Hardware-Assisted Control-Flow Integrity Enhancement for IoT Devices

Weiyi Wang[†] Lang Feng[*] Zhiguo Shi[†] Cheng Zhuo[†‡] Jiming Chen[†]
[†]Zhejiang University, China [*]Sun Yat-sen University, China
[‡]Key Laboratory of Collaborative Sensing and Autonomous Unmanned Systems of Zhejiang Province, China
czhuo@zju.edu.cn

***Abstract*—Internet of Things (IoT) devices face an escalating threat from code reuse attacks (CRAs) as they can reuse existing code for malicious purpose. Thus a practical cost-effective Control-Flow Integrity (CFI) mechanism for IoT devices is urgently needed. However, existing CFI solutions suffer from impracticalities, including high performance overhead and a heavy reliance on offline perfect Control-Flow Graph (CFG) generation. To tackle these challenges, we propose a fine-grained dependable CFI scheme for IoT devices that real-time updates the CFG of devices. We evaluate the implementation on RISC-V architectures and the results show that our CFI scheme provides both backward- and forward-edge protection with almost no performance overhead in the case of fixed CFG, negligible power overhead, and low hardware overhead. Compared to the current hardware-assisted CFI designs, our design eliminates the dependence on the offline perfect CFG generation and performs real-time CFG updating for better practicality.**

I. Introduction

Internet-of-Things (IoT) edge devices play pivotal roles across various sectors of today's world [1], while they are often designed using memory-unsafe programming languages (like C and C++) and simple hardware architectures, which often lack comprehensive security measures [2]. Due to these software and hardware deficiencies, IoT devices, if unprotected, are susceptible to an extensive range of attacks that could substantially disrupt modern life. Code reuse attacks (CRAs), including return-oriented programming (ROP) [3] and jump-oriented programming (JOP) [4], stand out as significant threats. These attacks redirect a target program's control flow maliciously by reusing existing code sequences without injecting new code.

Although techniques like Address Space Layout Randomization (ASLR) [5] aim to counter CRAs, it is not a trivial task [6]. To more robustly secure the control flow, Control-Flow Integrity (CFI) has emerged as a promising alternative. Introduced by Abadi *et al.* [7], CFI ensures that a program at runtime follows a predefined legitimate Control-Flow Graph (CFG). This CFG models permissible control flow transitions, enabling the system to identify and halt unexpected behaviors.

Researchers have explored CFI in various aspects, primarily using software-focused implementations. For example, reference [7] integrates runtime check code preceding branch instructions, ensuring control flows align with CFG constraints. However, compiler optimizations can unintentionally leak information from registers containing CFI-related data into memory [8]. With enhanced policy precision, these methods can also degrade performance, particularly concerning for edge devices.

Hardware-assisted CFI offers a potential solution to these challenges. With specialized hardware designs, these methods can achieve faster responses and bolstered security. A popular component in many hardware-based CFIs is the shadow stack [9], [10]. While the shadow stack efficiently protects the backward-edge, it lacks adequate forward-edge protection. Some efforts, such as HCFI [10] and Sullivan *et al.* [11], offer forward-edge protection by adapting Abadi's label-based approach in hardware. However, these adaptations may introduce performance overheads and necessitate code instrumentation. Table-based work [12] use a table of permissible branches inside the core, incurring considerable area and power overheads. Alternative strategies involve Instruction Set Randomization (ISR) [13], with solutions like SOFIA [14] encrypting instructions in new CPU pipeline stages, although these too can drastically impact performance. FastCFI [15] showcases precise CFI mechanisms with minimal performance overhead, while it cater to complex, high-performance systems rather than IoT devices. Other notable efforts have applied machine learning to CFI [16], [17], which fall short in real-time checks.

Despite individual shortcomings, these fine-grained CFI techniques share two general drawbacks:

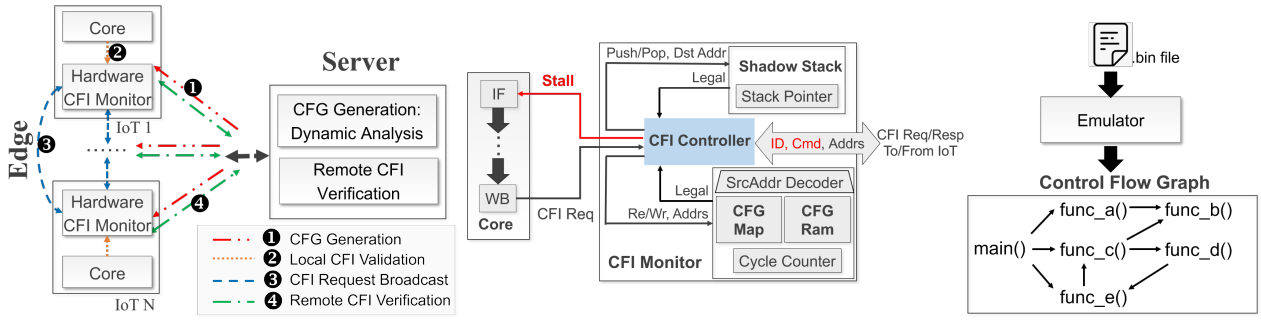
- All presume a perfectly precise CFG, yet *achieving such accuracy offline remains unresolved*. This makes their application to real-world IoT systems problematic.
- While these methods address the CFI issue, they introduce performance, area, and power overheads that often *render them unsuitable for budget-conscious IoT edge devices*.

Thus, a truly efficient CFI mechanism tailored for IoT edge devices remains elusive, emphasizing the need for an innovative, cost-effective solution.

In this paper, we introduce **a novel hardware-assisted CFI solution tailored specifically for IoT devices**. By leveraging the inherent interconnectivity of the IoT framework and including hardware CFI monitor into the IoT device, we dynamically maintained the CFG ram within each device. To further ensure robustness and accuracy, we've incorporated a high-efficiency lightweight Neural Network (NN) model, which is hosted on a main server as a top-tier remote CFI verification. Such an approach addresses the challenges previously mentioned, with a notable solution to the dependency on offline perfect CFG generation as well as runtime performance overheads. The following summarizes our contributions:

- We propose a novel hardware-assisted CFI design for IoT devices that overcomes the shortcomings of conventional CFI methods, which emphasizes dynamic CFGRam maintenance, eliminating the need for offline perfect CFG generation.

979-8-3503-4859-0/DATE24/© 2024 EDAA



(a) Workflow of the system (b) Hardware design of CFI monitor on the edge (c) Dynamic analysis on the server

Fig. 1. Overview of the proposed system: (a) Workflow of the systeym; (b) Hardware design of CFI monitor on the edge; (c) Dynamic analysis on the server.

- Given the limitations of traditional CFG generation such as high complexity and low accuracy, we utilize dynamic analysis to enhance the precision of initial CFG.
- To ensure the system robustness, we further deploy a NN hosted in the server to undertake top-tier remote CFI verification.

we have implemented the proposed system on different RISC-V architectures using FPGAs and then compared to the prior state-of-the-art works [10]–[12], [14], [15]. Experimental results show the design achieves almost no performance overhead and consumes only 2-3% additional resources, while it can achieve impressive accuracy rate of 0.983 in terms of control flow classification with a marginal false positive score of 0.033.

II. Preliminaries

Threat Model: Our model assumes attackers aim to alter an edge device's control flow to execute arbitrary code or leak private data. They can manipulate data, read code segments due to device defects, and possess knowledge of common security mechanisms like ASLR. This model doesn't consider code injection or inherently malicious programs.

Assumptions: The system is designed for simpler IoT devices, such as temperature controllers and actuators. We assume these devices *run singular, uncomplicated programs that don't self-modify*. Our hardware CFI monitor, discussed in Section III, is inaccessible externally, and the network transmitting CFI data remains secure and encrypted. These assumptions align with prevalent CFI research principles.

Design objective: Our approach is desired to be a cost-effective CFI for IoT edge devices, which don't degrade performance while accounting for practical security needs.

III. Proposed System Design

A. Workflow Overview

We introduce a hardware-assisted CFI method that allows for real-time updates of the CFG stored in each device. As illustrated in Fig. 1(a), our system follows four primary steps: *CFG generation*, *local CFI validation*, *CFI request broadcast* and *remote CFI verification*.

❶ **CFG Generation:** When a device connects to the IoT system, the server immediately generates a CFG through dynamic analysis (Fig. 1(c)) and sends it to the IoT device. This initial CFG is then stored in the device's CFI monitor.

❷ **Local CFI Validation:** As depicted in Fig. 1(b), if the CPU comes across a jump instruction, it checks with the hardware CFI monitor using address information.

❸ **CFI Request Broadcast:** If the previous step fails, the CFI monitor stalls the CPU pipeline (as indicated by the red arrow in Fig. 1(b)) and shares the address details with peer devices running the same program.

❹ **Remote CFI Verification:** If the previous step is unsuccessful, the device sends a log packet to the server. The server processes this log data through a pre-trained NN. Depending on the results, if the jump is deemed valid, the device updates its monitor and continues operations. If not, an alert is triggered, necessitating additional investigation.

The subsequent subsections will detail these processes and design details on both edge and server.

B. Proposed Design on the Edge

IoT devices on the edge need to conduct ❷ *local CFI validation* and ❸ *CFI request broadcast* in Fig. 1(a). The overall hardware design of the proposed CFI monitor is shown in Fig. 1(b). It is noted that forward-edge (i.e., indirect jump) protection counters Jump-Oriented Programming (JOP) attacks that exploit code sequences ending in indirect jump instructions. Moreover, runtime CFI checks aren't needed for direct jumps, as their destinations are fixed at compile time. The CFGRam and CFGMap structures, shown in Fig. 1(b), validate every indirect jump. Built in SRAM, they feature a source address decoder, a cycle counter, and supporting circuits. The stall line in Fig. 1(b) is typically low, raising only during potential CFI violations to prevent pipeline stalls from SRAM latency. Entries of CFGMap and CFGRam are birthed through dynamic analysis, detailed in Section III-C. Together, CFGMap and CFGRam store CFG details.

Allocating fixed space for every indirect jump, as in FIXER [12], is inefficient. Instead, we map access via CFGMap. Jump source address decoding determines the access index. CFGMap lines consist of 32 bits, subdivided into *Start* and *End* sections, further split into *Head* and *Tail* blocks. The 10-bit *Head* defines destination indices in CFGRam, with the 6-bit *Tail* doing the same from the opposite end. Each CFGRam line, at 64 bits, notes four valid destination addresses. The CFGRam size hinges on the *Head* block's bit count, totaling $2^{10} = 1024$ entries. TailRegister tracks the number of existing tail entries. Fig. 2(c) illustrates destination retrieval using source address decoding.

The CFI controller issues CFGMap and CFGRam's read/write operations. It processes core requests and IoT network interactions, which can be abstracted into six types: re-

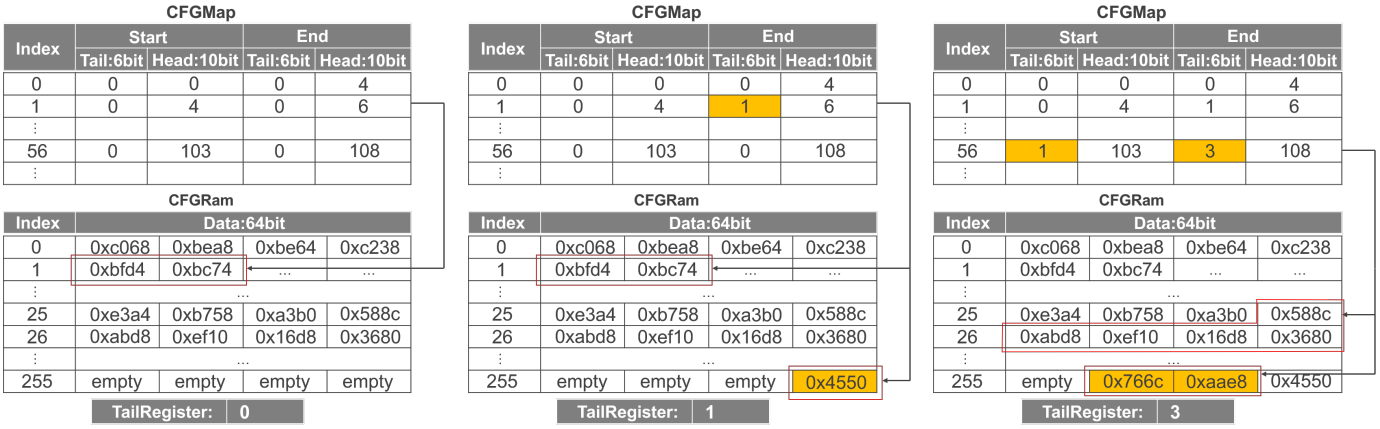


Fig. 2. Workflow of CFGMap and CFGRam: Red-boxed blocks indicate the destinations for a specific source address; Orange blocks denote new entries.

quest from the core (core), initialization (init), halt, supplement (sup), req broadcast (req_b), and search fail (fail). Alongside the received ID in Fig. 1(b), these commands direct CFGMap and CFGRam's read and write signals. The logic is presented in Table I. The CFI controller also sends commands, including sup, req_b, fail and request to server (req_s).

TABLE I
CFGMAP AND CFGRAM READ/WRITE SIGNAL GENERATION LOGIC: '0' REPRESENTS READ SIGNAL AND '1' REPRESENTS WRITE SIGNAL.

	core	init	halt	sup	req_b	fail
ID==SelfID	0	1	\	1	\	\
ID!=SelfID	\	\	\	\	0	\

Fig. 2(a) and Fig. 2(b) delve deeper into the process of *local CFI validation* and *CFI request broadcast*. When decoding an indirect jump instruction, the CPU sends an address pair to the CFI controller. This controller quickly issues a read request to CFGMap and CFGRam. If the address pair isn't found or is illegal (the latter being likelier), the CFI monitor temporarily stalls the CPU pipeline and broadcasts the address pair to other devices running the same program. A broadcast query's failure, signaled by either a timeout or unanimous negative feedback, necessitates further server validation. If the jump is verified, the address pair gets stored in CFGRam, and other devices can swiftly recognize it and proceed. This method, aside from rare initial lookup failures, leaves the CPU pipeline unaffected, ensuring performance and making CFGMap and CFGRam invisible to software. It promotes device communication on the IoT network, eliminating the need for a flawless starting CFG.

As for backward-edge (i.e., return) protection, attackers can manipulate memory vulnerabilities such as buffer overflows by injecting harmful data to overwrite a function's return address. According to [10], [18], for enhanced backward-edge security, the proposed approach introduces a unique non-memory-mapped shadow stack, which relies on the core's decode information instead of an ISA extension. This crucial distinction ensures that the proposed approach incurs no performance overhead. This stack is fashioned as a non-memory-mapped register file with a 32-bit width and a depth of 32, positioned within the CFI monitor as illustrated in Fig. 1(b). This shadow stack's design is rooted in the principle that a function's return

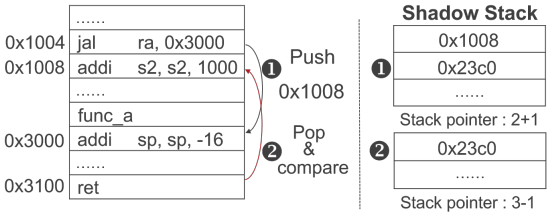


Fig. 3. Workflow of shadow stack in the proposed CFI monitor.

should align with its most recent call. The procedural steps to detect a backward-edge CFI violation using this stack are depicted in Fig. 3.

Aligning with the RISC-V calling conventions [19], the functions employ the *jal* and *jalr* instructions, using the *x1/x5* register to store their return address. As the function call's return address is saved in the *x1/x5* register during the write-back stage, the CFI controller issues a push command, which also pushes this address onto the shadow stack. Simultaneously, the stack pointer advances by one unit. Given that the *x1/x5* register might be repurposed during function execution, compilers save its value on the main function stack—a tangible segment of memory. Notably, malicious entities can overwrite this memory-stored value. On function return, the address from the function stack reloads into the *x1/x5* register. It's immediately juxtaposed with the shadow stack's popped value. A match allows the pipeline to continue smoothly. Conversely, a mismatch triggers a violation alert, halting execution.

C. Proposed Design on the Server

On the server side, it needs to conduct the tasks of ❶ *CFG generation* and ❹ *remote CFI verification* of Fig. 1(a). Traditional static analysis struggles with indirect jumps' precision [20]. We utilize dynamic analysis on server side (as shown in Fig. 1(c)) to generate initial CFG, running tests in the NEMU [21] emulator. By altering NEMU's code, we record every indirect jump's destinations during emulation. After running, each address pair (source, destination) is dispatched to its relevant device. Yet, dynamic analysis isn't flawless. When destinations depend on inputs, some address pairs, though genuine, might be absent from the CFG. Hence, we enhance accuracy of whole system using machine learning below.

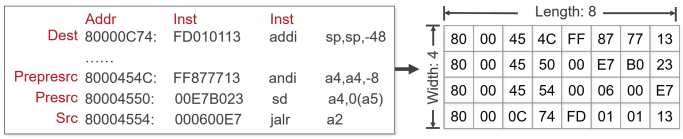


Fig. 4. An example of feature conversion.

To mitigate the inherent constraints of dynamic analysis, we deploy a Neural Network (NN) on the server for efficient top-tier remote CFI verification. To ensure easy integration and limited overheads, *this streamlined NN only consists of five layers*: an input layer, an output layer, and three intervening hidden layers with 16, 8, and 4 nodes respectively. It deploy ReLU as the activation function in intermediary layers, while the output probabilities were determined using softmax. It undergoes training using data sourced from the CFG, encompassing both benign and malicious data—the former represents CFG edges, while the latter denotes edges absent from the CFG.

It is noted that the proposed design is targeted at IoT devices running singular and uncomplicated programs that don't self modify. Given the sensitivity of IoT devices on communication and computation costs, a lightweight NN model as individual classifier for each program is more suitable than more complicated one-size-fits-all deep learning models. More importantly, a perfect offline CFG generation isn't mandatory, as the NN model is adept at discerning CFG features. During the detection phase, the encoded log data transmitted from the IoT device is deciphered to construct the test data. This is then fed to the pretrained classifier. If the data is classified as an illicit control flow transfer, an alarm activates, simultaneously sending a termination directive to the edge.

The dataset is derived from the CFG. At the binary level, fine-grained details about jumps, like function address boundaries, remain elusive without appended hardware or software modules. To enrich sample information, NEMU records not only the (src, dest) address pair, but also the two antecedent addresses and their four corresponding instructions. Direct and indirect jumps both exhibit similar characteristics, such as jumping range and standard destination instructions (like *auipc* or *addi*). The *auipc* primes the memory access address for upcoming *load* instructions, whereas *addi* earmarks stack territory for functions. Consequently, we enhance NEMU to chronicle all program jumps—these form our benign samples. For malicious counterparts, the initial trio of addresses and instructions mirror benign iterations, but the fourth address is randomized. This fourth instruction aligns with the random address in the program's *.bin* file. Random generation of malicious samples is a current, albeit pragmatic, widely adopted approach [16], [17], given the challenge of amassing ample malicious real-world samples for training, even if they might stray from actual CRAs.

Leveraging RISC-V ISA's constant instruction length and the strong image processing capability of the neural networks [22], we introduce a novel feature representation by transmuting instruction addresses and codes into images. Fig. 4 showcases this feature configuration. Each instruction address and code byte can be perceived as a pixel, transforming each sample into a 4×8 gray-scale image. During the operational hours, if deemed necessary, the device transmits a (prepresrc, presrc, src, dest) address quaternion coupled with its device ID. For instance, referencing Fig. 4, a failed broadcast query in step three of Fig. 1(a) prompts the CFI monitor to dispatch a 0x8000-(0x454c, 0x4550, 0x4554, 0x0c74) address quaternion to the server. Utilizing these addresses, the server extracts the pertinent instructions from the program binary file, crafting a gray-scale image to nourish the NN.

IV. Evaluation

A. Experiment Setup

We deployed the proposed hardware CFI monitor into two open-source RISC-V designs: NutShell [23] and Rocket Chip [24]. Least but essential alterations were undertaken to establish a connection between the monitor and the core. The experimental platform is based on multiple Xilinx Artix-7 FPGAs equipped with 1GB DDR3.

For evaluation, we employed benchmarks from various suites as prior works [11], [25]. MiBench2 [26], [27] is explicitly crafted for IoT device performance evaluation, featuring algorithms like AES and FFT. CoreMark [28] is tailored to gauge the performance of embedded processor cores. Fceux [29] stands out as an open-source NES emulator, which, given its range of input patterns and states, can help examine the efficacy of proposed NN module in Fig. 1. Deepsjeng and exchange2 from SPEC CPU 2017 [30] are selected due to better alignment with typical IoT setups. Our assessments of the proposed design were executed on bare-metal without an operating system. For dynamic analysis, NEMU [21] was employed, while the static analyses leveraged SVF [31].

B. Hardware & Power Overhead

We allocated 512 lines for CFGMap, translating to a storage of 2KB, and 1024 entries for CFGRam for approximately 2KB storage as well. After embedding the proposed monitor, we synthesized the two different RISC-V designs on multipe FPGAs to evaluate their resource consumption and the consequential power overhead. The ensuing data on resource consumption has been collated in Table II. A dissection of the metrics reveals that the integration of the proposed CFI monitor results in a relatively marginal uptick in resource utilization. For the NutShell, we observed increments of 1.78% in LUTs, 2.48% in registers, and 1.57% in BRAMs. The Rocket Chip showcased a slightly different profile with increments of 1.84% in LUTs, 3.26% in registers, and 4.76% in BRAMs. Thus, the ensuing hardware overhead, chiefly engendered by the inclusion of the shadow stack, CFGMap, CFGRam, and their corresponding components, remains reasonably constrained.

The average power consumption of the proposed monitor was a mere 0.002W. It accounted for only 1.32% of NutShell (measured at 0.151W) and 2.53% of Rocket Chip (measured at 0.075W). The power consumption of the proposed CFI monitor is virtually negligible for IoT devices.

C. CFG Generation Using Dynamic Analysis

Instead of using the conventional static analysis, our approach for CFG generation employed dynamic analysis, which

TABLE II
HARDWARE OVERHEAD

	LUTs	Registers	BRAMs
NutShell	14579	12306	62.5
CFI Monitor	267	316	1
Nut + CFI	14975	12702	63.5
Rocket Chip	14616	5713	20
CFI Monitor	275	196	1
Rocket + CFI	14902	6020	21

was motivated by our ambition to amplify the precision of the original CFG. In pursuit of a comprehensive evaluation, we applied both static and dynamic analysis to the above five benchmarks on server side. Since it is infeasible to traverse all input vectors, we consequently resorted to randomly generating test inputs within the constraints defined by the programs. The results are illustrated in Table III. It was found that dynamic analysis demonstrated its prowess in ascertaining the destinations of indirect jumps. It's essential to spotlight the challenges posed by static analysis, especially its struggles with scrutinizing indirect jumps embedded within standard functions like *printf()*. Remarkably, such functions account for a significant chunk of the overall indirect jumps. On the other hand, for the larger programs with diverse inputs, like Fceux and deepsjeng, static analysis stands out in exhaustively navigating all direct jumps. However, given that the destinations of direct jumps are hard-coded and immune to modifications, as delineated by our threat model, dynamic analysis emerges as the front-runner for forging a high-precision CFG within our system framework.

TABLE III
COMPARISON OF STATIC ANALYSIS AND DYNAMIC ANALYSIS

	Dynamic		Static	
	Indirect	Direct	Indirect	Direct
MiBench2	66	893	42	830
Coremark	12	432	2	288
Fceux	327	1466	211	6889
deepsjeng	23	2856	0	3732
exchange2	3	282	0	224

D. Performance Overhead

As prior CFI researches [10]–[12], [14], [15], we evaluated performance overhead under fixed CFG conditions. Additionally, we also assessed the performance overhead under incomplete CFG conditions, which primarily stems from these factors: cycle counter in CFI Monitor, network latency, and NN inference latency. Such incompleteness scenario is common in practice, but has not been well studied in these works.

Above prior CFI researches employed a presumptive perfect and fixed CFG and incurred performance overhead due to ISA extension or code instrumentation. Unlike the prior works, our hardware based approach does not require ISA extension nor code instrumentation. Fig. 5 compared the performance overhead using the proposed approach, where the first two bars in each group represent the performance of the original hardware without any modification (normalized to 1) and the modified hardware with the proposed approach. Clearly, with fixed CFG, our approach has no performance degradation.

By intentionally removing 10%, 20%, and 30% of the entries from the initial CFGs across benchmarks, we mimicked the

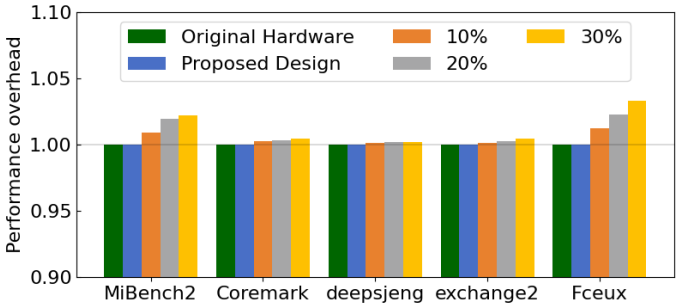


Fig. 5. Evaluation of extra performance overhead caused by the missing entries in the initial CFG.

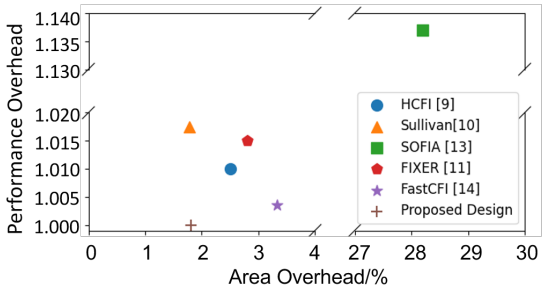


Fig. 6. Comparison on performance overhead v.s. area overhead tradeoff with prior works.

situation of incomplete CFG and measured the corresponding performance overhead. The last three bars in each group of Fig. 5 revealed less than 5% performance degradation across different benchmarks. Remarkably, benchmarks like Coremark, deepsjeng, and exchange2 reported an overhead of less than 1%. The overhead is inherently influenced by the program's runtime and the magnitude of missing entries. For instance, MiBench2's shorter runtime combined with its higher missing entries count leads to a comparably larger overhead than deepsjeng. Notably, each missing entry's performance toll is a one-time cost. The degree of acceptable performance hit is contingent on the specific program in question. A further analysis showcased that all MiBench2 programs, tailored for IoT, could generate a comprehensive CFG via dynamic analysis. This implies that for the majority of real-world IoT scenarios, our design effectively operates with almost zero performance overhead.

E. Comparisons to Prior Works

As discussed in the last few sections, our design was to devise CFI solutions that reduce reliance on offline immaculate CFG generation, while ensuring no performance overhead. Fig. 6 further visualizes the comparison between the proposed CFI design and prior hardware-centric CFI methods [10]–[12], [14], [15], where the performance of the original design without modification is normalized to 1 as the reference. Though the hardware overhead of our model is slightly better than prior works [10]–[12], [14], [15], it is noted that the proposed approach outperforms in terms of much lower performance overhead. In other words, the proposed CFI design renders this overhead entirely justifiable for IoT devices.

F. Effectiveness of the proposed remote CFI verification

The effectiveness of the proposed top-tier remote CFI verification was evaluated using Fceux, which was selected for its judicious mix of jump varieties and intricate input dynamics

that mirror IoT use cases. To train the proposed NN, the dataset was curated from its CFG using NEMU. The split between training and test data was at 9:1, where the training has 1614 positive and similar number of negative instances.

We measured the efficacy of proposed system using NN as remote CFI verification through its false negative and false positive metrics in terms of control flow classification (false negatives: overlooked malicious instances; false positives: innocent instances erroneously flagged as malicious). Our model has an impressive accuracy rate of 0.983, accompanied by a marginal false positive score of 0.033 on the testing subset (as shown in Table IV). The ROC graph in Fig. 7 further accentuates the model's prowess, reflected through an area-under-the-curve (AUC) value of 0.99. In comparison to other commonly used machine learning strategies [32], including SVM, Logistic Regression (LR), and Random Forest (RF), we replaced the NN model in our system with the other models and re-evaluate the efficacy. It is found that the proposed NN still trumps its peers as shown in Table IV and Fig. 7.

We also simulated attacks based on buffer overflows and pointer corruptions to test the security effectiveness. According to threat model, we directly tamper indirect jumps and return addresses. All the tampered indirect jumps resulted in pipeline stall and further request to server. Deployed NN detects all malicious indirect jumps and finally terminates the program.

TABLE IV
COMPARISON OF CONTROL FLOW CLASSIFICATION USING THE PROPOSED NN AND OTHER MACHINE LEARNING MODELS.

Model	False Positive	False Negative	Accuracy
NN	0.033	0	0.983
SVM	0.233	0.280	0.744
LR	0.354	0.445	0.601
Random Forest	0.061	0.103	0.918

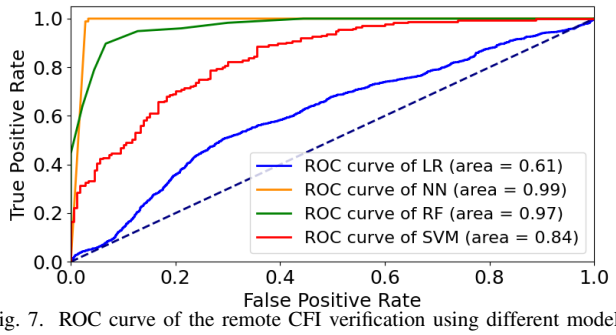


Fig. 7. ROC curve of the remote CFI verification using different models.

V. CONCLUSIONS

This research presented a hardware-assisted CFI design tailored to shield IoT devices from CRAs, obviating the need for code instrumentation. Despite adversaries' potential to directly alter return addresses and indirect jumps, the proposed CFI monitor remains invisible to software and only communicates minimal information externally. Although resilient against common threats like backward-edge and forward-edge attacks, the proposed design does show vulnerabilities to advanced threats like COOP. Even with the limitations of the fixed size of CFGRam and the need for distinct classifiers per program, the proposed work's primary contribution lies in its dynamic nature and reduced dependence on offline CFG generation, making it more adaptable to real-world IoT scenarios. Our experiments on two different RISC-V platforms and subsequent FPGA evaluations affirmed negligible power and hardware overheads, alongside zero performance costs for fixed CFG scenarios.

ACKNOWLEDGMENT

This work was partially supported by NSFC (Grant No. 62034007, 62141404, 62204111) and SGC Cooperation Project (Grant No. M-0612).

REFERENCES

[1] C. Zhuo *et al.*, "Noise-aware dvfs for efficient transitions on battery-powered iot devices," *IEEE TCAD*, 2019.
[2] S. M. Alnaeli *et al.*, "Vulnerable c/c++ code usage in iot software systems," in *IEEE WF-IoT*, 2016.
[3] R. Roemer *et al.*, "Return-oriented programming: Systems, languages, and applications," *ACM TISSEC*, 2012.
[4] T. Bletsch *et al.*, "Jump-oriented programming: a new class of code-reuse attack," in *ACM CCS*, 2011.
[5] P. Team, "Pax address space layout randomization (aslr)," *http://pax. grsecurity. net/docs/aslr. txt*, 2003.
[6] V. Katoch, "Whitepaper on bypassing aslr/dep," *S. Technologies, Ed., ed*, 2011.
[7] M. Abadi *et al.*, "Control-flow integrity principles, implementations, and applications," *ACM TISSEC*, 2009.
[8] R. De Clercq *et al.*, "A survey of hardware-based control flow integrity (cfi)," *arXiv:1706.07257*, 2017.
[9] D. Arora *et al.*, "Hardware-assisted run-time monitoring for secure program execution on embedded processors," *IEEE VLSI*, 2006.
[10] N. Christoulakis *et al.*, "HCFI: Hardware-enforced control-flow integrity," in *ACM CODASPY*, 2016.
[11] D. Sullivan *et al.*, "Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity," in *ACM/EDAC/IEEE DAC*, IEEE, 2016.
[12] A. De *et al.*, "FIXER: Flow integrity extensions for embedded RISC-V," in *DATE*, 2019.
[13] G. S. Kc *et al.*, "Countering code-injection attacks with instruction-set randomization," in *ACM CCS*, 2003.
[14] R. De Clercq *et al.*, "SOFIA: software and control flow integrity architecture," *Computers & Security*, 2017.
[15] L. Feng *et al.*, "Fastcfi: Real-time control-flow integrity using fpga without code instrumentation," *ACM TODAES*, 2021.
[16] L. Chen *et al.*, "Henet: A deep learning approach on intel® processor trace for effective exploit detection," in *IEEE SPW*, 2018.
[17] J. Zhang *et al.*, "DeepCheck: A Non-intrusive Control-flow Integrity Checking based on Deep Learning," *arXiv:1905.01858*, 2019.
[18] N. Carlini *et al.*, "{Control-Flow} bending: On the effectiveness of {Control-Flow} integrity," in *USENIX Security*, 2015.
[19] W. Andrew *et al.*, *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*.
[20] N. Burow *et al.*, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys*, 2017.
[21] Z. Yu, "NEMU." https://github.com/NJU-ProjectN/nemu.
[22] J. Deng *et al.*, "Energy-efficient real-time uav object detection on embedded platforms," *IEEE TCAD*, 2019.
[23] "NutShell." https://github.com/OSCPU/NutShell.
[24] K. Asanovic *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech.*, 2016.
[25] R. J. Walls *et al.*, "Control-flow integrity for real-time embedded systems," in *ECRTS*, 2019.
[26] M. R. Guthaus *et al.*, "Mibench: A free, commercially representative embedded benchmark suite," in *IEEE WWC*, 2001.
[27] M. Hicks, "Mibench2." https://github.com/impedimentToProgress/MiBench2, 2016.
[28] S. Gal-On *et al.*, "Exploring coremark a benchmark maximizing simplicity and efficacy," *EEMBC*, 2012.
[29] L. Sabota, "FCEUX." https://github.com/TASEmulators/fceux, 2016.
[30] The Standard Performance Evaluation Corporation, "SPEC CPU 2017." https://www.https://www.spec.org/cpu2017/, 2017.
[31] Y. Sui *et al.*, "SVF: interprocedural static value-flow analysis in LLVM," in *ACM CC*, 2016.
[32] D. Pfaff *et al.*, "Learning how to prevent return-oriented programming efficiently," in *ESSoS*, 2015.