

CVA6-CFI: A First Glance at RISC-V Control-Flow Integrity Extensions

Simone Manoni*, Emanuele Parisi†, Riccardo Tedeschi*, Davide Rossi*§, Andrea Acquaviva*, Andrea Bartolini*

*Department of Electrical, Electronic, and Information Engineering - University of Bologna, Italy

†High Performance Domain-Specific Architectures Group - Barcelona Supercomputing Center, Spain

§Department of Digital Design and Open Hardware - Chips-IT, Italy

{s.manoni, riccardo.tedeschi6, davide.rossi, andrea.acquaviva, a.bartolini}@unibo.it, emanuele.parisi@bsc.es

Abstract—This work presents the first design, integration, and evaluation of the standard RISC-V extensions for Control-Flow Integrity (CFI). The `Zicfiss` and `Zicfilp` extensions aim at protecting the execution of a vulnerable program from control-flow hijacking attacks through the implementation of security mechanisms based on shadow stack and landing pad primitives. We introduce two independent and configurable hardware units implementing forward-edge and backward-edge control-flow protection, fully integrated into the open-source CVA6 core. Our design incurs in only 1.0% area overhead when synthesized in 22 nm FDX technology, and up to 15.6% performance overhead based on evaluation with the MiBench automotive benchmark subset. We release the complete implementation as open source.

Index Terms—Control-Flow Integrity, Shadow Stack, Landing Pad, RISC-V

I. INTRODUCTION

RISC-V embedded platforms are experiencing rapid adoption across security and safety-critical industrial domains. These computing systems frequently execute critical workloads implemented in memory-unsafe languages, while operating in untrusted environments exposed over the network. These features make them prime targets for cyberattacks [1]. In fact, an attacker could exploit memory corruption bugs to modify vulnerable code pointers in memory, hijack program control-flow, and trigger arbitrary code execution by leveraging code-reuse attacks such as return-oriented programming (ROP), threatening the security of the whole system at risk [2–4]. Control-Flow Integrity (CFI) has emerged as an effective defense mechanism against control-flow hijacking attacks by ensuring that programs execute only along intended control-flow paths, thereby preventing unauthorized deviations at runtime. A typical CFI policy enforces protection on both “forward edges” and “backward edges” of control flow. Forward-edge policies protect indirect jumps where target addresses are computed dynamically during execution, such as those resulting from function pointer dereferencing in C. Conversely, backward-edge protection secures function returns, whose target addresses depend on values retrieved from the stack

This work was supported by the ISOLDE Chips JU project (grant no. 101112274) and by Technology Innovation Institute, Secure Systems Research Center, Abu Dhabi, UAE (PO Box 9639). Emanuele Parisi is supported by AI4S fellowship within the “Generación D” initiative, Red.es, Ministerio para la Transformación Digital y de la Función Pública, for talent attraction (C005/24-ED CV1).

memory. While various hardware-level approaches protect against control-flow diversion [5–8], dedicated ISA extensions have been the most adopted solution in commercial architectures like Intel and ARM, both featuring instructions devoted to the enforcement of CFI [9, 10]. RISC-V has recently ratified two CFI ISA extensions, `Zicfiss` and `Zicfilp`, which standardize an instruction-level mechanism for backward and forward-edge protection, implementing shadow-stack and landing pad CFI protection mechanisms, respectively. To date, no study has evaluated their microarchitectural impact. This paper explores how the RISC-V CFI extensions impact the performance and area of RISC-V application-class embedded cores.

- We design two hardware components implementing the RISC-V CFI extensions: a shadow stack unit implementing the `Zicfiss` extension and a landing-pad unit implementing the `Zicfilp` extension.
- We integrate the designed components into CVA6 [11], an embedded application-class core, presenting CVA6-CFI, a CVA6 variant with full CFI support. The complete implementation is available open-source¹.
- We evaluate the impact of our RISC-V CFI extensions implementation, showing only 1.0% area overhead and up to 15.6% performance overhead across the MiBench automotive benchmark subset. Additionally, our implementation introduces the lowest area overhead compared to other hardware-centric CFI methods.

II. BACKGROUND AND RELATED WORKS

A. Commercial ISA Extensions for CFI

Both Intel and ARM have extended their ISAs with CFI instructions. Intel’s Control-flow Enforcement Technology (CET) includes two key features: Shadow Stack and Indirect Branch Tracking (IBT). The Shadow Stack is a write-protected memory page dedicated to storing return address pointers whenever the core executes a `call` instruction. The `ret` instructions raise an exception if the target address does not match the address stored in the shadow stack. IBT introduced the `endbranch` instruction, which marks valid code targets for indirect calls and jumps. If an indirect control transfer

¹<https://github.com/AlSaqr-platform/cva6/tree/pulp-v1-culsans>

attempts to target any invalid target, the processor raises an exception [10]. ARM introduced the Branch Target Instructions (BTI) to protect indirect forward jumps and Pointer Authentication for function return protection. BTI restricts indirect branches to target only a designated instruction, similar to Intel’s endbranch working principle [12]. Pointer authentication utilizes the unused upper bits of 64-bit pointers to store a Pointer Authentication Code (PAC). During function calls, a PAC is generated from the return address and stored in the pointer’s upper bits. Before the function returns, the core authenticates the pointer, and an exception is triggered if an attacker tampers with the return address [9].

B. CFI RISC-V ISA extension

Recently, RISC-V extended its ISA to support CFI through the introduction of Landing Pads and Shadow Stack extensions. The `Zicfilp` extension introduces a labeled landing pad instruction (`lpad`) encoded as `auipc x0, label`. The execution of an indirect jump forces the processor into a state where the only instruction allowed to retire next is a `lpad` instruction with a matching label, or a software-check exception is triggered. The expected label is pre-loaded in register `x7` and must match the label encoded in the immediate field of the `lpad` instruction. The `Zicfiss` extension provides backward-edge protection via a shadow stack managed through dedicated instructions, including shadow stack push (`sspush`), pop and check (`sspopchk`), pointer read (`ssrpd`), and atomic swap (`ssamoswap`). Non-leaf functions save the link register into the shadow stack entry tracked by a dedicated shadow-stack pointer. Programs maintain both the regular and shadow stacks to protect return addresses from tampering. Unlike Intel’s CET, the return address is not transparently pushed and popped from the shadow stack in conjunction with function calls and returns.

C. CVA6 microarchitecture

CVA6 is a 64-bit core designed for application-class systems. It features an in-order, single-issue, six-stage pipeline with hardware support for virtualization. The pipeline features two front-end stages responsible for generating the next program counter and interfacing with the instruction cache to fetch instructions. The decode stage realigns and decodes the instructions, storing them in the issue queue. The issue stage contains the issue queue, scoreboard, and reorder buffer, and it issues instructions to the execute stage once all operands are ready. The execute stage integrates the core functional units, and the commit stage retires up to two instructions per cycle, updates the register file, and resolves write-back conflicts through the reorder buffer.

III. METHODOLOGY

A. Shadow Stack ISA extension design

Figure 1 illustrates the CVA6 pipeline, enhanced with the CFI extensions. The modules highlighted in blue have been introduced, or modified, to support the `Zicfiss` extension. The Control Status Registers (CSRs) have been extended to include

the Shadow Stack pointer, and the environment configuration registers for machine, supervisor, and hypervisor modes have been integrated and expanded with a Shadow Stack enable field. This field is utilized in the frontend to compute the Shadow Stack-enabled state across various privilege levels and is then forwarded to the decode and execute stages. The decoder and compressed decoder are extended to handle the decoding of `sspush`, `sspopchk`, `ssprd`, and `ssamoswap` instructions, as well as their compressed counterparts. The `sspush` and `sspopchk` instructions have been mapped to store and load instructions, respectively, with special operation tags to ensure that `Zicfiss` operations are differentiated from standard memory operations in later pipeline stages. The same applies to the `ssamoswap` instructions, which are tagged with dedicated Shadow Stack swap W/D identifiers. We have implemented a dedicated Shadow Stack Unit (SSU) in the execution stage, responsible for performing shadow stack pop checks and early instruction filtering before allowing the instructions to access the LSU. If the SSU detects a `ssamoswap` operation executing in machine mode, or any `Zicfiss` instruction running with Supervisor or Virtual address translation and protection disabled, and at a privilege level below machine mode, a store access fault exception is raised and forwarded directly to the scoreboard. If neither of these two conditions is met, access to the LSU is allowed. Furthermore, when the SSU detects a `sspopchk` instruction, it buffers the link register content and the load transaction ID in dedicated registers and then waits for the load result. It also sets a flag to indicate that a `sspopchk` is in progress. When a `sspopchk` is in-flight, the SSU compares the load transaction ID of each load result coming from the LSU with the buffered transaction ID. If these IDs match, the SSU then checks the recorded link register value against the load result. If there is a mismatch between these values, a software check exception is raised. Additionally, we extended the MMU to enforce checks based on the type of page accessed. `Zicfiss` instructions can access only shadow stack pages, while non-`Zicfiss` can access only non-shadow stack pages. Any violation of this access policy results in a store access fault.

B. Landing Pads ISA extension design

The modules highlighted in orange in Figure 1 have been introduced, or modified, to support the `Zicfilp` extension. To enforce forward-edge CFI, RISC-V compilers emit a dedicated `lpad` as the first instruction of any indirect jump targets. To support this mechanism, we integrate a Landing Pad Unit (LPU) at the interface between the scoreboard and the commit stage of the core pipeline. The LPU monitors ordered, non-speculative, and valid instructions, and it checks two events. First, when an update to the `x7` register is detected, the LPU records this as the most recent valid landing pad label configured by the program. Second, when an indirect jump is observed, the pipeline transitions to a state in which only the `lpad` instruction matching the current landing pad label is permitted to execute. Upon reaching this state, if a matching `lpad` instruction is encountered, it is retired without

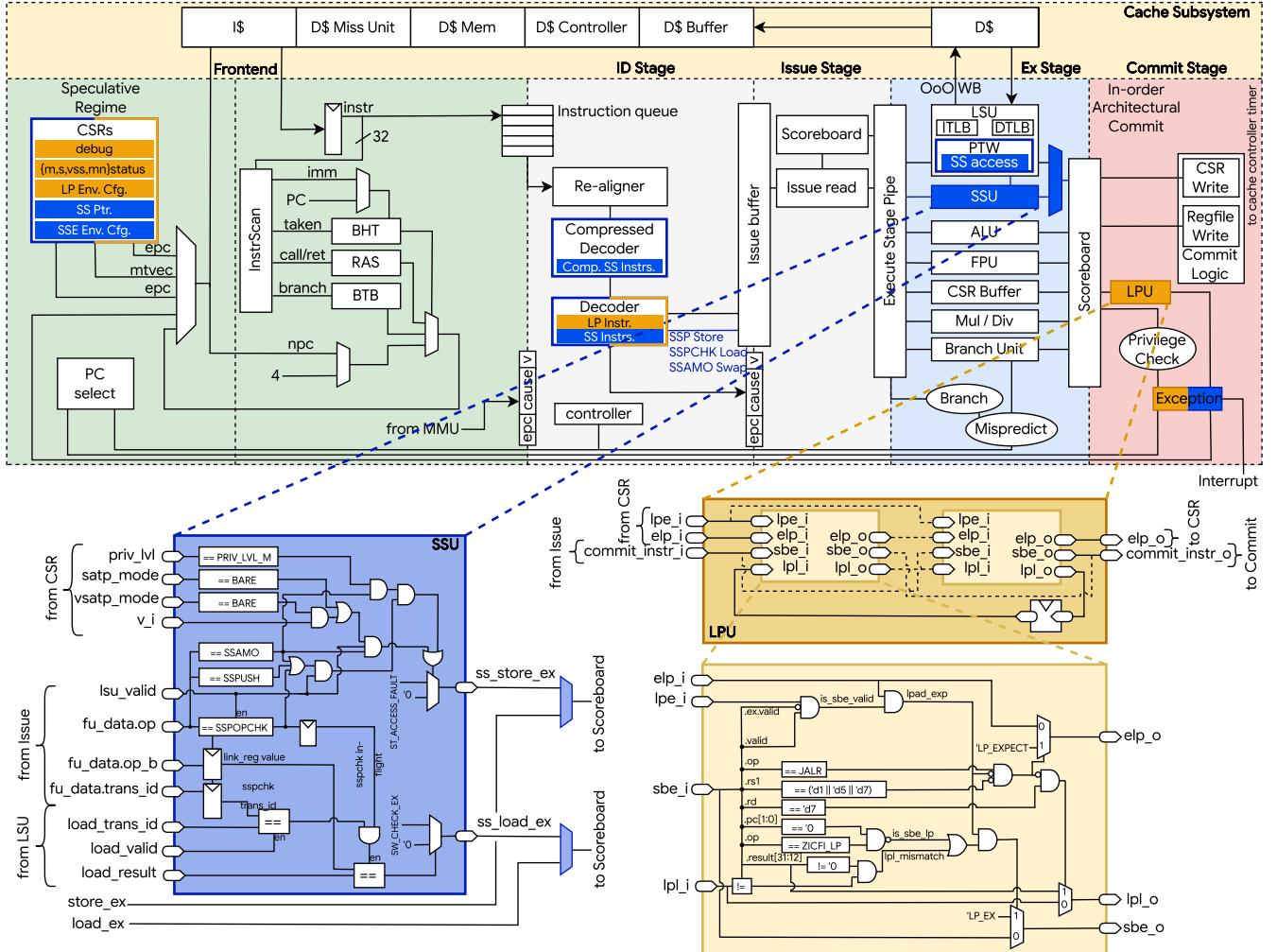


Fig. 1: CVA6-CFI microarchitecture.

side effects; if not, the LPU raises an exception. By observing fully executed, non-speculative instructions, the LPU design facilitates seamless integration into pipelines such as CVA6, which feature multiple commit ports capable of retiring more than one instruction per cycle. In these configurations, a chain of LPUs is instantiated, one per commit port, and each instance propagates information about lpad executions or landing pad label updates to the next unit. While this design choice increases propagation delay in the commit stage, it addresses corner cases, such as when an indirect jump and its landing pad are retired in the same cycle. Similarly to the Zicfiss extension, we extended the CSR register file and the decode stage with the necessary control and decoding logic to support the Zicfilp extension.

IV. RESULTS

A. Hardware Characterization

To evaluate the impact on area and performance of the RISC-V ISA extensions, we synthesized the CVA6 core, with and without the Zicfilp and Zicfiss extensions, using Synopsys Design Compiler targeting GF22FDX technology. The target operating frequency was fixed at 800 MHz under

worst-case conditions (SSG corner, 0.72 V, -40°C to 125°C). Table I reports the impact of the CFI functional units on the core pipeline area. The cache subsystem is excluded to avoid masking the overhead introduced by our CFI extensions, as the cache remains identical between the baseline CVA6 and the CFI-enhanced version. Our implementation introduces a low area overhead across most pipeline stages, except for the commit stage. Although this stage doubles in size due to the two LPUs, its relative contribution to the total pipeline area is low; therefore, the overall area increase for CVA6-CFI remains negligible, approximately equal to 1.0%. Moreover, the integration of the shadow stack and landing pad functionalities does not impact the core's operating frequency. Figure 2, extended from [13], shows the area overhead comparison between our proposed CFI design and prior hardware-centric CFI methods [7, 13–17]. Each overhead value is measured relative to its baseline architecture, with CVA6-CFI introducing the least overhead.

B. Software Characterization

We evaluated the code size and runtime overhead of the RISC-V CFI extensions using the MiBench automotive bench-

TABLE I: CVA6 Area Overhead Analysis in 22nm FDX

CVA6 Pipeline	Area [μm^2]		Overhead [%]
	Baseline	w/ CFI ext.	
CSR Reg. File	7831	8220	5.0
Fetch & Decode	1101	1124	2.1
Issue	25637	25788	0.6
Execute	61631	61854	0.4
Commit	182	372	103.6
Total	96382	97358	1.0

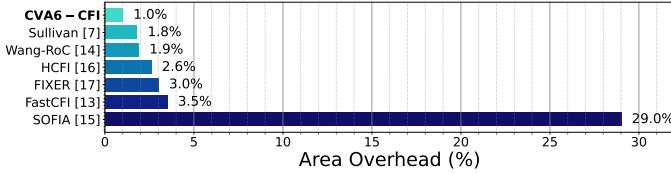
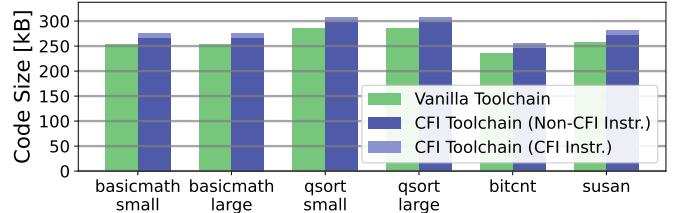
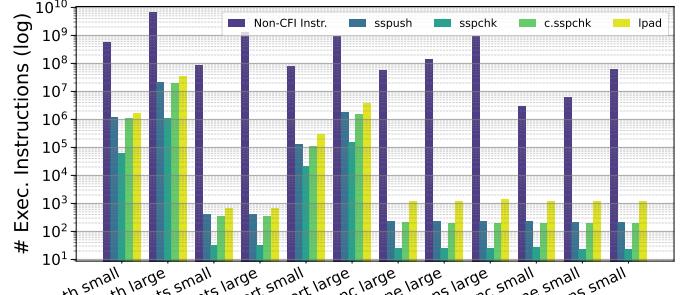


Fig. 2: CVA6-CFI area overhead comparison.

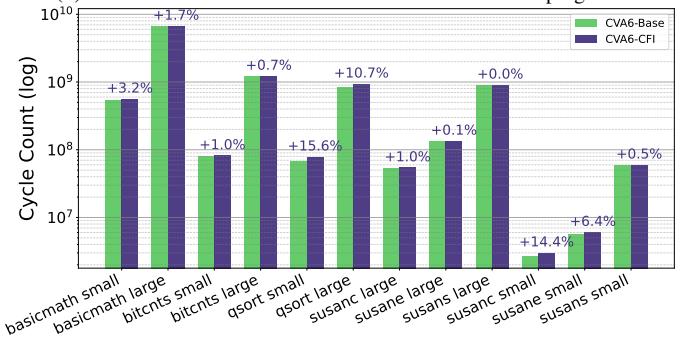
mark subset [18] compiled with the CFI-aware SiFive GCC 13.3 toolchain². Due to the absence of a stable CFI-patched Linux release, all benchmarks were executed on QEMU [19] in user-emulation mode, using a cycle-accurate model of the CPU calibrated against RTL simulations performed with Questa 2023.1. Figure 3a presents the code size overhead introduced by the CFI implementation across the MiBench automotive subset, compiled using the CFI-enabled toolchain. The results demonstrate consistent instrumentation costs with several key observations: The CFI extensions introduce a stable absolute overhead of approximately 22-23 kB across all benchmarks, representing a relative increase of 7.6-9.2%. This consistency suggests that the instrumentation adds a largely fixed cost independent of specific application characteristics. Notably, the overhead remains identical between small and large input variants of the same benchmarks (*basicmath* and *qsort*), indicating that the CFI cost is primarily determined by program structure rather than data size or complexity. Figure 3b shows the instruction count for the selected MiBench programs, distinguishing between CFI and non-CFI instructions. *basicmath_large* and *qsort_large* exhibit the highest CFI instruction counts, while *bitcount* and *susan* show minimal CFI activity. This pattern directly correlates with function call density: *basicmath_large*'s deeply nested loops calling a cubic equation solver repeatedly generate intensive shadow stack activity. In contrast, *qsort_large*'s recursive partitioning and frequent comparisons create substantial CFI overhead. The *bitcount*'s bitwise operations in tight loops and *susan*'s image processing with direct function calls result in minimal CFI instructions execution. Across all benchmarks, CFI instructions represent less than 0.5% of total executed instructions. Notably, *sswap* instructions are absent from all benchmarks. We assume they are primarily required in the Linux kernel for context switching and shadow stack management, so they are irrelevant to characterizing applications' behavior in the user space. Similarly, *ssrdp* only appears once during the initial benchmark setup to read the shadow stack pointer for



(a) Code size overhead for MiBench Automotive benchmarks.



(b) Number of instructions executed for each MiBench program.



(c) Performance comparison between CVA6 baseline and CVA6-CFI.

Fig. 3: CVA6-CFI Software Characterization

stack frame establishment. Therefore, we did not include them in the figure. Figure 3c compares the cycle counts of programs compiled with the vanilla and CFI-enabled toolchains running on CVA6 and CVA6-CFI, respectively. The CFI extension introduces a modest slowdown across all benchmarks, with a maximum relative overhead of 15.6%. This overhead is primarily determined by shadow stack operations (*sspush*, *sschk*), which account for all CFI-related memory accesses. Despite their prevalence, *lpad* instructions have a lower performance impact, as the LPU resolves them in a single cycle without requiring memory access in the execute stage.

V. CONCLUSION

This work presents CVA6-CFI, the first design and evaluation of the RISC-V standard extensions for CFI. Our approach introduces two independent hardware units implementing backward-edge and forward-edge control-flow protection. CVA6-CFI incurs in only 1.0% of area overhead and up to 15.6% performance overhead on the MiBench automotive subset; the full implementation is released open-source. Future work will aim at the silicon prototyping, power measurements and studying the impact of the RISC-V CFI extensions on advanced out-of-order and multi-thread core architectures.

²<https://github.com/sifive/riscv-gnu-toolchain/tree/cfi-dev>

REFERENCES

- [1] M. Pałka, J. Woodruff, D. Chisnall, M. Roe, and S. W. Moore, “Trends in Memory UnsaFety,” in *IEEE Symposium on Security and Privacy*, IEEE, 2024.
- [2] Y. Li, S. Liu, B. Liu, W. Huang, P. Wang, and L. Liu, “SoK: The Progress, Challenges, and Perspectives of Directed Grey-Box Fuzzing,” in *USENIX Security Symposium*, USENIX, 2024.
- [3] S. Yang, Y. Sun, Q. Yang, Z. Yang, Z. Qian, C. Wang, and Y. Wang, “SoK: All You Ever Wanted to Know about x86/x64 Binary Disassembly But Were Afraid to Ask,” in *IEEE Symposium on Security and Privacy*, pp. 957–974, IEEE, 2021.
- [4] H. Shacham, “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86),” in *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 552–561, 2007.
- [5] R. Kumar, S. Goyal, A. Handa, F. McKeen, I. Alexandrovich, V. Scarlata, and R. Sahita, “Cats: Composable and agile tracking for security,” in *IEEE Symposium on Security and Privacy (SP)*, pp. 819–836, IEEE, 2020.
- [6] C. Spang, Y. Lavan, M. Hartmann, F. Meisel, and A. Koch, “DExIE - An IoT-Class Hardware Monitor for Real-Time Fine-Grained Control-Flow Integrity,” in *Workshop on Design and Architectures for Signal and Image Processing (14th Edition)*, DASIP ’21, (New York, NY, USA), p. 26–34, Association for Computing Machinery, 2021.
- [7] D. Sullivan, O. Arias, V. P. Kemerlis, L. Davi, Y. Jin, A.-R. Sadeghi, and P. Larsen, “Efficiently Enforcing Control-Flow Integrity with a Security Coprocessor,” in *IEEE Secure Development Conference (SecDev)*, 2016.
- [8] E. Parisi, A. Musa, S. Manoni, M. Ciani, D. Rossi, F. Barchi, A. Bartolini, and A. Acquaviva, “TitanCFI: Toward Enforcing Control-Flow Integrity in the Root-of-Trust,” in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, 2024.
- [9] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, “PAC it up: towards pointer integrity using ARM pointer authentication,” in *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC’19, (USA), p. 177–194, USENIX Association, 2019.
- [10] V. Shanbhogue, D. Gupta, and R. Sahita, “Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity,” in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP ’19, (New York, NY, USA), Association for Computing Machinery, 2019.
- [11] F. Zaruba and L. Benini, “The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, 2019.
- [12] T. Yue, K. Lu, Z. Ning, P. Wang, L. Zhou, X. Zhou, Y. Wang, F. Zhang, and G. Zhang, “Efficient Forward-Edge Control-Flow Integrity for COTS Binaries via Arm BTI,” *IEEE Transactions on Information Forensics and Security*, vol. 20, pp. 7137–7152, 2025.
- [13] W. Wang, L. Feng, Z. Shi, C. Zhuo, and J. Chen, “Hardware-assisted control-flow integrity enhancement for iot devices,” in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, 2024.
- [14] L. Feng *et al.*, “Fastcfi: Real-time control-flow integrity using fpga without code instrumentation,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 26, no. 5, pp. 1–25, 2021.
- [15] R. D. Clercq, A. V. Herrewge, and I. Verbauwhede, “Sofia: Software and control flow integrity architecture,” *Computers & Security*, vol. 68, pp. 16–35, 2017.
- [16] N. Christoulakis, G. Christou, S. Ioannidis, and A. Papadogiannakis, “Hcfi: Hardware-enforced control-flow integrity,” in *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pp. 37–48, ACM, 2016.
- [17] A. De *et al.*, “Fixer: Flow integrity extensions for embedded risc-v,” in *Design, Automation and Test in Europe (DATE)*, pp. 348–353, IEEE, 2019.
- [18] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, “Mibench: A free, commercially representative embedded benchmark suite,” in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pp. 3–14, 2001.
- [19] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC ’05, (USA), p. 41, USENIX Association, 2005.