implemented through label verification optimally utilizes processor resources and often obviates the need for internal processor architecture modifications, thereby facilitating deployment and achieving fine-grained CFI with enhanced security and minimal execution overhead. Nevertheless, the table also highlights certain limitations associated with label-based CFI implementations.

- To prevent the unauthorized disclosure of tag data, label-based CFI technology necessitates the allocation of dedicated storage space and the extension of instructions for accessing this space. This safeguards against tampering by attackers but introduces additional overhead in terms of both area and execution.
- The efficacy of label-based CFI implementation heavily relies on static CFG analysis. Consequently, the protection scope of label-based CFI hinges on the precision of CFG extraction and code coverage. Present CFG extraction tools struggle to achieve accurate CFG extraction, limiting their ability to indirectly jump to target addresses. As a result, label-based CFI implementation fails to assign labels for indirect jump target addresses, leaving a vulnerability.
- The placement of extended instructions within branch instruction positions requires the support of code instrumentation tools. However, the security of these tools is not guaranteed and may introduce new vulnerabilities. Additionally, these tools typically access and modify the source code to insert verification instructions at appropriate locations. This creates a reliance on the source code, rendering the tool unusable in scenarios where the source code is unavailable or unmodifiable.

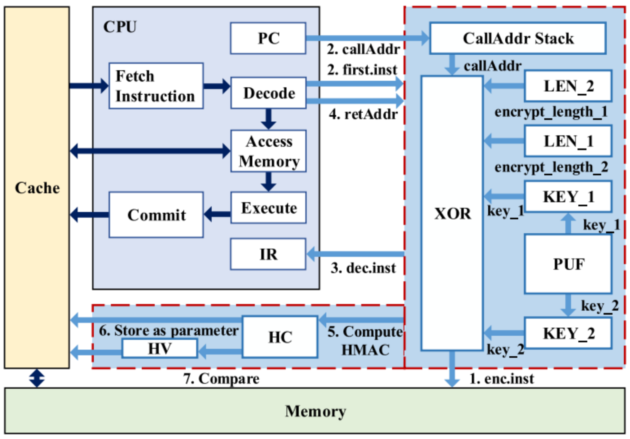


FIGURE 9. **The instruction encryption architecture employed in FH-CFI [36].**

B. INSTRUCTION ENCRYPTION

In contrast to CFI implemented through label verification, CFI implemented via instruction encryption [36], [42], [74], [75], [76] demonstrates reduced dependence on the source code and eliminates the need for expanding the instruction set or employing code instrumentation tools for adding source code size. Instruction encryption facilitates dynamic encryption and decryption of instructions while ensuring the protection of indirect jump instructions.

To initiate a code reuse attack, the attacker first analyzes the positioning of program branch instructions and subsequently manipulates them. To address issues related to instruction address obscurity, HCIC [42] implemented CFI by employing hamming distance calculations and XOR encryption on both the return address of the 'ret' instruction and the target instruction of the branch instruction. This approach resulted in the establishment of coarse-grained static CFI. During program execution, when a 'call' instruction is executed, EHD1 is derived by computing the Hamming distance between a randomly generated key_1 from the Physically Unclonable Function (PUF) and the return address of the target function. Subsequently, both the function's return address and EHD1 are pushed onto the stack. Upon execution of the 'ret' instruction, similar calculations are performed to determine EHD2 using the current return address and key_1 value. A comparison between EHD2 and EHD1 is then conducted to verify equality, thereby detecting any tampering with the return address.

Compared to the return address encryption method used by Qiu et al. [76] using a basic XOR operation, the EHD-based encryption approach significantly enhances the security of the return address. To safeguard 'call' and 'jmp' instructions, XOR encryption is applied to the initial instruction of all functions. This process utilizes key_2 generated by PUF and the address of the initial function instruction before program memory loading. Since the same key is used for encrypting all function's initial instructions, this scheme qualifies as coarse-grained CFI. However, it lacks sufficient capability to differentiate between all valid target addresses. Despite HCIC [42] generating two distinct keys through PUF, these keys are not inherently random, thereby introducing a potential vulnerability related to key exposure.

In contrast to the approach proposed by Sullivan et al. [74], HCIC [42] refrains from making modifications to the compiler or extending the instruction set. Additionally, it incurs a minimal overhead of only 0.78% in binary code execution. Compared with the study conducted by de Clercq et al. [34], testing outcomes obtained from the RIPE framework demonstrate that HCIC [42] exhibits an average execution overhead of just 0.95% and achieves a false positive rate of 0%. However, it requires modification of the processor and relies on hardware circuitry that is exclusively compatible with a single processor, thus presenting limitations in terms of portability. Furthermore, this scheme's adoption of a coarse-grained CFI approach remains susceptible to potential circumvention.

The overall architecture of FH-CFI [36] is illustrated in Fig. 9. Similar to HCIC [42], FH-CFI employs PUF to generate two distinct keys, which are utilized for encrypting the destination address of both the first instruction and the return instruction within each function. In contrast to HCIC, FH-CFI implements a fine-grained defense against JOP attacks and ROP attacks. Moreover, FH-CFI adopts a different approach by utilizing the function call and function jump instruction addresses along with key_2 to encrypt the length of the initial instruction within the called function before loading it into processor memory. This unique method ensures that function call and jump instructions are exclusively direct to valid instructions, thereby enhancing effective target differentiation. when a function call or jump instruction is executed, FH-CFI will decrypt the first instruction of the target function based on the address of the branch instruction itself to protect the program transfer target.

The protection of return addresses is verified through message verification codes. When a function call instruction is executed, both the target return address and key_1 are pushed into the calculation module of the message verification code. Once the computation is completed, both the message verification code and return address are stored in the function stack. Upon encountering a return instruction, the corresponding return address is popped from the stack. The popped return address will be subjected to similar calculations as those performed during the computation of the message verification code, to validate its legality.

Subsequently, the computed message verification code will be compared with its counterpart saved in the function stack. If they do not match, it indicates an attack on the integrity of return addresses; otherwise, this verified return address will be sent to the PC for further execution.

FH-CFI [36] encounters a many-to-one issue with the de Clercq et al. [34] scheme, wherein multiple function call instructions calling the same function can only one function decrypt the first instruction of the called function, thereby causing decryption errors for other function calling instructions and resulting in program execution errors. However, FH-CFI effectively resolves this predicament by introducing novel nodes amidst multiple nodes targeting the same destination. These additional nodes enable direct transitions to the shared target without necessitating encryption and decryption processes. By encrypting and decrypting the initial instruction within these new nodes, security experts indirectly address the security challenges posed by multiple nodes converging onto a common destination. Nevertheless, integrating new nodes amplifies both program execution overhead and code capacity. Finally, rigorous testing on Raspberry Pi 3 validates this methodology's effectiveness in preventing exploitable vulnerabilities for potential attackers when compared to studies conducted by Qiu et al. [75] (false negative rate of 0.91%) and LEA [76] (false negative rate of 0.26%), FH-CFI approach exhibits zero false negatives.

Table 3 compares CFI implementation using the introduced instruction encryption approach described above. Table 3 shows that this method requires sophisticated yet lightweight encryption algorithms to enhance security while minimizing execution overhead. Unlike label-based CFI, it incurs minimal memory overhead and imparts comparatively modest execution overhead. It also enables efficient CFI implementation without requiring compiler modifications or instruction extensions, which is impossible with label-based CFI.

However, the many-to-one challenge inherent in instruction encryption often results in coarse-grained schemes generated through instruction encryption methods. While

TABLE 3. **Comparison of related work on instruction encryption.**

	Encryption algorithm	Key protect	Memory overhead	Runtime overhead	False negative	Defense granularity	Reduce gadgets
HCIC [42]	XOR，EHD	Y	0.78%	0.95%	0%	Coarse	100%
Qiu [75]	XOR	Y	0.63%	0.61%	0.26%	Coarse	91.92%
FH-CFI [36]	XOR，HMAC	Y	1.49%	2.42%	0%	Fine	100%
LEA [76]	AES	Y	0.62%	3.19%	0.91%	Coarse	88.93%

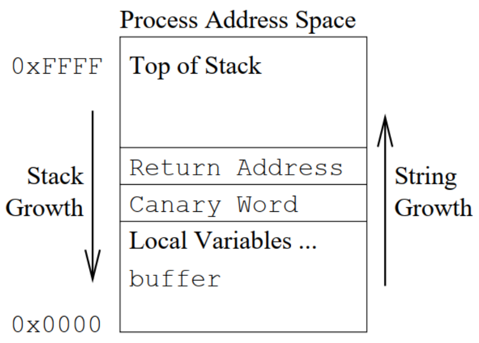


FIGURE 10. **Stack frame modification for canary placements [78].**

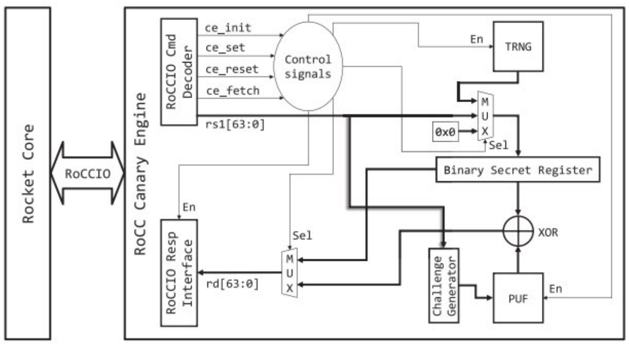


FIGURE 11. **Canary Engine architecture is used to generate stack secret words in PUFCanary [39].**

FH-CFI [36] proposes a solution for this issue, its application requires an increase in code capacity and introduces key exposure vulnerability if attackers successfully bypass the instruction encryption mechanism. Moreover, encrypting control flow transfer instructions mandates hardware-based implementation of lightweight algorithms to mitigate execution overhead and entails precise alterations to processor architecture. Furthermore, inherent deficiencies within the encryption algorithm may lead to false positives or incomplete coverage resulting in false negatives as demonstrated by Qiu et al. [76], as outlined in Table 2.

C. STACK EDGE DETECTION

The adoption of stack edge checking technology plays a crucial role in achieving CFI verification, serving as a pivotal strategy. By adopting this approach, the implementation of CFI reduces its reliance on CFG graphs and eliminates the need for encrypting and decrypting individual branch instructions, as depicted in Fig. 10. This implementation ensures the security of both branch instructions and their associated arguments by introducing canary words into the stack. The direct execution strategy involves incorporating canaries near the control program's data transfer points within the stack vicinity, as discussed in [39], [65], [77], [78], [79], and [80]. However, this method requires additional dedicated storage space and a mechanism to generate random canaries to ensure security and randomness. Consequently, including these features increases processor area resource consumption and necessitates code instrumentation, thereby contributing to increased execution overhead. Noteworthy advancements in this field include Li et al.'s work [70] and the Zipper Stack [80], which introduced indirect stack edge detection technology. This approach modifies the stack structure, applies a lightweight Advanced Encryption Standard (AES) encryption algorithm [81], and performs Hash calculations to verify the message authentication code of transfer instructions. By employing this strategy, data integrity within the stack is fortified against malicious tampering. Importantly, this approach not only reduces execution overhead but also minimizes resource utilization.

PUFCanary, proposed by De et al. [39], is a hardware-based security technique designed to protect buffers within the embedded RISC-V architecture from code injection and code reuse attacks. The block diagram of PUFCanary is illustrated in Fig. 11. This method utilizes PUF and a true random number generation (TRNG) module to detect stack overflow by generating canaries based on insertion addresses. In contrast to Zhu et al.'s [77] approach, where canaries were stored within buffer RCB before being allocated to the stack, this scheme eliminates the need for canary storage, effectively reducing resource consumption. Test results demonstrate a minimal overhead of only 2.2% for this implementation, significantly lower than the 3.2% execution overhead observed in Zhu et al.'s [77] work.

Contrary to the conventional StackGuard [78] canary scheme, which uniformly inserts identical canaries across all buffer positions, PUFCanary [39] generates distinct canaries based on their intended locations within the stack. Consequently, even if an attacker acquires knowledge of a canary, launching a successful attack remains infeasible. To elaborate, PUFCanary leverages PUF responses and True Random Number Generation (TRNG) outcomes to generate unique canaries. During program execution, the PUF delivers a response corresponding to the specific position of each canary within the stack. This response is then XOR with a TRNG-generated random number to produce the respective canary value. The generated canaries are subsequently placed into the stack after branch instructions. To ensure precise insertion of these unique canaries, an analysis of the assembler file is conducted before program execution to identify stack addresses requiring placement. Through collaboration with the compiler and code instrumentation technology, instructions for both setting up and verifying these unique canaries are integrated into the code. When the key placement instruction in the program is executed, canary calculation and canary push operations will be performed. Later, when the canary verification instruction is executed, a new canary will be calculated in the same operation as the placement canary instruction, and the new canary will be compared with the canary placed in advance to verify whether the stack is subject to a stack overflow attack at this time. This method can not only protect the control flow instructions from being modified but also protect the data flow from being modified, expanding the protection scope of CFI.

The work of [39], [77], [78], and [79] necessitates the generation of canaries and the execution of additional instruction operations to insert the canary at a designated position in the stack. This approach impacts both processor area and execution overhead, requiring supplementary hardware module support and compiler assistance. Therefore, related work has been proposed by designing a new stack structure combined with a message verification code. However, implementing CFI based on message authentication code verification often results in processor halts during new Message Authentication Code (MAC) calculations, thereby reducing processor execution efficiency and amplifying execution overhead. Mashtizadeh et al. [65] implemented a lightweight AES [81] algorithm through hardware to enhance pointer integrity detection and improve MAC calculation speed. Nevertheless, due to frequent function calls and returns, this method incurred an average expense of 52%. Differencing from Mashtizadeh et al.'s [65] approach, the Zipper Stack [80] utilizes a hash algorithm for computing the hash value of the return address. Subsequently, a chain structure verifies the message authentication code of the stack's return address to counteract code reuse attacks. Specifically, upon function call, the hash value is computed based on the current return address and stored within a designated register called the TOP register. Upon function return, recalculating the hash value based on the return address allows comparison with the stored hash value in the TOP register; equality leads to removal of that specific return address from stack memory. The reliable TOP register makes it impractical for attackers to bypass hash checks; however, this strategy incurs an execution overhead of 23%.

In a notable contribution, Li et al. [70] introduced an innovative Stack mode that incorporates a Dislocated Stack and buffer Return Address Buffer (RAB). This enhanced stack includes an additional RAB space with a predetermined capacity, designed to store recent return addresses for stack push and pop. The RAB buffer functions as a cache for return addresses during MAC computation, providing a time interval for uninterrupted MAC code computation to prevent processor halts. However, it still faces challenges related to processor halts when executing multiple call instructions successively. In this study, Lazy verification and batch verification techniques are implemented based on the stack structure. In the initial approach, return addresses entering the Dislocated Stack are considered potentially subject to rewriting, while return addresses solely residing in RAB are deemed secure and trustworthy. This approach introduces additional stack pointers denoted as q and v within RAB, which serve the purpose of recording whether ongoing entries require calculation and verification for the MAC code. Because it needs to ensure the security of the return address so lazy verification computes the MAC for each return address entering and exiting the Dislocated Stack, to further reduce the verification count, batch verification is employed to validate the entire volume within RAB. This strategy significantly reduces both calculation and verification instances, effectively mitigating the processor's execution overhead. The final two verification methodologies were tested on the Xilinx Zynq VC707 board using varying RAB sizes and SPEC CINT 2000 benchmark suite. In lazy verification mode with a RAB size of 8, execution overhead amounts to 1.23%, accompanied by consumption of 1699 LUTs and FFs 1056. For batch verification with an RAB size of 4, execution overhead is reduced to 0.78%, consuming only 546 LUTs and 321 FFs. This novel stack structure enables concurrent execution of message authentication code computation and processor operations, thus enhancing overall processor efficiency. Moreover, Lazy Verification reduces over 99% of MAC verification on function returns and Batch Verification cuts down 78% of MAC calculations.

Two proposed verification methodologies effectively reduce the frequency of message authentication code verifications and subsequently mitigate processor execution costs. This achievement is attained while maintaining a reasonable area overhead, thereby demonstrating practical feasibility. However, the stack structure currently exists only in simulation without real-world implementation on specific processors.

Therefore, functional validation of this approach remains pending. Importantly, this scheme exclusively protects against ROP attacks but may still be vulnerable to other potential attack vectors.

PUFcanary [39] and DiffGuard [77] (as listed in Table 4) employ Canary word valuation within the stack, necessitating code instrumentation and reliance on the source code, which could potentially introduce supplementary security vulnerabilities to the source program. Moreover, due to

TABLE 4. Comparison of stack detection related work.

	Random canary	Code instrument	Defense ROP	Defense JOP	Performance overhead
StackGuard [78]	N	Y	Y	Y	N/A
Shrivastava [79]	Y	Y	Y	Y	5%
DiffGuard [77]	Y	Y	Y	Y	3.2%
PUFCanary [39]	Y	Y	Y	Y	2.3%
Zipper Stack [80]	No need canary	N	Y	N	23%
Li [70]	No need canary	N	Y	N	1.78%

the constant need for calculation and verification of data within the stack, its execution cost exceeds that of label and instruction encryption. While this approach enhances CFI security at the expense of execution cost, stack edge detection extends its protective scope beyond solely the return address to encompass crucial parameters within the stack. Li et al. [70] and Zipper Stack [80] circumvented dependence on the source code by reshaping the stack structure, thereby eliminating the need for code instrumentation support. Notably, Li et al. [70] approach achieved additional reduction in execution overhead, indicating significant potential for enhancing in CFI through stack edge detection.

D. INSTRUCTION TRACKING DETECTION

The CFI implemented by inserting canary words in the stack detection method requires code instrumentation. Code instrumentation will process the program execution code in the offline stage, which increases the deployment time of CFI. Hardware auxiliary modules such as instruction tracking modules PTM [37], PT [47], TE [48], and co-processors are the key to implementing CFI based on instruction tracing detection. They obtain the current program transfer path by analyzing the processor instruction execution information and detecting whether the current path complies with CFG. Moreover, the implementation of CFI through tracing and detecting instruction execution information obviates the need for code instrumentation, thereby effectively reducing program capacity and execution overhead. This approach also facilitates dynamic CFI implementation compared to label-based or stack detection methods. Furthermore, code instrumentation is inherently insecure, potentially giving rise to additional security concerns. To circumvent the necessity of code instrumentation for CFI implementation, recent research has shifted its emphasis towards leveraging the instruction trace and debug modules embedded within the processor itself. Certain endeavors were established based on Intel's processor trace [47], [85], [86] while others implemented CFI through utilization of processor debug tools [87], [88]. Furthermore, there has been significant interest in implementing CFI using processor auxiliary modules, as demonstrated by examples such as the CFI scheme relying on processor counters [82] and its implementation on the open-source RISC-V processor [48]. FastCFI [37] ingeniously employed FPGA resources to design a dedicated hardware detection circuit for CFI, effectively analyzing and verifying control flow-related instructions dispatched by the instruction execution information tracking module PTM [37] of ARM processors. As a result, dynamic fine-grained forward and backward CFI detection is achieved, all without necessitating code instrumentation. Similar to Intel's PT [47] module, PTM [37] serves as an integral instruction monitoring module within the ARM processor, capable of condensing instruction execution information during runtime (as shown in Fig. 12). In the context of FastCFI [37], PTM establishes data communication with the FPGA-based CFI detection module through the Trace Port Interface Unit (TPIU) interface. While relying on CFG, this approach diverges from Anthony et al.'s [48] methodology which involved storing indexable and searchable metadata in RAM after CFG processing.

The designer of Fast-CFI [37] developed an automated tool capable of extracting the CFG from the binary file of the target program, which serves as input for generating a Verilog HDL file. This Verilog HDL file contains transformation details for each basic block within the CFG diagram. Subsequently, the conversion information of basic blocks will be stored in hardware circuits and utilized by the CFG Checker module on FPGA for runtime verification. This processing approach accelerates the verification process of branch instruction target addresses but incurs significant consumption of FPGA resources. Moreover, for excessively large execution programs, comprehensive verification of all control flow information transformations may become impractical on FPGA due to the proliferation of numerous CFG nodes resulting from program execution. The storage limitation concerning nodes poses an additional constraint on CFI implementation.

To address this challenge, Fast-CFI [37] presents a novel solution involving both CFG compression technology and CFG subgraph verification technology. The former technique leverages the inherent security of direct branch addresses, which are effectively protected through Write OR Execute mechanisms to ensure address integrity. To reduce node size within the CFG, Fast-CFI introduces a process for compacting direct branches.

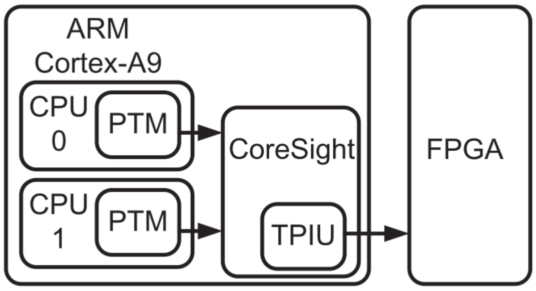


FIGURE 12. The system platform of instruction execution information tracking and decoding is proposed in FastCFI [37].

The latter technique involves deploying and verifying critical CFG subgraphs within the CFG Checker. This novel approach focuses on implementing and verifying pivotal data's CFG to safeguard it against potential leaks. The efficacy of the proposed scheme [37] was demonstrated through its implementation on both the Altera DE1-SoC development board and the Cyclone V FPGA development board. Rigorous testing was conducted using the RIPE [89] program test set to assess the security of program execution. The outcomes reveal the scheme's proficiency in countering ROP and JOP attacks, effectively thwarting specific dynamic CFH endeavors. To evaluate program execution overhead, SPEC CPU2006 [90] benchmarks were employed.

The results underscore the scheme's efficiency, demonstrating an exceedingly low execution overhead of just 0.36%. In comparison to CFI detection accomplished through software means, which yields an execution overhead ascribed to [85], Fast-CFI [37] execution overhead becomes inconsequential. Moreover, contrasted with a similar endeavor [83] utilizing PTM and the TPIU interface, with an execution overhead reduction of over 30%, Fast-CFI [37] surpasses it due to the avoidance of code instrumentation. In summation, Fast-CFI unifies the capabilities of processors and FPGAs to achieve dynamic fine-grained forward and backward CFI with minimal overhead. Importantly, this achievement is realized without necessitating code instrumentation or modifications to the processor architecture.

The contributions encompass the introduction of CFG compression technology and subgraph CFG technology. However, this solution relies on PTM support and has poor portability. Moreover, this solution parses the data packets generated by PTM to obtain information about the instructions during program execution, which means that the solution can only detect the attack after the attack occurs and cannot prevent attackers from launching attacks in advance so this scheme cannot be applied to security scenarios such as banking and autonomous driving. Nile [50] serves as a specialized coprocessor for analyzing trace information logs generated during processor instruction execution, facilitating the detection of program execution processes. In contrast to PUMP [91], which involves altering the processor pipeline, Nile enhances its flexibility by interfacing with the RISCV processor through the Rocket Custom Coprocessor (RoCC) [92] interface.

Nile [50] is a coprocessor used to analyze and detect the trace information logs of processor instruction execution during program execution. Compared to PUMP [91], which modifies the processor pipeline, Nile improves its flexibility by communicating with the RISCV processor through the RoCC [92] interface. Nile receives commit logs sent by the processor through the RoCC interface, including undecoded instructions, current instruction PC value, next instruction PC value, current instruction storage address, and register address, as well as data accessed by the current instruction.

These data are broadcasted in packet form to multiple configurable MU units. The MU units are used to parse the packets in the commit log and obtain current instruction execution information for monitoring different events. By using the Nile function, this scheme utilizes the Nile to implement shadow stack and plays a role when configuring two MU units to monitor call instructions and ret instructions. One MU unit writes return addresses into shared memory space while another MU unit reads them and compares them with ret instruction return addresses for detecting ROP attacks. The advantage of this scheme is that it can balance the security and execution overhead relationship by configuring the number of MU, and this coprocessor can track instruction information in out-of-order executed processors in parallel effectively reducing execution overheads. However, this scheme has an area overhead of 15% and requires operating system support with limited applicability to embedded systems. ACE-M [35] uses the memory protection unit MPU to monitor control flow errors during program execution. MPU can configure the read and write execution permissions of the code area. Before program execution, use the LLVM [95] compiler to insert the MPU permission configuration function. This function can configure permissions for specific code areas. According to High-level read and write permissions will overwrite low-level read and write permissions rule the permission configuration function will configure higher permissions for the target execution area before the execution of the call instruction and before the execution of the ret instruction, ensuring that only the part of the area called by the call instruction and returned by the ret instruction can be performed. Through the configuration of permissions, only specific code areas can be executed to ensure the legality of program transfer. Therefore, this solution can ensure the security of program execution. However, the permission configuration of the hardware MPU needs to be implemented in software, and the execution program needs to be code instrumented, resulting in the execution overhead of this solution reaching up to 56%.

In Table 5, a succinct overview of the instruction tracking modules, namely PTM [30], Intel Processor Trace, and TE [41], employed within the current processor ecosystem is provided. Upon comparative analysis of these methodologies, it becomes evident that FastCFI [37] emerges as a frontrunner in terms of performance. This solution ingeniously employs supplementary FPGA resources to establish

TABLE 5. Comparison of related work on instruction execution information tracking detection.

	Instruction tracking component	Introduction Components	Storage CFG	Code instrumentation	Dynamic CFI	Performance overhead	Area overhead
FastCFI [37]	PTM	generates compressed control-flow traces according to instructions processed by the ARM core.	Y	Y	Y	0.36%	30K ALMs (Adaptive Logic Module)
Nile [50]	MU	MUs are programmed to monitor a distinct event indicated by a wildcard match on the commit log.	N	N	N	<3%	15%
Liu [86]	IPT	IPT generates trace information of running programs in the form of packets.	N	Y	Y	3.79%	N/A
Zgheib [48]	TE (Trance Encoder)	The TE reports the uninferable discontinuities in its control flow in form of trace packets.	Y	N	N	N/A	10%
ACE-M [35]	MPU	MPU may provide attributes read-write-execute to the memory area.	N	Y	N	5.87%	N/A

a hardware-assisted enforcement CFI mechanism external to the processor, effectively surmounting the resource constraints often encountered by contemporary processors.

Leveraging the powerful computational capabilities inherent in FPGA technology, FastCFI [37] manages to maintain an impressively minimal execution overhead of only 0.36%. Furthermore, the CFI implemented on FPGA exhibits high portability and can be upgraded for security defense in specific application scenarios with great practicality.

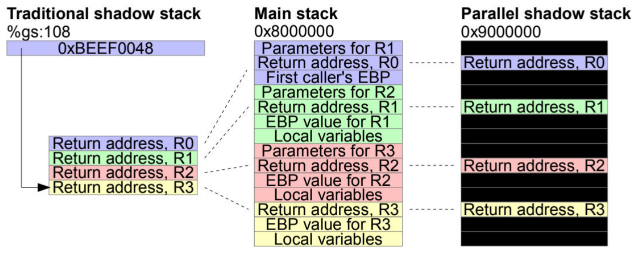


FIGURE 13. Comparative contents of the traditional shadow stack structure and the parallel shadow stack [73].

E. ISOLATE SENSITIVE DATA

Employing data isolation, particularly through shadow stack implementations, stands as a prevalent strategy for achieving CFI. Among these, the shadow stack is very representative, exemplified by its capability to effectively thwart ROP attacks by storing return addresses independently. The structural depiction of the shadow stack model is showcased in Fig. 13. In the domain of shadow stack application, De et al. [39]. presented FIXER, as a significant contribution. A notable aspect of FIXER is its utilization of custom instructions to effectively manage the shadow stack and Policy Matrix Memory, both located external to the processor, by leveraging the RoCC [92] interface.

Different from the shadow stack implemented by HAFIX [46] inside the processor, FIXER [39] implements a secure shadow stack outside the processor, as shown in Fig. 14. The size of the shadow stack implemented inside the processor will be limited by the processor area resources. If the program is large, the entire program will not be protected. FIXER [39] employs code instrumentation tools to analyze program code pre-execution, strategically inserting CFI_CALL and CFI_RET labels before call and ret instructions, respectively. Subsequently, during the program compilation, the compiler translates these labels into executable disassembly instructions aligned with the RoCC [92] instructions. The execution of the compiled program code by the processor involves storing the return address of the function in the shadow stack when a call instruction is executed. Upon encountering a ret instruction, a verification process occurs, which includes comparing the current return address with its stored counterpart in the shadow stack. This enables the detection of any potential tampering with return addresses. Empirical findings highlight FIXER's effectiveness, demonstrating only a 1.5% increase in execution overhead.

However, it should be noted that despite its advantages, while expanding processor area, and the shadow stack approach inherently focuses on mitigating ROP attacks and cannot defend against JOP attacks.

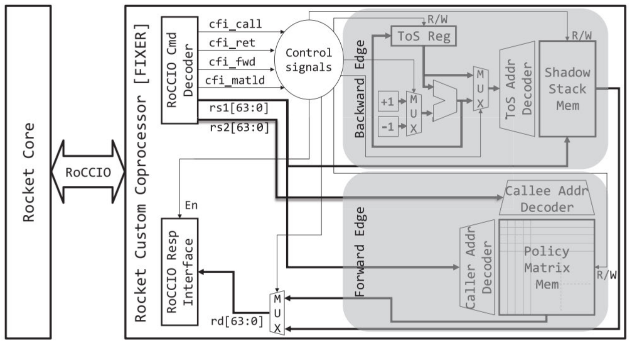


FIGURE 14. The comprehensive CFI protection architecture implemented in FIXER is governed by custom instructions for the shadow stack and Policy matrix memory [39].

TrustFlow-X [40], an innovative hardware/software co-design framework implemented on the RISC-V instruction set architecture in conjunction with the Clang/LLVM [95] compiler infrastructure, introduces a robust and fine-grained mechanism for static control flow integrity protection. The TrustFlow-X [40] approach follows a top-down design methodology, consisting of two essential components: software design and a secure execution environment. In terms of software design, the TrustFlow-X framework enables secure differentiation of sensitive data during program execution by utilizing a safe function library that allows security practitioners to appropriately tag sensitive data. To ensure the integrity of these labels, the toolchain ensures effective discrimination between sensitive and normal data during program execution.

The latter component, the secure execution environment, is dedicated to preserving the integrity of sensitive data. Notably deviating from conventional approaches, TrustFlow-X places paramount emphasis on source code security by generating secure executable code right from its inception. By utilizing a secure database, TrustFlow-X systematically differentiates sensitive data through the utilization of a secure toolchain and function library, subsequently facilitating the execution of the secure code on an extended RISC-V processor. Diverging from existing hardware CFI implementations [39], [41], [42], [43], [44], [45], [46], [49], which halt the processor and generate an error signal upon detecting code reuse attacks, TrustFlow-X employs a novel strategy that records error information and leverages isolated secure data for repairing compromised data without disrupting normal device operation. Specifically, TrustFlow-X introduces a trusted buffer known as Translation Lookaside Buffer (TLB), which utilizes custom extension instructions sws and lws to store the addresses of sensitive data along with their corresponding values in the TLB during program execution for subsequent verification. This mechanism ensures detection of any tampering attempts on sensitive data. During sws instruction execution, sensitive data is stored both in regular memory and TLB, while lws instructions extract and compare the current data stored in regular memory with the sensitive data stored in TLB, thereby guaranteeing the integrity of sensitive data residing in regular stack.

In contrast to the shadow stack implemented by De et al. [39], which solely protects the return address of backward CFI, TrustFlow-X [40] utilizes an isolated and trusted external memory outside the processor to ensure the integrity of all sensitive data. Consequently, TrustFlow-X [40] protects against ROP and JOP attacks, as well as safeguarding program data integrity. The proposed approach was validated on Xilinx Arty-35 Field Programmable Gate Array (FPGA). Empirical evaluations demonstrate minimal execution overhead, below 1%, and marginal area overhead increases of 1.03% in LUTs and 1.16% in Flip-Flops (FFs). TrustFlow-X [40] has good security performance under the condition of negligible performance overhead, but it needs the support of the compiler and needs to modify the processor pipeline, which is poor in portability.

Beyond the conventional approach of isolating return addresses through a shadow stack, as elucidated by TrustFlow-X [40], a broader application can be envisioned, wherein multiple categories of sensitive data isolation would be pursued to enhance the security of critical information. An embodiment of this concept is exemplified in RCFI [38], which minimizes the execution overhead of the processor by isolating parameters that impact the key CFI detection frequency, reducing frequent CFI verifications. Ultimately, RCFI [38] achieves static coarse-grained forward CFI with high security and low execution overhead through randomized verification of branch instructions. The verification probability denoted as ‘p’ regulates the frequency at which verifications occur and depends on both a random number ‘S’ and a counter ‘C.’ Consequently, if either ‘S’ or ‘C’ are tampered with by an attacker, RCFI [31] becomes vulnerable to attacks. To mitigate this vulnerability, values for ‘S’ and ‘C’ are stored within an isolated storage arena to ensure data security. Furthermore, RCFI [38] effectively handles recursions at varying depths while incurring less than 1% execution cost, rendering it negligible. However, it is crucial to acknowledge that this scheme relies on compiler support and remains susceptible to exploitation when substantial verification frequencies are involved.

In Table 6, we present a summary of the data isolation schemes for achieving CFI. It is evident that this approach not only effectively ensures the integrity of return addresses and related parameters but also safeguards other critical data. Thus, data isolation holds universal applicability and can be employed in CFI implementations utilizing label verification or instruction encryption to protect keys and mitigate key leakage risks. In essence, data isolation serves as a prevalent means to realize hardware-based CFI. Security personnel can ensure data security by storing sensitive information within trusted isolated regions. To further enhance data safety, security personnel may expand dedicated access instructions for isolating spaces to prevent attackers from tampering with individualized data, thereby enhancing overall security measures. However, it should be noted that allocating additional space will consume processor resources and necessitate

TABLE 6. Comparison of sensitive data isolation related work.

	Isolated data	Isolated locations	The way of isolation	Dependent code or not	Data repair supported	Perform overhead	Area overhead
FIXER [39]	Return address	Off-chip	Shadow stack	Y	N	1.5%	2.9%
TrustFlow-X [40]	Return address and return address related parameters	Off-chip	TLB	N	Y	<1.0%	22% (18643LUTs 12099Flip-flops)
RCFI [38]	Random Source S and Counter C	In-chip	Dedicated register	Y	N	4.0%	N/A
DangP [73]	Return address	Off-chip	Shadow stack	N	N	3.5%	N/A
MengB [93]	Return address	In-chip	Shadow stack	Y	N	3.5%	1.4%

supplementary operations for transferring data into isolated regions, resulting in certain execution overhead.

F. THE BASIC BLOCK VERIFICATION CALCULATION

The CCFI-Cache [43] leverages both hardware storage and detection modules located external to the RISCV processor. Its primary goal is to ensure the integrity of both the code and the CFG. Distinct from the single-instruction encryption and decryption validation, CCFI-Cache accomplishes fine-grained static CFI by employing basic block hash value verification. Notably, to optimize CFI verification's execution efficiency, synchronization is established between the number of instructions in each metadata basic block and their corresponding instruction basic blocks. This entails congruence in the number of instructions and offsets, facilitating parallel access to metadata information during instruction retrieval and thus obviating intricate address calculations.

During program execution, CCFI-Checker concurrently extracts instruction blocks from the instruction cache while calculating the Hash value of the said instruction block. This calculated Hash value is then compared against the results stored in CCFI-Cache, serving as a verification step. The outcome determines if any erroneous jumps occur in the program's control flow, effectively defending JOP attacks. To mitigate ROP attacks, a shadow stack is deployed to protect the processor. This measure entails inserting null instructions within mutual basic blocks to achieve congruence between instruction counts in metadata and corresponding basic blocks. The goal is to mitigate processor cessation during the computation of the instruction basic block's hash value. However, introducing null instructions comes with repercussions. The processor's execution overhead experiences a 32% increment due to null instruction insertion, signifying that instruction synchronization contributes to elevated execution costs. Furthermore, cache capacity expansion leads to an approximate 10% increase in hardware overhead pertaining to LUTs and FFs.

Analysis of the Way to identify basic Blocks in Work [41], [86], [96] We divide the way to get basic blocks into two ways. One is to divide basic blocks from static CFG by static analysis of binary code before program execution. The other is a dynamic way to extract basic blocks by analyzing the transition instructions and transition rules of the program execution information.

Among these endeavors, BBB-CFI [41] proposed a lightweight dynamic and static combination method for basic block identification. This technique effectively minimizes the overhead associated with basic block identification and analysis. In particular, BBB-CFI stands out as a lightweight CFI defense mechanism aimed at defense code reuse attacks by verifying basic block information. Basic Block Boundary (BBB) refers to a linear instruction fragment featuring a single entry and exit point. The approach mandates that program control flow transitions exclusively from one BBB's exit to another's entry, ensuring control flow integrity. During program execution, BBB-CFI employs two lightweight identification techniques, namely Code-inspired Check and Data-inspired Check, to identify the BBB of the program. The Code-inspired Check efficiently identifies BBB by exploiting the continuity between a BBB's exit and entry points, using branch instructions as exit markers. Specifically, when the target instruction of a branch instruction is executed, a disassembly operation is performed starting from the current target instruction to obtain the assembly instruction of the program. If a branch instruction is found, it means that the current target instruction is a valid target instruction. The data-inspired Check, on the other hand, analyzes the program binary file before program execution.

Data-inspired Check involves scanning read-only data to identify jump tables and creating a bitmap based on them. Bitmap managed by the OS and stored in the secure kernel, the bitmap enables swift checks of whether the target address is a BBB entry point during execution. To execute this verification, security personnel employ the processor's instruction tracking module LBR, which retains the latest 16 or 32 branch instructions, monitoring the program's branching behavior. By combining these two lightweight detection techniques, the branch program pointer is confined to BBB entry addresses,

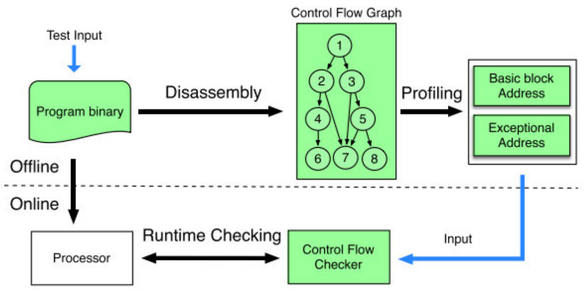


FIGURE 15. The CFI process is implemented in BBB-CFI according to the basic block verification: Offline and Online [41].

substantially reducing the availability of gadgets by 90%. However, it is important to note that the forward CFI implemented here is coarse-grained, as it does not discern which basic block to jump to during basic block identification.

Notably, the safeguarding of backward return addresses is entrusted to a shadow stack, with the added support for returning to the Exception Handler (EH). The EH, calculated offline, is stored in the processor's on-chip storage. During program execution, the exception handler's address is promptly retrieved for use. When compared to BB-CFI [96], FlowGuard [86], and, TypeArmer [97] the BBB-CFI implementation exhibits significantly lower execution overheads: less than 1%, 2.5%, and 3.8% respectively. BBB-CFI [41] own execution overhead stands at a mere 0.0003%, rendering it negligible.

Similar to the work of BBB-CFI [41], BB-CFI [96] is also implemented by using a mixture of static and dynamic recognition basic blocks. The BB-CFI workflow consists of a basic block information collection phase and an execution phase. In the basic block information collection phase, the difference from the way BBB-CFI uses statically generated CFG to extract basic blocks is that BB-CFI performs dynamic analysis of the program to obtain the CFG containing the target address of the indirect jump instructions that execute the program. The extracted CFG is then used to segment basic blocks, capturing the initial address of each basic block, the first basic block of each function, and the exception handling target address EH. The collected basic block information is stored in the Control-Flow Checker (CFC) and serves to validate target addresses of call, jmp, and ret instructions during program execution. During program execution, CFC enforces call and jmp instructions to jump to the initial instruction position within the basic block. For ret instructions, it verifies the target address of the return instruction based on the call-ret relationship. After executing a ret instruction, a comparison between the return address and the address stored in the Return Address Stack (RAS) determines its legitimacy. The enforcement of target addresses of branch instructions achieves fine-grained static CFI. The security effectiveness of BB-CFI is rigorously evaluated using the RIPE [89] framework, demonstrating its efficacy against ROP, JOP, and return-into-libc [12] attacks. Gadget reduction averages an impressive 99.38% across test programs. Furthermore, experimental execution overhead remains below 1%. In summary, BB-CFI [96] embodies robust security, minimal overhead, and fine-grained static CFI with support for multithreading. However, it's worth noting that BB-CFI necessitates alterations to the processor architecture and introduces an increase in program execution time during the off-line information gathering phase due to the dynamic CFG extraction approach.

Through the analysis of the above work based on the basic block verification Calculation implementation of CFI comparison depends on the CFG. Fig. 15 overall approach of BBB-CFI [41] is the basic block Verification Calculation to verify implementation of CFI in a representative way.

The process of basic block calculation and verification can be divided into two stages. In the first stage of offline, basic block information is processed, and this can involve calculating the hash value of the basic block or obtaining the first instruction's address for each basic block. This information about basic blocks is then stored.

During program execution, this information is used to verify whether the hash value of the basic block matches and whether the target of a branch instruction aligns with the first address of the basic block. However, this method of storing basic block information consumes processor memory. To mitigate this, certain approaches, like BBB-CFI [41], circumvent the need for storing basic block information. They achieve this by segmenting and verifying basic blocks within the binary code during program execution. The CFI approach, based on the calculation and verification of basic blocks as summarized in Table 7, effectively mitigates the availability of gadgets for potential attackers. This proactive measure hinders attackers from launching successful attacks by restricting branch jump instructions to only lead to the entry address of a basic block.

A comparison between CCFI-Cache [43] and BBB-CFI [41] reveals that BBB-CFI significantly outperforms CCFI-Cache in terms of both area and execution overheads, without necessitating the storage of basic block information. These findings suggest that future advancements in CFI implementations should prioritize methods that circumvent the need for storing basic block information.

V. ANALYZE AND DISCUSS

In this part, we define the security levels for some literatures mentioned in this paper. We define CFI as five levels according to the functions implemented by CFI. The higher the level, the better the security.

- Level I, coarse-grained forward or backward CFI.
- Level II, fine-grained forward or backward CFI.
- Level III, coarse-grained forward and backward CFI.
- Level IV, fine-grained forward and backward CFI.
- Level V, stateful CFI with fine-grained forward and backward.

According to our own defined security rules, we classified the schemes discussed in the preceding materials.

TABLE 7. Comparison of related work on basic block verification calculation.

	Depends on source code	Whether or not to rely on CFG	Basic block extraction method	Whether basic blocks need to be stored	Reduce gadgets	Perform overhead	Area overhead
CCFI-Cache [43]	Y	Y	Analyze the static CFG	Y	N/A	32%	10%
BBB-CFI [41]	Y	Y	Static binary analysis and jump table analysis	N	90%	<1%	<1%
BB-CFI [96]	Y	Y	Analyze the static CFG	Y	99.38%	<1%	0.02%
FlowGuard [86]	Y	Y	Analyze the static CFG	Y	N/A	3.8%	N/A
TypeArmer [97]	Y	Y	Dyninst binary analysis framework	Y	N/A	2.5%	N/A

It also indicates whether the scheme's protective capacity is enhanced by utilizing a shadow stack. Currently, paramount considerations in CFI implementation encompass heightened security, minimal execution overhead, and as far as possible reducing the use of processor resource utilization. We will comprehensively assess security, execution overhead, resource consumption, and other pertinent factors. By comparing their interrelations, we shall deliberate upon the merits and demerits of the six hardware-based CFI implementation methods, alongside their inherent hardware characteristics.

CFI serves to thwart code reuse attacks through real-time detection of program jump behavior. It accomplishes this by discerning aberrant program behavior during execution and comparing it against the expected normal behavior, thus safeguarding the processor against potential intrusion attempts.

The CFI scheme implemented by label verification is based on the concept of enforcing control over program jumps. Before program execution, a label is inserted before the valid target address within the program. During execution, it verifies whether the jump target label matches. To enhance CFI's security, this approach inserts multiple labels to differentiate legitimate target addresses for meticulous protection against code injection and code reuse attacks. This strategy exhibits robust security and effectively thwarts such attacks in domains requiring precise control flow integrity, including network devices and cloud computing servers. However, this technique requires access to the program's source code and involves code instrumentation for inserting numerous labels into the codebase [74].

The hardware specifications underlying label verification implementation of CFI (as outlined in Table 8) necessitate label registers and memory resources. For large programs, inserting labels increases both code size and hardware consumption, while frequent validation incurs certain execution overheads. Consequently, implementing this approach faces challenges in large-scale programs. Furthermore, label verification often analyzes static programs to determine locations that require instrumentation; thus, it does not verify legal targets of indirect jump instructions. In future research, efforts should be made to develop label verification methods that can verify target addresses of indirect jump instructions and expand the defense range achieved by label verification.

The initial phase of a code reuse attack, as outlined in Table 3, involves the identification of available gadgets. In this stage of the attack, guided by the concept of program agnosticism, experts propose instruction encryption technology. This technology utilizes PUF to generate security keys and incorporates lightweight algorithms such as Hash and AES to implement an encryption engine within the processor. It encrypts the legitimate target address to prevent attackers from tampering with instructions during program execution. Through this mechanism, the legitimate target address is encrypted to thwart potential manipulation by assailants. In contrast to label-based methods, this implementation approach minimizes the reliance on source code and instruction instrumentation tools, thereby facilitating the dynamic assignment of protective attributes. In comparison to the label method, this implementation approach does not rely on instruction instrumentation tools, enabling dynamic protection features. Additionally, the encryption of target instruction function calls in indirect jump instructions is achieved [36]. Consequently, integrating a processor-embedded encryption engine represents a promising avenue for future development. However, the inherent drawback of current encryption algorithms may result in misjudgment of specific code sequences during decryption. Moreover, the high complexity of codes and incomplete coverage can lead to false positives or false negatives [75], [76]. These factors pose threats to the security of CFI implemented through encryption algorithms.

Additionally, the intricate encryption and decryption algorithm in this verification scheme [76] introduces significant execution overhead, making it impractical. Moreover,

TABLE 8. Security level classification and evaluation of CFI scheme flow integrity protection.

Reference	Hardware feature	Shadow stack	Requires operating system support	Level of security	Evaluation
Label verification					
HCFI [49]	Register & Modifying processor architecture	Y	N	IV	Can handle recursion of any depth but depends on the source code.
HAFIX [39]	Label state memory & ISA	N	N	II	The utilization of processor resources is reduced, but the security level is low.
Davi [46]	Label State Table & ISA	N	N	IV	Processor resources are less utilized but depending on the source code require code instrumentation.
Sullivan [74]	LSS and LSR	N	Y	V	The security performance is good, but the Code size is large.
ABCFI [72]	FSM	N	N	III	Reduced Code size, dependent on source code.
Instruction encryption					
HCIC [42]	EDH PUF	N	N	III	It effectively reduces the deployment time of the scheme, but it needs to modify the processor architecture and has poor portability.
Qiu [75]	XOR PUF	N	N	III	The execution overhead and area overhead are ignored but the encryption algorithm is simple.
FH-CFI [36]	Hash PUF	N	N	IV	This is safer but requires new code blocks to be inserted.
LEA [76]	AES PUF	N	N	III	It uses a lightweight encryption algorithm, but there are false positives.
Stack edge detection					
StackGuard [78]	Register	N	Y	III	It can effectively prevent buffer overflow attacks, but it requires compiler modifications.
PUFCanary [32]	PUF TRNG	N	Y	IV	It effectively prevents the disclosure of the secret key, but it depends on the source code.
Zipper Stack [80]	Register AES	N	N	II	It removes the dependence on source code but incurs high execution overhead.
Li[70]	Register Hash	N	N	II	The overhead is low and the security is high, but the scheme is realized by simulation.
Instruction execution information tracking detection					
FastCFI [37]	PTM FPGA	N	N	V	The security is the highest but depends on the source code and requires additional FPGA resources.
Nile [50]	MU	Y	N	II	Code instrumentation is avoided, but the area overhead is high.
Liu [86]	IPT	N	Y	II	Improved security but requires specific hardware support.
Zgheib [48]	TE	Y	N	II	It avoids the modification of processor architecture and has good portability, but it can only defend against ROP attacks.
ACE-M [35]	MPU	N	N	IV	Fine-grained CFI is implemented, but it depends on source code and code instrumentation.
Sensitive data isolation					
FIXER [39]	Policy Matrix Mem and Shadow Stack Mem	Y	N	IV	It accelerates CFI detection and is easy to transplant, but it needs code instrumentation and additional FPGA resources.
TrustFlow-X [40]	TLB	Y	N	IV	Designing CFI from the software layer is highly secure but requires modifications to the processor architecture.
DangP [73]	Stack	Y	N	II	Parallel shadow stacks reduce execution overhead but require additional resources.
Meng [93]	Stack	Y	N	II	It reduces the execution overhead of traditional shadow stack but depends on source code and code instrumentation.
Basic block verification					
CCFI-Cache [43]	Cache	Y	N	IV	Parallel CFI detection reduces the execution overhead but requires the use of additional resources.
BBB-CFI [41]	Bitmap and Shadow stack	Y	N	III	Free from dependence on source code, enabling fast deployment. However, this scheme is a coarse-grained CFI with the risk of being bypassed.
BB-CFI [96]	RAS and TAB (TA buffer)	N	Y	IV	This scheme has high security and can verify the indirect jump target address, but it needs to modify the processor architecture.

implementing instruction encryption technology for CFI implementation often requires modifications to the processor architecture, thereby reducing its portability. Importantly, this solution is well suited for non-open-source applications or firmware as it cannot require source code modifications or additional instruction insertions. Future improvements can focus on optimizing and scrutinizing the encryption algorithm to mitigate encryption and decryption overheads, thus enhancing system performance.

The label and instruction encryption methodology enhances hardware functionalities to enable the identification of abnormal code execution during program runtime, with a heightened emphasis on software execution. Meanwhile, stack edge detection monitors the state of the stack during program execution, effectively thwarting stack overflow attacks and ROP attacks, thereby enhancing the protection of return addresses and local variables. Similar to the instruction encryption algorithm, stack edge detection frequently requires the utilization of PUFs for generating randomized keys. These keys are then combined with hardware-based encryption algorithms to secure "canary words" at the boundary of sensitive data within the stack. This strengthens the validation process against potential data modifications by attackers during program execution [39], necessitating adjustments to the structural configuration of the stack. These adjustments may involve source code analysis or incorporating additional hardware resources for stack restructuring [70]. Furthermore, due to frequent scrutiny of sensitive stack data, implementing stack edge detection incurs greater execution overhead compared to alternative strategies. Looking ahead, exploring and optimizing hardware support could potentially reduce this overhead.

Instruction trace detection allows for the analysis of instruction execution details during processor execution, independent of source code. By utilizing built-in instruction trace modules such as PTM [37] in ARM processors, PT [47] in Intel processors, and TE [48] in RISCV processors known for their open-source nature, this approach eliminates additional execution overhead and resource consumption associated with code instrumentation. It enables the recording of contextual program execution details and facilitates stateful program verification, thereby enhancing CFI security. Furthermore, by combining instruction trace detection with high-speed FPGA technology, it becomes possible to collect and analyze instruction execution information without modifying the processor pipeline, ensuring secure program operation and enabling rapid deployment of CFI security defense methods. As a result, this method offers exceptional flexibility and portability advantages. In light of evolving attack methods, instruction trace detection can effectively enhance FPGA-based CFI schemes making it suitable for systems requiring high security and real-time performance. Going forward, leveraging functional modules within a processor's instruction tracking capabilities is expected to emerge as a prominent area of research in the field of CFI.

Sensitive data isolation serves as a supplementary measure for enhancing security. This approach utilizes registers and memory repositories to securely store sensitive data, thereby enhancing the security of sensitive data within programs. Insights derived from Table 8 emphasize that, in contrast to alternative CFI implementations, sensitive data isolation primarily relies on a shadow stack to enforce return address protection. However, it is important to acknowledge that substantial storage resources may be consumed if extensive data isolation is required [40]. Nevertheless, sensitive data isolation can serve as an ancillary avenue for synergizing with other approaches to fortify CFI security. For instance, by integrating CFI with techniques like instruction encryption and stack edge detection, isolation space can be established to safeguard pivotal cryptographic keys and critical data related to CFI implementation [38]. The application domain of this CFI implementation extends to financial systems and encrypted communication platforms, ensuring the protection of vital data and system security. To further optimize the scheme of sensitive data isolation in the future, hardware costs can be reduced while combining various methods of providing comprehensive control flow integrity protection. The approach of basic block verification involves monitoring transitions between basic blocks during program execution.

Analogous to the concept of label verification, basic block verification enforces a constraint where the program exclusively progresses to the entry point of a basic block [97]. In contrast to the stack edge detection method, basic block verification effectively curtails the frequency of CFI verification, consequently mitigating processor execution overhead to a certain extent. This implementation method of basic block verification uses a static method to extract the basic blocks from the CFG of the source code [45], and can also identify the basic blocks according to the rules of the basic blocks during program execution [97]. Basic blocks obtained before program execution or the data associated with processed basic blocks are preloaded into the processor's memory, facilitating rapid verification of target program validity. The implementation scheme of basic block verification can ensure the security of the stored basic block data combined with Sensitive data isolation. In comparison to alternative CFI implementation approaches, this scheme significantly relies on source code and CFG, which raises security concerns stemming from CFG extraction challenges and coverage gaps. Future CFI implementations predicated on basic blocks should center on refining basic block monitoring mechanisms and addressing hardware storage concerns to reduce dependence on CFG.

VI. CONCLUSION AND OUTLOOK

Hardware-based CFI techniques have made significant strides in enhancing system security and mitigating control-flow hijacking attacks. In this study, we categorize the current hardware implementations of CFI into six distinct categories, each offering a unique perspective on different aspects of control-flow integrity.

Label verification ensures the accuracy of labels associated with program instructions by validating them during runtime, thereby mitigating potential vulnerabilities to code injection or unauthorized modifications.

The encryption of instructions serves as a protective measure against unauthorized modifications, thereby enhancing the program's security with an additional layer of defense.

The stack edge detection mechanism monitors the integrity of the stack to identify any abnormal behavior associated with function call and return operations within the stack memory region. This effectively enables the detection of potential buffer overflow or underflow attacks that could potentially lead to control-flow hijacking.

The tracking of instruction execution information enables the analysis and validation of control transfers during runtime, facilitating early detection of any anomalies in the program's flow for enhanced academic and professional purposes.

Sensitive data isolation prevents attackers from manipulating control flows through direct manipulation of important variables or data structures by isolating critical data structures within secure enclaves or protected memory regions.

Basic block verification verifies the integrity and correctness of basic blocks within a program's code structure sequences of consecutive instructions without any branching.

Drawing insights from the comparative analysis of these six implementation strategies, we derive the following discerning conclusions:

- The implementation of the CFI scheme through label verification depends on the source code but the scheme can minimize the use of hardware resources.
- The instruction execution trace module combined with additional FPGA resources can enhance the portability and flexibility of hardware implementation of CFI.
- Although the way of instruction encryption and instruction tracking require a lot of extra area resources, the execution overhead is very low.
- Augmenting CFI Security through Shadow Stack in Sensitive Data Isolation: Incorporating a shadow stack within the sensitive data isolation method not only facilitates the realization of stateful CFI but also elevates the overall security of the CFI mechanism.

Ultimately, we propose that the incorporation of lightweight encryption algorithms into the processor through instruction encryption facilitates CFI in mitigating additional security vulnerabilities arising from code instrumentation and enhances the reliability of CFI itself. Moreover, by employing instruction execution information tracking and independent FPGA resources, not only can verification speed be enhanced but also the upgradability of the CFI solution can be enabled, thereby augmenting its security and practicality. In future advancements of CFI, we assert that instruction encryption and instruction execution information tracking represent two prominent avenues for implementing robust CFI solutions.

REFERENCES

[1] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? An analysis of topics and trends in stack overflow," *Empirical Softw. Eng.*, vol. 19, no. 3, pp. 619–654, Jun. 2014.

[2] E. Bosman and H. Bos, "Framing signals—A return to portable shellcode," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 243–258.

[3] K. Lhee and S. J. Chapin, "Buffer overflow and format string overflow vulnerabilities," *Software: Pract. Exper.*, vol. 33, no. 5, pp. 423–460, Mar. 2003.

[4] W. Wu, Y. Chen, X. Xing, and W. Zou, "Kepler: Facilitating control-flow hijacking primitive evaluation for Linux kernel vulnerabilities," in *Proc. 28th USENIX Secur. Symp. (USENIX Secur.)*, 2019, pp. 1187–1204.

[5] R. Riley, X. Jiang, and D. Xu, "An architectural approach to preventing code injection attacks," *IEEE Trans. Depend. Secure Comput.*, vol. 7, no. 4, pp. 351–365, Oct. 2010.

[6] M. Kayaalp, M. Ozsoy, N. A. Ghazaleh, and D. Ponomarev, "Efficiently securing systems from code reuse attacks," *IEEE Trans. Comput.*, vol. 63, no. 5, pp. 1144–1156, May 2014.

[7] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proc. 17th ACM Conf. Comput. Commun. Secur.*, Oct. 2010, pp. 559–572.

[8] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 574–588.

[9] T. Zhang, M. Cai, D. Zhang, and H. Huang, "SeBROP: Blind ROP attacks without returns," *Frontiers Comput. Sci.*, vol. 16, no. 4, Jan. 2022, Art. no. 164818.

[10] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proc. 6th ACM Symp. Inf., Comput. Commun. Secur.*, Hong Kong, Mar. 2011, pp. 30–40.

[11] P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, and X. Yin, "Automatic construction of jump-oriented programming shellcode (on the x86)," in *Proc. 6th ACM Symp. Inf., Comput. Commun. Secur.*, Mar. 2011, pp. 20–29.

[12] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, Oct. 2007, pp. 552–561.

[13] N. Stojanovski, M. Gusev, D. Gligoroski, and S. J. Knapskog, "Bypassing data execution prevention on MicrosoftWindows XP SP2," in *Proc. 2nd Int. Conf. Availability, Rel. Secur. (ARES)*, 2007, pp. 1222–1226.

[14] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 601–615.

[15] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proc. ACM Conf. Comput. Commun. Secur.*, Oct. 2012, pp. 157–168.

[16] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 1–40, Oct. 2009.

[17] M. Almgren, V. Gulisano, and F. Maggi, "Fine-grained control-flow integrity through binary hardening," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, Milan., Italy, 2015, pp. 144–164.

[18] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Mar. 2016, pp. 179–194.

[19] C. Tice, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Proc. USENIX Secur. Symp.*, 2014, pp. 941–955.

[20] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *Proc. 6th ACM Symp. Inf., Comput. Commun. Secur.*, Mar. 2011, pp. 40–51.

[21] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 559–573.

[22] R. M. Farkhani, S. Jafari, S. Arshad, W. Robertson, E. Kirda, and H. Okhravi, "On the effectiveness of type-based control flow integrity," 2018, *arXiv:1810.10649*.

[23] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 745–762.

[24] A. Biondo, M. Conti, and D. Lain, "Back to the epilogue: Evading control flow guard via unaligned targets," in *Proc. NDSS*, San Diego., CA, USA, 2018, pp. 1–15.

[25] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 575–589.

[26] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 952–963.

[27] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *Proc. USENIX Secur. Symp.*, 2014, pp. 385–399.

[28] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proc. USENIX Secur. Symp.*, 2014, pp. 401–416.

[29] Q. Hao, Z. Zhang, D. Xu, J. Wang, J. Liu, J. Zhang, J. Ma, and X. Wang, "A hardware security-monitoring architecture based on data integrity and control flow integrity for embedded systems," *Appl. Sci.*, vol. 12, no. 15, p. 7750, Aug. 2022.

[30] S. Park, D. Kang, J. Kang, and D. Kwon, "Bratter: An instruction set extension for forward control-flow integrity in RISC-V," *Sensors*, vol. 22, no. 4, p. 1392, Feb. 2022.

[31] M. C. Park and D. H. Lee, "BGCFI: Efficient verification in fine-grained control-flow integrity based on bipartite graph," *IEEE Access*, vol. 11, pp. 4291–4305, 2023.

[32] D. Jung, M. Kim, J. Jang, and B. B. Kang, "Value-based constraint control flow integrity," *IEEE Access*, vol. 8, pp. 50531–50542, 2020.

[33] R. de Clercq, J. Götzfried, D. Übler, P. Maene, and I. Verbauwhede, "SOFIA: Software and control flow integrity architecture," *Comput. Secur.*, vol. 68, pp. 16–35, Jul. 2017.

[34] Y. Wang, Q. Li, Z. Chen, P. Zhang, G. Zhang, and Z. Shi, "BCI-CFI: A context-sensitive control-flow integrity method based on branch correlation integrity," *Inf. Softw. Technol.*, vol. 136, Aug. 2021, Art. no. 106572.

[35] S. Lee and J. Cho, "ACE-M: Automated control flow integrity enforcement based on MPUs at the function level," *Electronics*, vol. 11, no. 6, p. 912, Mar. 2022.

[36] A. Fu, W. Ding, B. Kuang, Q. Li, W. Susilo, and Y. Zhang, "FH-CFI: Fine-grained hardware-assisted control flow integrity for ARM-based IoT devices," *Comput. Secur.*, vol. 116, May 2022, Art. no. 102666.

[37] L. Feng, J. Huang, J. Hu, and A. Reddy, "FastCFI: Real-time control-flow integrity using FPGA without code instrumentation," *ACM Trans. Design Autom. Electron. Syst.*, vol. 26, no. 5, pp. 1–39, Sep. 2021.

[38] M. C. Park and D. H. Lee, "Random CFI (RCFI): Efficient fine-grained control-flow integrity through random verification," *IEEE Trans. Comput.*, vol. 70, no. 5, pp. 733–745, May 2021.

[39] A. De, A. Basu, S. Ghosh, and T. Jaeger, "Hardware assisted buffer protection mechanisms for embedded RISC-V," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 12, pp. 4453–4465, Dec. 2020.

[40] C. Bresch, D. Hély, R. Lysecky, S. Chollet, and I. Parissis, "TrustFlow-X: A practical framework for fine-grained control-flow integrity in critical systems," *ACM Trans. Embedded Comput. Syst.*, vol. 19, no. 5, pp. 1–26, Sep. 2020.

[41] W. He, S. Das, W. Zhang, and Y. Liu, "BBB-CFI: Lightweight CFI approach against code-reuse attacks using basic block information," *ACM Trans. Embedded Comput. Syst.*, vol. 19, no. 1, pp. 1–22, Jan. 2020.

[42] J. Zhang, B. Qi, Z. Qin, and G. Qu, "HCIC: Hardware-assisted control-flow integrity checking," *IEEE Internet Things J.*, vol. 6, no. 1, pp. 458–471, Feb. 2019.

[43] J.-L. Danger, A. Facon, S. Guilley, K. Heydemann, U. Kühne, A. Si Merabet, and M. Timbert, "CCFI-cache: A transparent and flexible hardware protection for code and control-flow integrity," in *Proc. 21st Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2018, pp. 529–536.

[44] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-assisted data-flow isolation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 1–17.

[45] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *Proc. 51st ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2014, pp. 1–6.

[46] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "HAFIX: Hardware-assisted flow integrity eXtension," in *Proc. 52nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2015, pp. 1–6.

[47] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "PT-CFI: Transparent backward-edge control flow violation detection using Intel processor trace," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2017, pp. 173–184.

[48] A. Zgheib, O. Potin, J.-B. Rigaud, and J.-M. Dutertre, "A CFI verification system based on the RISC-V instruction trace encoder," in *Proc. 25th Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2022, pp. 456–463.

[49] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "HCFI: hardware-enforced control-flow integrity," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2016, pp. 38–49.

[50] L. Delshadtehrani, S. Eldridge, S. Canakci, M. Egele, and A. Joshi, "Nile: A programmable monitoring coprocessor," *IEEE Comput. Archit. Lett.*, vol. 17, no. 1, pp. 92–95, Jan. 2018.

[51] S. Sayeed, H. Marco-Gisbert, I. Ripoll, and M. Birch, "Control-flow integrity: Attacks and protections," *Appl. Sci.*, vol. 9, no. 20, p. 4229, Oct. 2019.

[52] S. Kumar, D. Moolchandani, and S. R. Sarangi, "Hardware-assisted mechanisms to enforce control flow integrity: A comprehensive survey," *J. Syst. Archit.*, vol. 130, Sep. 2022, Art. no. 102644.

[53] L. V. Davi, "Code-reuse attacks and defenses," Ph.D. dissertation, Technische Universität Darmstadt, Germany, 2015.

[54] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib(C)," in *Proc. Annu. Comput. Secur. Appl. Conf.*, Dec. 2009, pp. 60–69.

[55] S. Jeong, J. Hwang, H. Kwon, and D. Shin, "A CFI countermeasure against GOT overwrite attacks," *IEEE Access*, vol. 8, pp. 36267–36280, 2020.

[56] C. Canella, "A systematic evaluation of transient execution attacks and defenses," in *Proc. USENIX Secur. Symp.*, 2019, pp. 249–266.

[57] P. Kocher, "Spectre attacks: Exploiting speculative execution," *Commun ACM*, vol. 63, no. 7, pp. 93–101, Jun. 2020.

[58] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, and R. Strackx, "Meltdown: Reading kernel memory from user space," *Commun. ACM*, vol. 63, no. 6, pp. 46–56, May 2020.

[59] S. El Sherei. *Return to Libc*. Accessed: Aug. 29, 2023. [Online]. Available: https://www.exploit-db.com/docs/english/28553-linux-classic-return-to-libc-&-return-to-libc-chaining-tutorial.pdf

[60] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, p. 26.

[61] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi, "Address oblivious code reuse: On the effectiveness of leakage-resilient diversity," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.

[62] H. Oh, Y. Cho, and Y. Paek, "A metadata-driven approach to efficiently detect code-reuse attacks on ARM multiprocessors," *J. Supercomput.*, vol. 77, no. 7, pp. 7287–7314, Jul. 2021.

[63] M. Zhang and R. Sekar, "Control flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks," in *Proc. 31st Annu. Comput. Secur. Appl. Conf.*, Dec. 2015, pp. 91–100.

[64] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 27–30.

[65] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically enforced control flow integrity," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 941–951.

[66] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 1470–1486.

[67] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of path-sensitive control security," in *Proc. USENIX Secur. Symp.*, 2017, pp. 131–148.

[68] B. Niu and G. Tan, "Per-input control-flow integrity," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 914–926.

[69] N. Burow, D. McKee, S. A. Carr, and M. Payer, "CFIXX: Object type integrity for C++," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–14.

[70] J. Li, Q. Xu, Y. Li, L. Chen, G. Shi, and D. Meng, "Efficient return address verification based on dislocated stack," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3398–3407, Nov. 2020.

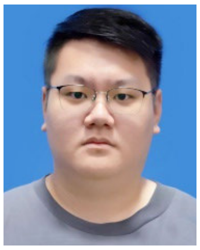
[71] A. Y. C. Zhu, W. Q. Yan, and R. Sinha, "ROP defense using trie graph for system security," *Int. J. Digit. Crime Forensics*, vol. 13, no. 6, pp. 1–12, 2021.

[72] J. Li, L. Chen, G. Shi, K. Chen, and D. Meng, "ABCFI: Fast and lightweight fine-grained hardware-assisted control-flow integrity," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3165–3176, Nov. 2020.

[73] T. H. Y. Dang, P. Maniatis, and D. Wagner, ''The performance cost of shadow stacks and stack canaries,'' in *Proc. 10th ACM Symp. Inf., Comput. Commun. Secur.*, Apr. 2015, pp. 555–566.
[74] D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin, ''Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity,'' in *Proc. 53rd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2016, pp. 1–6.
[75] P. Qiu, Y. Lyu, D. Zhai, D. Wang, J. Zhang, X. Wang, and G. Qu, ''Physical unclonable functions-based linear encryption against code reuse attacks,'' in *Proc. 533rd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2016, pp. 1–6.
[76] P. Qiu, Y. Lyu, J. Zhang, D. Wang, and G. Qu, ''Control flow integrity based on lightweight encryption architecture,'' *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 7, pp. 1358–1369, Jul. 2018.
[77] J. Zhu, W. Zhou, Z. Wang, D. Mu, and B. Mao, ''DiffGuard: Obscuring sensitive information in Canary based protections,'' in *Proc. Int. Conf. Secur. Privacy Commun. Syst.*, 2018, pp. 738–751.
[78] C. Cowan, ''StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks,'' in *Proc. USENIX Secur. Symp.*, 1998, pp. 63–78.
[79] R. K. Shrivastava, K. J. Concessao, and C. Hota, ''Code tamper-proofing using dynamic canaries,'' in *Proc. 25th Asia–Pacific Conf. Commun. (APCC)*, Nov. 2019, pp. 238–243.
[80] J. Li, ''Zipper stack: Shadow stacks without shadow,'' in *Proc. Eur. Symp. Res. Comput. Secur.*, 2020, pp. 338–358.
[81] P. B. Ghewari, J. Patil, and A. B. Chougule, ''Efficient hardware design and implementation of AES cryptosystem,'' *Int. J. Eng. Sci. Technol.*, vol. 2, no. 3, pp. 213–219, 2010.
[82] Y. Xia, Y. Liu, H. Chen, and B. Zang, ''CFIMon: Detecting violation of control flow integrity using performance counters,'' in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2012, pp. 1–12.
[83] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, ''Using CoreSight PTM to integrate CRA monitoring IPs in an ARM-based SoC,'' *ACM Trans. Design Autom. Electron. Syst.*, vol. 22, no. 3, pp. 1–25, Jul. 2017.
[84] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, ''Integration of ROP/JOP monitoring IPs in an ARM-based SoC,'' in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2016, pp. 331–336.
[85] X. Ge, W. Cui, and T. Jaeger, ''GRIFFIN: Guarding control flows using Intel processor trace,'' *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 585–598, May 2017.
[86] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, ''Transparent and efficient CFI enforcement with Intel processor trace,'' in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 529–540.
[87] J. Lee, I. Heo, Y. Lee, and Y. Paek, ''Efficient security monitoring with the core debug interface in an embedded processor,'' *ACM Trans. Design Autom. Electron. Syst.*, vol. 22, no. 1, pp. 1–29, Jan. 2017.
[88] Z. Guo, R. Bhakta, and I. G. Harris, ''Control-flow checking for intrusion detection via a real-time debug interface,'' in *Proc. Int. Conf. Smart Comput. Workshops*, Hong Kong, Nov. 2014, pp. 87–92.
[89] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, ''RIPE: Runtime intrusion prevention evaluator,'' in *Proc. 27th Annu. Comput. Secur. Appl. Conf.*, Dec. 2011, pp. 41–50.
[90] J. L. Henning, ''SPEC CPU2006 benchmark descriptions,'' *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.
[91] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, B. C. Pierce, and A. DeHon, ''Architectural support for software-defined metadata processing,'' in *Proc. 20th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2015, pp. 487–502.
[92] K. Asanovic, ''The rocket chip generator,'' EECS Dept., Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. Ucb/Eecs-2016, 2016, pp. 2–6, vol. 17.
[93] X. Meng, B. Chamith, and R. Newton, ''Profile-guided, multi-version binary rewriting,'' 2020, *arXiv:2002.07748*.
[94] Y. Bai, ''ARM memory protection unit (MPU),'' in *Practical Microcontroller Engineering with ARM Technology*. Hoboken, NJ, USA: Wiley, 2016, pp. 951–974.
[95] C. Lattner, ''LLVM and Clang: Next generation compiler technology,'' in *Proc. BSD Conf.*, vol. 5, 2008, pp. 1–20.
[96] S. Das, W. Zhang, and Y. Liu, ''A fine-grained control flow integrity approach against runtime memory attacks for embedded systems,'' *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 11, pp. 3193–3207, Nov. 2016.
[97] V. van der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, ''A tough call: Mitigating advanced code-reuse attacks at the binary level,'' in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 934–953.

SENYANG LI was born in Bozhou, Anhui, China, in 1997. He received the B.S. degree in communication engineering from Huainan Normal University, China, in 2021. He is currently pursuing the master's degree in the field of embedded system security with the School of Electronic Information Engineering, Shandong University of Science and Technology, Qingdao, Shandong. His research interests include the areas of embedded system security and computing-level system security.

WEIKE WANG (Member, IEEE) received the B.S. degree in electronic information and the M.S. degree in circuits and systems from the Shandong University of Science and Technology, Qingdao, China, in 2010 and 2013, respectively, and the Ph.D. degree in microelectronics and solid-state electronics from Beihang University, Beijing, China, in 2019. He is currently a Lecturer with the College of Electronic and Information Engineering, Shandong University of Science and Technology. His research interests include the areas of SoC design, hardware security, integrated circuit and FPGA design, and embedded systems.



WENXIN LI (Graduate Student Member, IEEE) was born in Jinan, Shandong, China, in 2000. He received the B.S. degree in electronic information science and technology from the Shandong University of Science and Technology, Qingdao, China, in 2022, where he is currently pursuing the master's degree in the field of embedded system security. His research interests include RISC-V embedded systems and control flow integrity.

DEXUE ZHANG received the B.S. and Ph.D. degrees from the University of Science and Technology of China, Hefei, China, in 2000 and 2006, respectively. He is currently an Associate Professor with the College of Electronic and Information Engineering, Shandong University of Science and Technology, Qingdao, China. His research interests include SoC design, network on chip design, and many-core processors design.