Efficient Forward-Edge Control-Flow Integrity for COTS Binaries via Arm BTI

Tai Yue, Kai Lu, Zhenyu Ning, Pengfei Wang, Lei Zhou, Xu Zhou, Yaohua Wang, Fengwei Zhang, *Senior Member, IEEE*, and Gen Zhang

***Abstract*—CONTROL-FLOW Integrity (CFI) has been widely recognized as an effective technique for mitigating control-flow hijacking attacks. However, many binary-level CFI approaches suffer from weaknesses in safeguarding forward edges, particularly for the obfuscated binaries, due to the imprecision in binary analysis or heuristic algorithms. Moreover, these approaches often involve non-negligible overhead and are challenging to deploy, as they instrument plenty of code or employ hardware tracing to enforce the CFI policies. This paper introduces MOBIUS, the first complete implementation of security-instruction-based binary-only CFI solution on commercial processors. MOBIUS leverages the Branch Target Identification (BTI) technology in Arm v8.5 to safeguard the forward edges of binaries and shared libraries efficiently. It determines the forward-edge targets without false negatives and carefully instruments the bti instructions to conduct the CFI checking efficiently. Then, it mounts a runtime monitor to detect potential attacks. We deploy MOBIUS on an Alibaba Cloud server with Yitian 710 processors in practice without modifying the kernel or loader. Remarkably, MOBIUS successfully provides efficient protection for real-world applications, including obfuscated code, with marginal overhead (5.78% on SPEC2006).**

***Index Terms*—Arm BTI, control-flow integrity, binary-only.**

I. INTRODUCTION

CONTROL-FLOW hijacking attacks exploit memory corruption vulnerabilities, such as buffer overflow, to divert program execution away from the intended control flow [1], [2], [3]. Traditional code-injection attacks are ineffective under the popular protection mechanisms, including Data Execution Prevention (DEP) [4] and Address Space Layout Randomization (ASLR) [5]. However, attackers have proposed powerful Code-Reuse Attacks (CRA), such as Return-Oriented Programming (ROP) [1], [2] and Jump-Oriented Programming (JOP) [3]. CRA can alter the run-time behavior of a program even if ASLR and DEP are enabled.

To mitigate these control-flow hijacking attacks, Abadie et al. [6] have proposed Control-Flow Integrity (CFI) technique. CFI forces the indirect control-flow transfers in the program to adhere to a precomputed Control-Flow Graph (CFG), which is usually constructed with the rich semantic information, such as symbols provided by the source code [6], [7]. Unfortunately, in practice, the demand for CFI is also critical in Commercial Off-The-Shelf (COTS) binaries, legacy binaries, and dynamic libraries, where source code is unavailable [8], [9]. For stripped binaries, the absence of source-level semantics poses significant challenges in inferring forward-edge indirect transfer targets (e.g., indirect calls/jumps) and reconstructing high-precision CFGs [8], [10]. Unlike backward indirect transfers (i.e., return instructions) that can be protected via shadow stacks at runtime [11], securing forward edges fundamentally requires precise target identification. Therefore, *efficiently implementing the binary-level forward-edge CFI is important but challenging*:

1) **A key problem is how to implement an efficient and practical runtime checking mechanism**. Many solutions [8], [9], [12], [13], [14] employ (dynamic or static) binary instrumentation to examine the indirect transfers. However, dynamic binary instrumentation (DBI) incurs heavy and unacceptable overhead, making it difficult to deploy in practice [12], [13]. Though the static binary instrumentation (i.e., binary rewriting) is more efficient than DBI, it poses certain prerequisites for binaries and has limitations in terms of applicability [15], [16]. Moreover, current rewriting-based CFI techniques [8], [9] need to instrument plenty of check code for enforcing CFI or dynamically translating code addresses of the original programs to the rewritten ones for compatibility, incurring extra performance and space overheads.

Apart from instrumentation, hardware tracing technologies have been employed by some CFI solutions [17], [18], [19], [20], [21], [22]. However, post-processing technologies (e.g., Intel Process Trace (PT) [23] and Arm CoreSight [24]) can trace all of the control-flow transfers in a light runtime overhead but requires heavy decoding workloads to obtain the control-flow information [25]. Though the decoding and analysing overhead can be partially addressed by offloading

Received 18 November 2024; revised 10 April 2025 and 29 May 2025; accepted 30 May 2025. Date of publication 16 June 2025; date of current version 16 July 2025. This work was supported in part by the Science and Technology Innovation Program of Hunan Province under Grant 2024RC3136; in part by the National Natural Science Foundation China under Grant 62272477, Grant 62421002, Grant 62272472, Grant 62302050, and Grant 62372121; in part by the Research Project of National University of Defense Technology under Grant ZK23-14; and in part by the Research Project of Key Laboratory of the State Administration of Science, Technology and Industry for National Defense under Grant WDZC20245250105. The associate editor coordinating the review of this article and approving it for publication was Dr. Fabio De Gaspari. *(Corresponding authors: Kai Lu; Zhenyu Ning.)*

Tai Yue is now with the Intelligent Game and Decision Lab, Academy of Military Sciences, Beijing 100091, China. He was previously with the College of Computer, National University of Defense Technology, Changsha 410073, China (e-mail: yuetai17@nudt.edu.cn).

Kai Lu, Pengfei Wang, Lei Zhou, Xu Zhou, Yaohua Wang, and Gen Zhang are with the College of Computer, National University of Defense Technology, Changsha 410073, China (e-mail: kailu@nudt.edu.cn; pfwang@nudt.edu.cn; zhoulcs@nudt.edu.cn; zhouxu@nudt.edu.cn; yaowangeth@gmail.com; zhanggen@nudt.edu.cn).

Zhenyu Ning is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China (e-mail: zning@hnu.edu.cn).

Fengwei Zhang is with the Department of Computer Science and Engineering, the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen 518055, China (e-mail: zhangfw@sustech.edu.cn).

Digital Object Identifier 10.1109/TIFS.2025.3580342

these workloads to a separate CPU core, this design reduces the number of usable CPU cores [18], [20], [22], [25], [26]. Last Branch Record (LBR) [27] is a lightweight technology but only records limited branches. LBR-based CFI tools [17], [28] may be vulnerable to specific attacks, such as endpoint-pruning or history flushing [17], [29]. Last but not least, since hardware tracing can directly monitor the CPU for capturing the instructions even when a Trusted Application is running in the Trusted Execution Environments (TEE) [30], the security risks inherent in hardware tracing technologies (e.g., nailgun [30] and branch-shadow [31]) make them less suitable for deployment in production environments [32].

2) **Current binary-level CFI techniques are weak in determining the set of valid destinations for forward edges**. The imprecision in binary analysis limits the CFG-based CFI solutions. Specific approach [33] employs conservative analysis strategies, resulting in over-approximated sets of indirect branch targets, which leaves enough wiggle room for an attacker to launch successful exploits [12], [14], [34]. Some tools [35], [36] only protect specific types of indirect transfers, such as virtual dispatches.

3) **Specific approaches based on heuristic detection may be imprecise due to the oversize search space**. Techniques [37], [38] detect the CRA by observing the behavior of programs and detecting the gadget patterns. However, they examine all of the indirect branches during the execution, while most of the branches are legal transfers. This brings significant difficulties and challenges in the design of the heuristic algorithm.

Historically, most of existing binary-level CFI techniques are designed for the X86 platform [8], [9], [12], [13], [17], [18], [19], [20], [28], [33]. However, the widespread popularity of Arm in embedded and mobile devices and its rapid development on personal computers and high-performance server front call for efficient CFI techniques on AArch64 binaries [39].

Hence, in this paper, **we focus on constructing an efficient and practical forward-edge CFI technique for AArch64 binaries**, which meets the following requirements:

- **Performance**. The system should incur negligible performance and space overhead.
- **Compatibility**. The system should modularly protect the binaries and libraries without certain prerequisites, such as debugging information, symbol tables, or relocatable code. It would be better to support some obfuscated binaries, as obfuscation has become a popular protection technique.
- **Security**. The system should resist specific CRA based on forward edges (e.g., JOP attack). It is noted that the arms race between attackers and defenders has given rise to numerous attacks [34], [40], [41], [42] that can break the CFI technique. Though our system may not be immune to these attacks, we believe that the adoption of our system remains of great importance as it can increase the difficulty for attackers to break the security of computer systems, particularly when the protection is provided cheaply.
- **Practice**. This system should be easily deployed in a production environment without relying on self-defined hardware architecture or heavy hardware tracing technologies. And the system should avoid modifying the kernel.

To satisfy these requirements, we propose the first complete implementation of **security-instruction-based binary-level CFI technique on commercial processors**. We notice that Arm recently proposes **Branch Target Identification (BTI)** technology in v8.5 to safeguard the forward edges [43]. BTI technology introduces the bti instruction to constrain the target of indirect branches. Benefiting from this instruction, we can implement minimally disruptive instrumentation and enforce CFI policy with negligible overhead.

Particularly, recent work FineIBT [7] has utilized the Intel BTI-type feature, Indirect Branch Tracking (IBT), to implement a fine-grain source-level CFI. This work also claimed that recent rewriters (e.g., Egalito [44], RetroWrite [15], and BinRec [45]) can be used to support IBT/FineIBT on binaries. However, our empirical evaluation reveals critical limitations in existing transformation-based rewriters: (1) they currently lack complete IBT/BTI instrumentation support. Though Egalito [44] provides software-level support for Intel IBT, its implementation is incomplete as CFI serves only as one application in its paper. Egalito only inserts endbr instructions at the entry points of address-taken functions while performing rigorous runtime validation to verify whether the target of every indirect branch (including both indirect calls and jumps) contains a valid endbr marker. Hence, legitimate indirect jumps may lead to abnormal exits of the rewritten program [46]. (2) crucially, many of them fail to reliably process even common binaries like libc due to incomplete symbolic recovery and code reconstruction [15], [44], [47], [48], raising concern about their practical capability for IBT-enabled CFI instrumentation. Hence, this paper is the first work to focus on extending the BTI technology from source-level CFI to binary-level CFI, orthogonal to FineIBT.

Based on Arm BTI, we propose MOBIUS, a prototype defense system consisting of a binary analyzer, a static binary rewriter, and a runtime monitor. 1) We propose the *profiling-based dynamic analysis method* in the binary analyzer. This method utilizes the DBI technique to instrument the indirect branches and recover the indirect transfers while executing the programs with pre-collected test cases. Compared to previous coarse-grain CFI techniques employing the over-approximated CFGs, our method determines the valid targets of intended indirect branches without any false positives, reducing the legitimate targets and enhancing the security. 2) We propose a *BTI-based incremental rewriting technique* in our static binary rewriter. This technique replaces the instructions in landing positions of indirect branches with bti instructions. It carefully recovers the overwritten instructions by introducing trampolines, avoiding changing the semantics of binaries or adjusting their layouts. Compared to specific binary rewriters [15], [48], [49], [50], our rewriter is efficient and independent of the specific requirements of binaries (e.g., symbolic information). Compared to previous binary-level CFI techniques based on heavy instrumentation or hardware tracing [12], [18], [20], [25], our rewriter enforces the CFI policy in negligible overhead. 3) We employ a runtime monitor to capture the exceptions and diagnose whether an exception is caused by an abnormal transfer or a legitimate but missed indirect branch in our dynamic profiling. Compared to previous

heuristic-detection-based techniques which observe all of the program behaviors [37], [38], our monitor only needs to analyse the indirect branches that trigger the exceptions, reducing the search space and enhancing the precision of detecting potential attacks. Moreover, our monitor supports the protection of multi-thread applications.

We implement MOBIUS on an Alibaba cloud server with Yitian 710 processors, which support the BTI feature. We conduct comprehensive experiments to evaluate its robustness, efficiency, and security. MOBIUS successfully rewrites many real-world applications and efficiently protects them. In detail, MOBIUS introduces 5.78% and 5.09% performance overhead on SPEC2006 and SQLite, and negligible overhead on Nbench and Redis. On average, MOBIUS reduces the 99.944% legitimate indirect targets, slightly higher than the original BTI strategy supported by GCC (i.e., GCC-BTI).

In summary, the contributions of this paper are as follows:

- We propose the first complete implementation of security-instruction-based binary-level CFI solution, MOBIUS, to protect the forward edges for COTS binaries via Arm BTI. The key design of MOBIUS is utilizing the bti instruction to achieve efficient checking and robust instrumentation.
- We propose the BTI-based incremental rewriting technique, which efficiently instruments the binaries without specific prerequisites. We utilize a profiling-based method to generate the under-approximated CFGs for binaries and employ a runtime monitor to diagnose the attacks precisely.
- We evaluate the robustness, efficiency, and security of MOBIUS. The results show that MOBIUS can correctly rewrite real-world applications and efficiently protect them.
- We open the source of MOBIUS at https://github.com/MoonLight-SteinsGate/Mobius

II. BACKGROUND

A. Code-Reuse Attack

CRA have become the prevalent attack techniques [1], [2], [3], [37]. As a powerful CRA technique, JOP hijacks the forward indirect transfers and carefully selects code snippets from the binary code to produce unintended instructions consequences [37]. Theses snippets are called gadgets and end in indirect branch instructions.

Fig. 1 portrays a straightforward JOP attack. In the code of test.c, there is a stack-overflow vulnerability on line 14, which the attack can exploit through carefully crafting input to arrange the stack layout. Then, when the jop_symbol function is indirectly called on line 15, the pointer of this function originally stored in register x1 has been replaced with 0xfffff7e3af8c (corresponding to the code offset $0 \times 2af8c$ in glibc). As a result, the control flow is redirected to glibc. Afterwards, the process executes the attacker-specified gadget, which stores the values 0xfffff7f5ddb8 and 0xfffff7e56d94 in registers x0 and x16, respectively. The former signifies the address of the string "/bin/sh" and the latter denotes the address of the —libc_system function. This gadget code is equivalent to the invocation of —libc_system("/bin/sh"). Finally, the attacker employs two indirect branches to perform a JOP attack and launch the unauthorized shell.

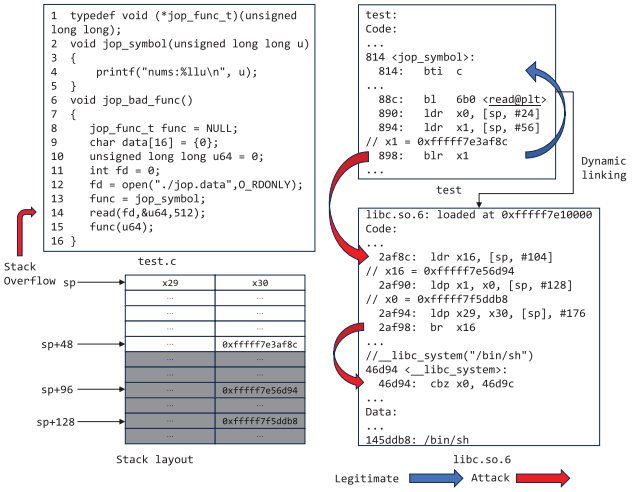


Fig. 1. An example of JOP attack.

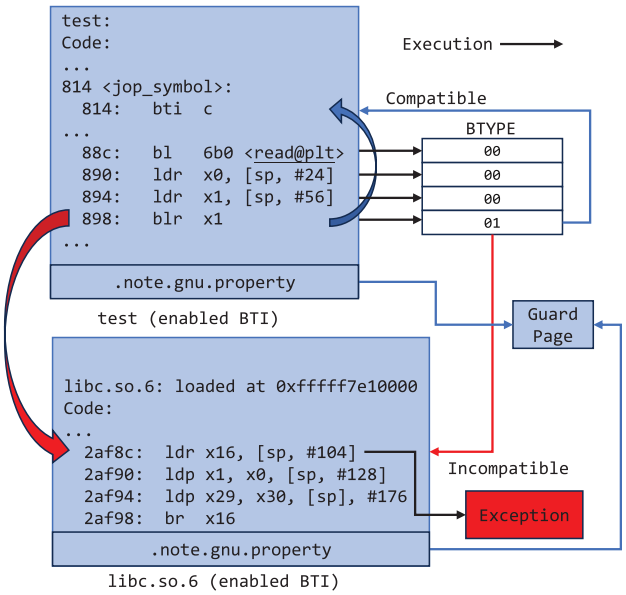


Fig. 2. The introduction of BTI technology.

B. Arm Bti

To protect the forward edges and mitigate the JOP attacks, Arm v8.5 introduced the BTI technology as a complement to the PA (Pointer Authentication) technology in Arm v8.3.

Fig. 2 illustrates the basic principle of this technology. Specifically, Arm introduces a 2-bit BTYPE field in the PSTATE registers to record the current CPU's BTYPE state. After executing each instruction, the CPU updates the value of BTYPE based on the types of the instruction and its memory page. Instructions that cause the value of BTYPE to become non-zero are indirect branches, such as the instruction at the offset 0×898 in Fig. 2. The value of BTYPE is also affected by whether the instruction is within the protected memory page, which is indicated by the GP (Guarded Page) bit in the Block and Page descriptors. If an instruction resides in the guarded memory region and current BTYPE is not zero, this instruction must be compatible with this BYTPE (e.g., bti instruction), otherwise a BTI exception will be raised.

For instance, as shown in Fig. 2, when the control flow indirectly transfers to 0xfffff7e3af8c in glibc, BTYPE is updated

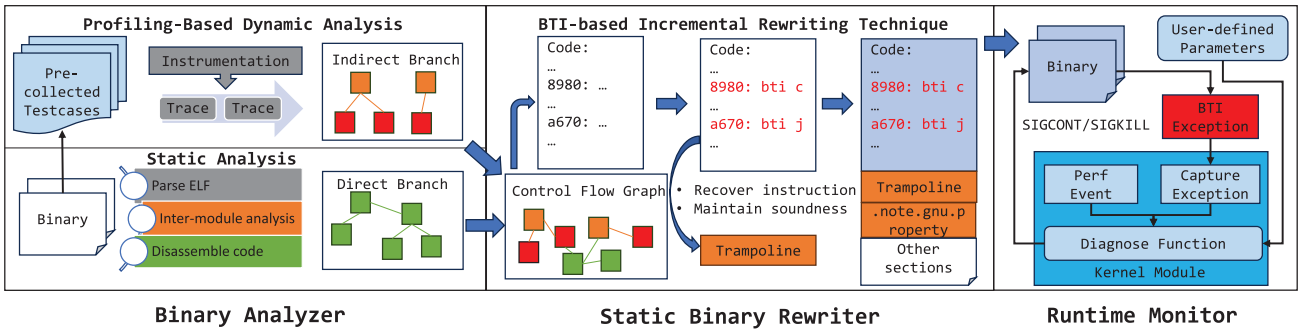


Fig. 3. The architecture of MOBIUS.

TABLE I
BTYPE ON EXECUTION OF AN INSTRUCTION AND THE COMPATIBILITY OF BTI INSTRUCTION

Instruction	Memory	Register	BTYPE	BTI
br, braa, braaz, brab, brabz	Guarded	Except x16 or x17	0b11	jc, j
blr, blraa, blraaz, blrab, blrabz	Any	Any	0b10	jc, c
br, braa, braaz, brab, brabz	Guarded	x16, x17	0b01	jc, j, c
br, braa, braaz, brab, brabz	Non-guarded	Any	0b01	jc, j, c
Others	Any	Any	0b00	-

to 0b01 and not compatible with the current instruction. Then, the process raises a BTI exception and exits. Conversely, if the program indirectly calls function jop_symbol, the followed instruction is bti c and compatible with the non-zero BTYPE, allowing the program to execute normally. Thus, the BTI technology can be used to restrict the destinations of indirect branches. Additionally, bti instructions are divided into three types: jc, j, and c, which are compatible with different BTYPE values. The details are outlined in Table I.

Importantly, whether a code page is marked as the GP page by the kernel or loader is primarily determined by whether the binary is marked with the BTI feature. For enabling this feature, the compiler needs to insert the .note.gnu.property section with the corresponding values into the binary.

C. Threat Model

We consider a powerful attacker with arbitrary control of process memory in user space but limited by the DEP policy, which disallows the attacker to modify the code region. Similarly to previous work [14], we also assume that the backward edges have been protected by some ideal backward CFI techniques such as the shadow stack in the lightweight overhead [11]. This means that the attacker can modify any forward pointer in data region to launch a runtime attack by utilizing the memory vulnerabilities. We seek to defend against such attacks with MOBIUS.

III. DESIGN

A. Overview

Fig. 3 presents the architecture of MOBIUS. To satisfy the four requirements proposed in Section I, MOBIUS employs the bti instruction to implement the runtime checking. Therefore, MOBIUS needs to determine the instrumentation points, instrument the check code, and diagnose the BTI exceptions, which are conducted by a binary analyzer, a static binary rewriter, and a runtime monitor, respectively.

To improve the compatibility and security of MOBIUS, the analyzer combines static analysis with *profiling-based dynamic analysis* to generate the under-approximated CFGs. Specifically, we aim to provide intra-module and inter-module protection for the binary and its linked libraries, including the obfuscated code. Hence, we use pre-collected testcases for training to obtain the indirect branches without any false positives. Then, the analyzer takes static analysis to obtain the layouts of binaries, disassemble the code, capture the direct edges, and construct the inter-module calls. These results are integrated to form the CFGs, used as the input by the rewriter. Particularly, above analysis is independent of specific prerequisites of binaries (e.g., symbols).

To enhance the performance, compatibility, and practicality of MOBIUS, we propose a *BTI-based incremental rewriting technique*. We replace the destined instructions of indirect branches with bti instructions and introduce trampolines at their next instructions to maintain the semantics of programs. Moreover, we propose four rewriting tactics in various scenarios. To reduce the space overhead, we propose a *Nop-based trampoline inserting method*, which utilizes the redundant nop instructions within the original code sections to place trampolines. Finally, the rewriter updates the metadata in binaries and re-initiates dynamic linking to ensure all binary modules are protected. Our rewriter does not rely on some prerequisites, such as symbol tables or relocatable code, and can also handle statically linked binaries.

To enhance the practicality and security of MOBIUS, we developed a kernel module as the runtime monitor to analyse the BTI exceptions during protected binary execution. These exceptions may result from either: (1) attacker-induced illegal indirect branches, or (2) legitimate branches missed during profiling. The latter case occurs when valid indirect branch targets lack inserting bti instructions, potentially causing false positives during normal execution. Hence, like prior heuristic-based approaches [28], [51], we mount a runtime monitor. This monitor tracks process execution via Performance Monitor Unit (PMU) counters, captures BTI exceptions, and diagnoses attacks based on exception frequency patterns, which effectively solves the issues of hampering the normal execution of programs under certain inputs and resists the JOP attacks.

B. Binary Analyzer

The analyzer combines dynamic profiling and static analysis to analyse the layouts of binaries and generate the CFGs.

1) Profiling-Based Dynamic Analysis: MOBIUS involves dynamic profiling to offline analyse the indirect control-flow transfers in the binaries. Similar to previous work [52], we consider the profiling results as the trusted computing base of MOBIUS. This can be ensured because the testcases are collected by the user. Even if some testcases erroneously execute illegal indirect transfers, which typically result from inadvertently triggered memory vulnerabilities, users can perform offline validation of the results using runtime diagnostic tools (e.g., Valgrind) and then exclude these testcases.

With regard to the collection of testcases, we present several recommended approaches: 1) Generating testcases via advanced testing techniques (e.g., greybox fuzzing [53], [54] or symbolic execution [55], [56]). 2) Referring to previous works [20], [52] based on profiling. 3) Utilizing the testcases provided by the software, if any are available. The combined implementation of these methods can effectively produce good-quality testcases to increase the coverage of profiling, which are crucial for MOBIUS to enhance its robustness in rewriting the binaries and accuracy in detecting the attacks.

By driving the program with pre-collected valid test cases, we employ binary tracing techniques to capture the performed indirect branches and count their execution frequency for the rewriter. Popular tracing techniques include DBI (e.g., Intel Pin [57] and Dynamorio [58]) and hardware tracing (e.g., Arm CoreSight). Considering that: 1) The DBI techniques are relatively mature at present (e.g., Dynamorio). 2) Compared to some hardware tracing techniques, such as Intel PT or Arm CoreSight, which only records the destinations of branches without sources and requires manual reconstruction of control flow, DBI techniques are easier to deploy and can directly collect the branches. Therefore, we select the popular DBI tool Dynamorio to complete the analysis. We use it to trace the executed indirect edges, recording the source addresses, destination addresses, and executed instructions. It is noted that this analysis is independent of the binary's debugging information or symbol tables, which extends the practicality and compatibility of our analyzer.

2) Static Analysis: Drawing from the format of ELF, MOBIUS parses binary headers to identify the sections and segments. Similar to Lockdown [12], MOBIUS analyses the .dynsym sections, which even exist in stripped binaries, to obtain the symbols of exported and imported functions and constructs inter-module call graphs. Additional metadata (e.g, memory layouts) aids subsequent disassembly and rewriting.

For disassembly, MOBIUS conducts linear disassembly on code sections by leveraging AArch64's fixed-length ISA feature, ensuring all instructions are correctly disassembled. Notably, the presence of inline data (data inside the code sections) may result in some data being erroneously disassembled into instructions [59]. The semantics of binaries may be broken if the inline data is destroyed in the rewriting. Fortunately, our rewriter only needs to focus on the instructions related to the destined instructions of indirect branches and their next instructions. The possibility of inline data being corrupted by our rewriter is extremely low.

After code disassembly, our analyzer extracts the direct edges. Then, it incorporates static analysis and dynamic profiling results to generate the CFGs.

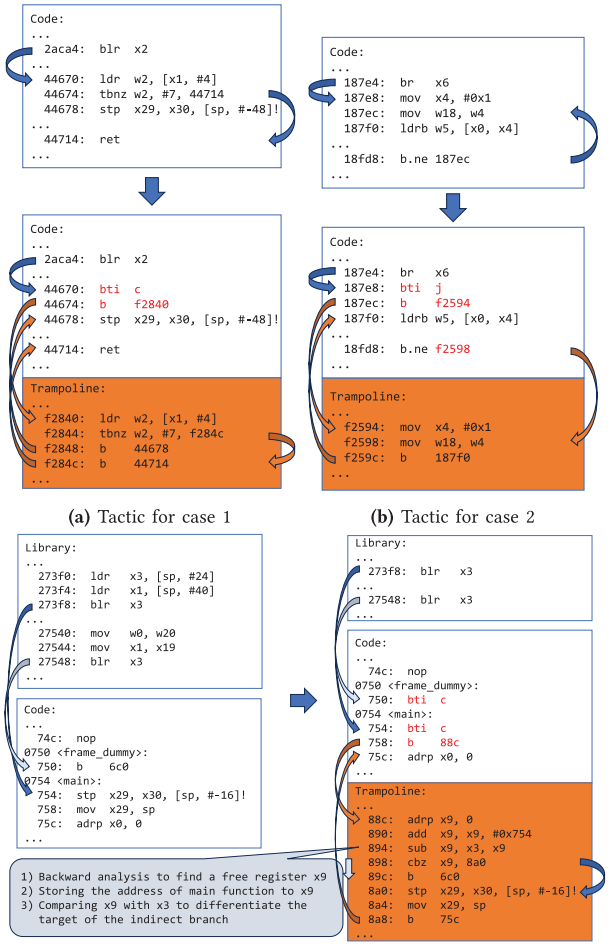
3) Under-Approximated Sets of Valid Targets: The binary analyzer determines the targets of indirect branches with: 1) *No false positives*. Generally, the control-flow transfers obtained via profiling are not susceptible to false positives (assuming that the DBI tool is error-free and the testcases are trusted). And the exported functions collected by MOBIUS are determined by the corresponding imported functions. It means that the destinations of indirect branches we collect are under-approximate compared to the ground truth. Hence, MOBIUS efficiently reduces the potential attack surface in the program and its shared libraries. Particularly, not all functions within the dynamic libraries are called by the program in most instances [60]. It may be possible for attackers to carry out specific attacks (e.g., return-to-libc) that use the addresses in glibc even under the CFI protection [33]. MOBIUS mitigates part of these attacks as MOBIUS would not instrument these unused functions with BTI instructions, while over-approximated CFI solutions [33] may misjudge these "intra-module called correctly" functions as legal targets. This inspires us to apply BTI in debloating binaries [60], while it is not the focus of this paper and reserved for future research. 2) *Potential false negatives*. Since it is difficult to cover all indirect branches in profiling, the indirect branch targets collected by MOBIUS may have false negatives. MOBIUS would not perform instrumentation on these missed but legitimate positions. If the control flow indirectly branches to these destinations, BTI exceptions would be raised, resulting in process termination. To solve this problem, we employ a runtime monitor and allow users to manually add the indirect destinations, which we will elaborate on in Section III-D.

C. Static Binary Rewriter

Upon obtaining the CFGs and layouts of binaries, MOBIUS employs the binary rewriter to instrument bti instructions.

Existing rewriters mainly fall into two categories: transformation or trampoline [15], [48], [49], [50]. 1) *Transformation-based approaches* translate binaries into reassemblable assembly code or Intermediate Representation (IR) for instrumentation and recompilation [15], [16], [47], [61]. While facilitating flexible instrumentation, their reliance on precise static analysis during the transformation requires specific binary assumptions. 2) *Trampoline-based techniques* redirect control flow at instrumentation points via branch instructions to dedicated trampoline segments, where instrumentation logic executes [50]. Compared to the transformation, they are better suited for additive instrumentation with looser binary requirements, though implementation complexity increases.

Our work aims to instrument bit instructions at the landing positions of indirect branches with minimal overhead and static analysis dependency (e.g., symbol information). However, *after our testing, we found that existing techniques fail to meet our requirements directly*. Transformation-based rewriters (Egalito [44], ARMore [61] and DDisasm [47]) cannot process glibc even without instrumentation in our testing. Upon analysis, the reason might be due to the lack of symbolic information and complex code of the stripped libraries, leading to difficulties in precisely transforming the binaries.



(a) Tactic for case 1 (b) Tactic for case 2

(c) Tactic for case 3

Fig. 4. The rewriting tactics of MOBIUS.

Particularly, Egalito [44] implements an incomplete support of Intel IBT. Moreover, it requires inputs to be position-independent code (PIC) and fails on obfuscated code or inline assembly, which embeds data. The trampoline-based rewriter [50] directly replaces the original instructions with branches rather than bti instructions. Although semantics-preserving, this design may trigger BTI exceptions when indirect branches target overwritten instructions. While CCFIR [9] constrains indirect transfers via trampoline sections, it inserts additional check code in original code sections before redirection, introducing higher implementation complexity than in-place instruction replacement [50].

1) BTI-Based Incremental Rewriting Technique: To carefully rewrite the binaries with negligible overhead and great robustness, we design a BTI-based incremental rewriting technique. Our core idea is directly overwriting the destined instructions with bti instructions and recovering the overwritten instructions by introducing the trampolines.

Fig. 4a illustrates an example of binary rewriting in SQLite. An indirect call occurs from offset 0×2aca4 to 0×44670. MOBIUS rewrites the instruction at the destination with bti and the next instruction with a branch instruction to redirect control flow to the trampoline code. It then recovers the overwritten instructions from the original code sections in the trampoline. Finally, MOBIUS appends a branch instruction to return to the original code. During rewriting, two crucial issues must be addressed: 1) Recovering the semantics of the rewritten instructions in the original sections; 2) Maintaining the correctness of program semantics in complex scenarios.

a) Recovering Instruction: Limited by the fix-length feature of AArch64 ISA, some addressing instructions and direct branch instructions may be broken and out of range if we move them from the original code to the trampoline (e.g., the instruction at 0×44674 in Fig. 4a), which we need to fix.

Algorithm 1 Binary Rewriting Strategy for BTI Instrumentation

```
linenosize=landing_points = GETLANDINGPOINTS()
db_edges = GETDIRECTBRANCHEDGES()
db_points = GETDIRECTBRANCHPOINTS(db_edges)
for l_i in landing_points do
    current_instr = GETINSTRUCTION(l_i)
    if ISBTIINSTRUCTION(current_instr) then
        UPDATEBTITYPE(l_i)
    else if l_i + 4 ∉ landing_points then
        REWRITEINSTRUCTION(l_i, bti)
        INSERTTRAMPOLINE(l_i + 4, addr_i) {Tactic 1}
        if l_i + 4 ∈ db_points then
            REDIRECTJUMPS(l_i + 4, addr_i + 4, db_edges)
            {Tactic 2}
        end if
    else if CHECKSHAREDREGISTER(l_i) or CHECK-
    SHAREDREGISTER(l_i + 4) then
        BACKWARDANALYSIS(l_i)
        if FINDFREEREGISTER(l_i) then
            REWRITEINSTRUCTION(l_i, bti)
            REWRITEINSTRUCTION(l_i + 4, bti)
            INSERTTRAMPOLINE(l_i + 8, addr_i) {Tactic 3}
        else
            CONTINUE {Tactic 4}
        end if
    else
        CONTINUE {Tactic 4}
    end if
end for
```

We categorize AArch64 instructions into two groups: (1) instructions encoded with addressing offsets, including direct branches (e.g., b, bl, b.cond, tbz/tbnz, and cbz/cbnz) and PC-relative addressing instructions (e.g., adr and adrp); (2) all other instructions, which remain unchanged. For the former group, we must recalibrate the offset based on the target address of the instruction and its updated address in the trampoline. However, if the trampoline is too far from the original target, the recalculated offset may exceed the addressing range of instruction. To resolve this, for the PC-relative addressing instructions (e.g., adr and adrp), we insert arithmetic instructions to revise the offsets. For the branch instructions, as the addressing range of the unconditional branches (i.e., b and bl) is up to 128MB, we can consider that only conditional branches may exceed their addressing range. Hence, as shown in Fig. 4a, we append a *relay branch instruction* in the trampoline, which jumps to the original target (i.e., 0×44714) of the conditional instruction. Then, we redirect the conditional instruction to this relay instruction.

b) Maintaining Soundness: The example in Fig. 4a is straightforward. However, real-world scenarios can be more complex. In our design, MOBIUS must rewrite at least two consecutive instructions from the original code to insert one bti instruction. This may merge two control flows: one branching to the bti instruction and another (if present) branching to the next instruction. Hence, we classify the scenarios into four cases and propose tailored tactics. Algorithm 1 presents the rewriting strategy of MOBIUS. Specifically, MOBIUS iterates through each landing points of indirect branches and adopts distinct tactics based on whether the next instruction is the target of a direct or indirect branch.

Case 1: No branch instruction directs to the next instruction. This is the most common case. The tactic for this case has been introduced in Fig. 4a.

Case 2: Only direct branch instructions are directed to the next instruction. Our tactic for this case is redirecting these direct branch instructions to the trampoline. As shown in Fig. 4b, a conditional branch instruction in offset $0\times18fd8$ may branch to the offset $0\times187ec$. For a correct rewriting, we insert the trampoline as tactic 1. Then, we redirect those direct branches, which originally branch to the next instruction, to the corresponding instruction in the trampoline.

Case 3: There exists an indirect branch instruction that branches to the next instruction. This rare case generally occurs when two adjacent functions are called indirectly and the preceding function contains only one instruction. In this situation, we need to rewrite the two adjacent landing positions with bti instructions and differentiate the intended destination for the actual control flow. Hence, as shown in Fig. 4c, we first take the backward analysis from the two landing positions (i.e., 0×750 and 0×754) to find a free register (i.e., x9). Then, we determine whether there exists a landing position A (e.g., 0×754) of them, where all of its predecessor nodes (e.g., instruction in $0\times273f8$) utilize the same register (e.g., x3 in Fig. 4c) to branch to it indirectly. We rewrite these landing positions in the original code and insert a trampoline. In the trampoline, we store the address of A in the free register x9 and compare x9 with the used register x3. If the indirect branch is taken from $0\times273f8$ to 0×754, there will be the same value being held in both x9 and x3. Then, the conditional branch at 0×898 in the trampoline will be taken to $0\times8a0$ while preserving the program's semantics intact. Moreover, we select the instructions cbz/cbnz rather than cmp inside our trampoline as they would not change the conditional flags. Note that this tactic relies on acquiring a free register to store the address of landing points, which may fail if no such register is available. In such cases, we take the tactic 4 (i.e., no operation).

Case 4: The others. There may be specific scenarios where none of the above tactics are effective. For instance, three consecutive instructions serve as the landing positions. Fortunately, such cases are extremely rare. In the event that these circumstances occur, we abstain from instrumentation to prevent any potential compromise of the program's semantics. Instead, we leave the handling of BTI exceptions caused by this non-operation tactic to our runtime monitor.

2) Nop-Based Trampoline Inserting Method: To minimize rewriting overhead in performance and space, we utilize the hollows in the original code sections to place the trampoline code. Due to address alignment when linking object files into binaries, there are usually some continuous and unreachable nop instructions in the code sections. We take a reachable analysis to identify those free nop instructions as our priority areas for placing trampolines. In detail, we insert the trampolines sequentially according to the descending order of the execution frequency reported by profiling. We search for the closest free area (if it exists) for each trampoline to place it. This approach lightly reduces the overhead caused by space and memory usage from instrumentation and superior maintenance of the locality features of the code. It should be noted that the nop instructions must be unreachable. Otherwise, the semantics of the program may change due to the placement of trampoline. Since the trampoline contains at least three instructions, three consecutive nop instructions are generally considered safe. However, if there is a function satisfying: 1) it is only be called indirectly, 2) the entry point is a nop instruction, 3) and its predecessor function introduces at least two nop instructions at the end due to instruction alignment, rewriting at these points will break the program. Fortunately, these cases are extremely rare.

Supposing the free areas are not enough to accommodate all trampolines. In that case, it is necessary to introduce a new code section for placing the remaining trampolines. Therefore, we analyse the memory layouts of binaries to find the free space to insert the trampolines without changing the offset between the code sections and the data sections.

Finally, after rewriting the instructions and inserting the trampolines, MOBIUS updates the metadata in the headers of binaries to maintain their soundness. Importantly, all of the above processes in our analyzer and rewriter are independent of any source-level information, such as symbol information. Moreover, benefiting from the PC-relative addressing mode of AArch64 ISA, MOBIUS inherently supports position-independent code (PIC) and is compatible with ASLR. Notably, MOBIUS typically requires rewriting at least two consecutive instructions when instrumenting a bti instruction. This approach partially relies on accurate identification of indirect branch targets. MOBIUS may preserve program semantics even when profiling erroneously introduces false positives (i.e., unintended indirect targets). However, when the coverage of profiling is extremely insufficient, MOBIUS may erroneously rewrite an instruction that was actually an indirect branch target (e.g., Case 3), potentially breaking the semantics. We will discuss this issue in our evaluation.

a) Inline Data Disruption: A legitimate concern arises regarding whether the rewriter might inadvertently corrupt inline data unrecognized during disassembly. We argue that such occurrences are highly improbable as the rewriter only modifies three types of instructions: (1) Instructions at the landing points. The landing points are valid branch targets during execution. Since inline data usually inherently lacks execution semantics, it cannot manifest as landing points.

(2) The next instructions of (1). If the position is placed with inline data, the landing point must be a branch instruction (e.g., a function that contains only a ret instruction). This probability is also extremely low, and we did not observe such case when testing the SPEC2006 in Section IV.

(3) Instructions which directly branch to (2). The probability of 32-bit data coincidentally encoding a valid branch

instruction (i.e., b/bl/b.cond/tbz/tbnz/cbz/cbnz) is $\frac{25}{2^9}$. Furthermore, the probability of such an instruction precisely pointing to a given position is no more than $\frac{1}{2^{19}}$. Given typical landing point counts (no more than 1,000), the cumulative collision probability remains below $\frac{25}{2^{18}}$.

From above analysis, MOBIUS introduces negligible risk of inline data corruption. This probabilistic guarantee stems from architectural constraints of AArch64 instruction encoding and the sparsity of valid landing points.

D. Runtime Monitor

In Section III-B, we mentioned that the false negatives in profiling may introduce unnecessary BTI exceptions during the normal execution of the application. To mitigate this, we develop the runtime monitor to capture and analyse the exceptions. Algorithm 2 elaborates on its workflow. The monitor encompasses three primary functionalities:

1) Counting Instructions: Before execution, the monitor initiates the perf events based on PMU to count the numbers of instructions and branches executed by the program in user space, which will be used by the diagnosis function.

Algorithm 2 The Workflow of Runtime Monitor

```
pmu_ins = PERFEVENTCREATEKERNELCOUNTER()
last_ins, last_branches = GETINSNUMS(pmu_ins)
repeat
    if BTIEXCEPTIONRAISED() and CHECKPID() then
        cur_ins, cur_branches = GETINSNUMS(pmu_ins)
        EL0_context = GETCONTEXT()
        BTI_ins_interval = cur_ins - last_ins
        BTI_branch_interval = cur_branches - last_branches
        if BTI_ins_interval < JOP_INS_THRESHOLD or BTI_branch_interval < JOP_BRANCH_THRESHOLD then
            RAISESIGNAL(SIGKILL)
        else
            last_ins = cur_ins
            last_branches = cur_branches
            EL0_context.BTYPE = 0
            RAISESIGNAL(SIGCONT)
        end if
    end if
until UNMOUNTMONITOR()
```

2) Handling Bti Exception: In the event of a BTI exception, the monitor takes over the BTI exception handling functions by kprobe. First, it verifies whether the current process belongs to the protected program, which can avoid the potential conflicts between the protected program and the other BTI-protected processes (e.g., programs compiled by GCC-BTI). It then analyses the kernel stack to retrieve the saved user-space context at the point of exception, which includes the status register containing the value of BTYPE. The diagnosis function further utilizes the profiling information to determine whether the exception is caused by attacks. If the exception is considered to be triggered by an illegal transfer, the monitor will terminate the process. Otherwise, the monitor resumes the process by altering the termination signal to a continuation signal and setting the BTYPE value to zero. As such, after returning to user space, the process will not cyclically trigger BTI Exceptions due to a non-zero BTYPE value when re-executing the instructions.

3) Diagnosis Function: The monitor employs the diagnosis function to determine the origins of BTI exceptions, which plays a pivotal role in the security of MOBIUS. Since we have assumed the presence of the backward CFI technique in our threat model, we need to safeguard against attacks based on forward control-flow hijacking, namely JOP attacks.

Typical JOP utilizes a dispatcher gadget to orchestrate the control flow among the gadgets. Though variants such as PCOP [62] and stealth JOP [38] have been proposed, most JOP attacks require multiple (at least two) illegal indirect branches to chain gadgets into shellcode [3]. Moreover, researches show shorter gadgets are preferred for JOP chains, as longer gadgets often overwrite registers and memory excessively, creating unwanted side effects [37], [38].

Therefore, we assume a JOP attack triggers at least two BTI exceptions within specific instruction sequences. As shown in Algorithm 2, we denote the numbers of executed instructions and branches between two adjacent BTI exceptions as BTI_ins_interval and BTI_branch_interval, respectively. Then, we compare them with the user-defined thresholds JOP_INS_THRESHOLD and JOP_BRANCH_THRESHOLD. Exceeding either threshold suggests a potential JOP attack. Notably, these thresholds greatly affects the precision of MOBIUS in detecting attacks. Lower thresholds can enhance the robustness of MOBIUS in handling the uncovered indirect branches but may render MOBIUS weakened in resisting the long-gadgets attacks (e.g., stealth JOP [38]). In contrast, higher thresholds may help MOBIUS to detect these attacks but misjudge the exceptions caused by the false negatives in our analyzer. Therefore, referring to previous works [37], [38] and our testing, we select 10 instructions and 2 branches as the default thresholds. Importantly, users can define the thresholds across applications. In extreme cases, users can choose the aggressive strategy that terminates the program on exception. Though this strategy may lead to false positives in detecting attacks and block some legitimate branches, it maximizes the protection of MOBIUS.

a) Manually Adding Landing Positions: It is noteworthy that an indirect branch directed to uncovered landing positions will lead to a context switch for MOBIUS to analyse the exception. Therefore, excessive uncovered positions may result in heavy runtime overhead and impair the precision of MOBIUS. In such scenario, MOBIUS allows the users to manually add landing positions to the CFGs and reconducts rewriting for improving the protected binary.

We implemented a wrapper to invoke the monitor and drive the protected programs. The wrapper allows users to set the thresholds for the diagnosis function. It employs a fork-execve execution model to mount the runtime monitor with passing the user-defined thresholds. The wrapper also passes the process ID (PID) of protected program before executing it and the monitor only capture the BTI exceptions from the processes spawned by the protected program. Finally, the wrapper unmounts the monitor after the program is exited.

Compared to the PT-based CFI technique [19], our runtime monitor does not change the kernel code, exhibiting excellent portability. The runtime defenses [37], [38] also employ heuristic algorithms to examine all the executed indirect branches. The detection strategy in MOBIUS may be stricter and more precise than theirs, as we focus on the branches that trigger the alarms (More analysis is in Section IV-E).

IV. EVALUATION

In this section, we conduct comprehensive evaluations to answer the following research questions:

RQ1: How about the performance overhead of MOBIUS? How does the profiling coverage affect its performance?
RQ2: How about the robustness of MOBIUS in rewriting the real-world applications?
RQ3: How about the security and accuracy of MOBIUS in detecting the JOP attacks?

A. Implementation

We implement MOBIUS on Linux 5.15. The profiling tool (263 LoC) is implemented as a Dynamorio plugin. A Python script is provided to facilitate the profiling. We develop a Python framework (about 5,896 LoC) to support the binary rewriter and the static analysis component in the binary analyzer. This framework used pyelftools to parse the header, sections, and segments of binaries and capstone as well as keystone to disassemble and assemble instructions. For the runtime monitor, we develop a kernel module in 273 LoC. We also implement a wrapper to mount the monitor and drive the programs in 125 LoC.

B. Evaluation Setup

We ran all experiments on a 32-core virtual machine on Alibaba Cloud featuring the Yitian 710 processors and 64GB RAM, as this processor supports Arm BTI. An Ubuntu 22.04 system with Linux kernel 5.15 is running on this machine. We compile all benchmarks with their default configurations by GCC-11.4. For the performance evaluation, the results are reported as average over five runs.

C. Performance Overhead

To evaluate the overhead of MOBIUS, we test MOBIUS on several popular benchmarks, including the SPEC2006, the Nbench, the SQLite database engine, and the Redis server. We report the relative performance overhead compared to the baseline (i.e., original programs) of all experiments. Moreover, according to our design, the coverage in the profiling directly influences the frequency of alarms in executing the protected program, consequently affecting the runtime overhead. We consider to conduct the experiments under different coverage.

1) SPEC2006: Several binary-only CFI works [8], [14], [17], [18], [19], [33], [36] have used SPEC2006 benchmarks for performance evaluation, which contains 29 programs divided into CINT (12 integer programs) and CFP (17 floating-point programs). We follow these works and evaluate MOBIUS on SPEC2006. In detail, we compile all benchmarks under the default configuration on our platform using GCC, except for wrf which failed to compile. Then, we utilize the train workload as inputs to conduct the profiling and measure the performance. Moreover, to explore the overhead under different coverage, we randomly and aggressively reduce the collected indirect edges in profiling by 90%, 99%, and 99.99% (denoted as MOBIUS-90%, MOBIUS-99%, and MOBIUS-99.99%). To evaluate the overhead attributed to three components of MOBIUS, including the wrapper, instrumentation, and exceptions handling, we set four groups: baseline, baseline with wrapper, MOBIUS without enabling the BTI feature (i.e., MOBIUS w/o BTI), and MOBIUS. The monitor is configured to continue execution upon attack detection, allowing systematic performance evaluation across coverage thresholds.

Fig. 5 presents the results. MOBIUS introduces a geometric mean overhead of only 5.78% compared to the baseline on SPEC2006. **The wrapper, instrumentation, and exception handling of MOBIUS bring 4.06%, 1.19%, and 0.53% overhead, respectively.** The wrapper dominates the overhead but only executes during initialization, making MOBIUS introduce negligible overhead for long-running programs. Moreover, as shown in Fig. 5, MOBIUS brings marginal overhead on most benchmarks (e.g., <2% on 16 programs) while incurs certain overhead on specific binaries (e.g., 27.1% on xalancbmk and 33.83% on gcc). We deeply analyse these results. Short-running programs (calculix, gcc, libquantum) primarily incur overhead from wrapper invocation, with greater variance than other benchmarks. On perlbench and xalancbmk, MOBIUS instruments 804 and 701 bti instructions, respectively, leading to about 10% overhead due to the inserted trampolines. Particularly on xalancbmk, a land position in the entry of a single-instruction function is not instrumented and incurs numerous BTI exceptions, which is in Case 4 proposed in Section III-C. In detail, the predecessor nodes of the function use different registers to indirectly call it, where our tactics for Cases 1- 3 cannot be conducted. This triggers frequent BTI exceptions when executing some testcases and indirectly calling the function, incurring additional 15.6% overheads. Fortunately, such cases are extremely rare. Our rewriter handles most scenarios effectively.

Fig. 5 also reports the results of MOBIUS under varying profiling coverage. MOBIUS-90% introduces 5.96% overhead on SPEC2006, approximately to MOBIUS. Even for MOBIUS-99% and MOBIUS-99.99%, the overhead on most of benchmarks is comparable to MOBIUS, contrary to the intuitive expectation that lower coverage rates should proportionally increase overhead. Actually, our trace analysis reveals a bimodal distribution of indirect branch targets. Specifically, a small subset of indirect targets accounts for the vast majority of executions. Thus, aggressive 99% edge pruning still covers enough hot indirect targets to maintain modest BTI exception rates, consistent with fundamental program behavior characteristics. Only on perlbench, calculix, gcc, libquantum, and soplex, MOBIUS-99.99% introduces more than 20% overhead due to the numerous BTI exceptions.

We also notice that MOBIUS-99% and MOBIUS-99.99% fails on evaluating Xalancbmk. After examining the programs, MOBIUS successfully rewrites all benchmarks while preserving their original semantics even when 99.99% of indirect edges are pruned, with the exception of Xalancbmk. For

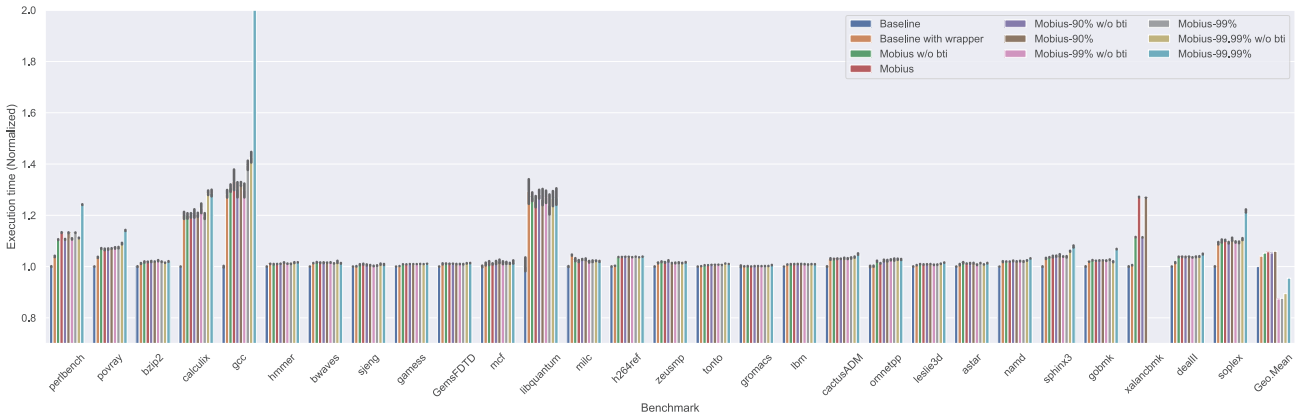


Fig. 5. The normalized running time of the baseline, baseline with wrapper, MOBIUS w/o bti, and MOBIUS (under different coverage) on the SPEC2006 benchmarks with unprotected programs as the baseline.

this particular benchmark, both MOBIUS-99% and MOBIUS-99.99% break its semantic during the rewriting process, leading to the program termination due to memory access violations during early execution phases. We deeply analyse the filtered edges and the rewriting operations of MOBIUS-99% and MOBIUS-99.99%. The complete execution trace reveals multiple indirect edges targeting both offset 0×81040 and 0×81044 with several registers. MOBIUS correctly identifies the inapplicability of Tactic 3 and implements conservative rewriting limited to offsets 0×81044-0×81048. However, MOBIUS-99% observes only a single edge to 0×81044 and incorrectly applies Tactic 3 due to the incomplete control flow, leading to overwrite multiple instructions beyond safe boundaries and introduce semantic violations. In contrast, MOBIUS-99.99% fails to find any edges to 0×81044 and takes Tactic 1 at 0×81040-0×81044, which also corrupts the program semantics. However, this phenomenon only occurs under extremely low coverage rates. Crucially, even in these challenging conditions, MOBIUS successfully rewrites the most of benchmarks.

2) Nbench: The Nbench benchmark [63] contains 10 different programs and is usually used to measure the performance of CPU, FPU, and memory system [64], [65], [66], [67]. Since the nbench binary contains the benchmark drivers and applications, we compiled the nbench in default configurations and directly ran it under the profiling mode of MOBIUS. Similarly, we also reduce the collected indirect edges by 90%, 99%, and 99.99% and rewrote nbench with MOBIUS, respectively. On average, all of the overheads introduced by MOBIUS under different coverage are negligible (no more than 0.027%).

3) SQLite: SQLite is a widely used transactional SQL database engine. We followed the work [7] and used the SQLite speedtest benchmark to conduct the experiments, which stress-tests SQLite and reports the time for performing a series of DB operations. The benchmark is distributed as a C file with the SQLite source code and linked with sqlite to create a binary, which contains the driver and application. Therefore, we tested speedtest in a consistent way with testing nbench. The programs rewritten by MOBIUS, MOBIUS-90%, MOBIUS-99%, and MOBIUS-99.99% speed 3.389s, 3.396s, 3.398s, and 3.630s to complete the DB operations, 5.09%, 5.31%, 5.36%, and 12.54% more than the baseline (3.225s), respectively. After our analysis, for the tools except MOBIUS-99.99%, executing numerous inserted branches incurs the most overhead. Even for MOBIUS-90%, where the protected process raises 3,494 exceptions, handling the alarms brings low overhead. However, the program rewritten by MOBIUS-99.99% raises 66,344 exceptions, incurring heavy overhead.

TABLE II
THE DETAILED RESULTS OF PERFORMANCE ON REDIS

Tools	Types	Ops/sec	Avg. Latency	KB/sec
MOBIUS	Sets	17841.6(0.51%)	1.308(-0.4%)	1374.4(0.51%)
MOBIUS-90%	Sets	17835.5(0.48%)	1.309(-0.33%)	1374.0(0.48%)
MOBIUS-99%	Sets	17766.6(0.09%)	1.314(0.07%)	1368.7(0.09%)
MOBIUS-99.99%	Sets	15883.3(-10.52%)	1.478(12.57%)	1223.6(-10.52%)
Baseline	Sets	17751.0	1.313	1367.5
MOBIUS	Gets	178394.5(0.51%)	1.304(-0.51%)	6951.3(0.51%)
MOBIUS-90%	Gets	178333.5(0.48%)	1.304(-0.49%)	6948.9(0.48%)
MOBIUS-99%	Gets	177644.7(0.09%)	1.31(-0.05%)	6922.0(0.09%)
MOBIUS-99.99%	Gets	158810.8(-10.52%)	1.464(11.69%)	6188.2(-10.52%)
Baseline	Gets	177489.4	1.311	6915.9
MOBIUS	Totals	196236.1(0.51%)	1.304(-0.5%)	8325.7(0.51%)
MOBIUS-90%	Totals	196169.0(0.48%)	1.305(-0.47%)	8322.8(0.48%)
MOBIUS-99%	Totals	195411.3(0.09%)	1.31(-0.04%)	8290.7(0.09%)
MOBIUS-99.99%	Totals	174694.1(-10.52%)	1.465(11.77%)	7411.8(-10.52%)
Baseline	Totals	195240.5	1.311	8283.4

4) Redis: Redis is a versatile data structure server and widely-used benchmark. Following the work [7], we used the memtier_benchmark to generate a stream of SET and GET reqs for a 32-byte object on a 1: 10 ratio for a while. We configured memtier_benchmark to use two threads with 128 simultaneous reqs and redis-server to use a worker thread on the same host. This configuration can evaluate the performance of MOBIUS in protecting the multi-thread applications. We tested redis-server for 10 seconds in profiling and 1 minute to measure the performance. Then, we repeated the trail five times and calculated the arithmetic means of metrics in memtier_benchmark. We also take the same strategy of reducing indirect edges as employed in our previous experiments. Table II reports the detailed results. MOBIUS, MOBIUS-90%, and MOBIUS-99% exhibit comparable performance to the baseline across all three evaluation metrics, with handling 6,681, 6,944, and 98,341 BTI alarms. However, MOBIUS-99.99% introduces approximately 12% overhead on all metrics due to too much BTI exceptions (5,596,814).

RQ1: Even removing 90% of the collected indirect edges, MOBIUS incurs marginal overhead on many applications and benchmarks. Performance overhead is high only when profiling coverage is extremely low, such as pruning the 99.99% edges.

D. Robustness

1) Real-World Applications: We further evaluate the robustness of MOBIUS using real-world applications beyond those in Section IV-C, including SQLite and GNU Binutils test suites.

In SQLite, we test two benchmarks: fuzzcheck (46,213 tests) and sessionfuzz (339 tests). Following the way to test speedtest, we compile and link them with sqlite to create distributed binaries. All tests pass successfully.

The GNU Binutils contained commonly used applications (e.g., objdump and readelf). We conducted the "make check" command to test them and implement the profiling during the testing. Then we rewrote them by MOBIUS. The rewritten binaries also passed all tests in the test suite, demonstrating the strong robustness of the rewriter in MOBIUS.

Moreover, we conduct some simple tests on ARMore [61] and find that it failed to rewrite several binaries (e.g., glibc, nbench, gamess, zeusmp, GemsFDTD, and xalancbmk). Compared to it, MOBIUS can handle all of these binaries.

2) Obfuscated Code: We also evaluate the effectiveness of MOBIUS in protecting the obfuscated binaries, which is not achieved by many other works [14], [33]. Using OLLVM with all strategies enabled (-mllvm -sub -mllvm -fla -mllvm -bcf), we recompile Redis and SQLite, then test them following previous ways. MOBIUS successfully rewrites them.

In fact, MOBIUS can effectively rewrite specific obfuscated binaries due to its minimal reliance on disassembly techniques and the fixed-length instruction feature of AArch64 ISA. Although code obfuscation significantly complicates static analysis of instructions and control/data flow, MOBIUS only requires identifying instructions related to the landing points. In AArch64, the linear disassembly approach of MOBIUS ensures accurate disassembly of all instructions in code sections. While this process may incorrectly disassemble inline data as instructions, as analyzed in Section III-C, the probability of disruption remains extremely low. Moreover, according to the evaluation in study [68], the proportion of inline data in AArch64 binaries is minimal. Our analysis of Redis compiled with OLLVM confirms that no inline data was introduced during this process.

RQ2: MOBIUS is robust in rewriting many real-world binaries, including specific stripped code and obfuscated code.

E. Security

1) Indirect Target Reduction: BinCFI [33] introduces the AIR (Average Indirect target Reduction) metric to evaluate the security of CFI techniques, which is defined as:

$$\text{AIR} = \frac{1}{N}\sum_{j=1}^{N}\left(1 - \frac{|T_j|}{S}\right) \tag{1}$$

N stands for the number of indirect transfers. T_j denotes the legitimate target set for the indirect transfer i_j. S represents the number of possible indirect transfer targets in an unprotected program. Since MOBIUS implements a coarse-grain CFI policy, following to the works [14], [33], we use this metric to evaluate the security of MOBIUS.

On the MOBIUS-protected binaries, for each indirect branch instruction, the legitimate targets are corresponding bti instructions. The possible target set S equals the number of instructions in code sections. Based on these, we calculate the AIR for all 44 tested applications and recompile these applications by enabling the BTI protection (via -mbranch-protection=bti) to compare MOBIUS with the GCC-BTI.

On average, MOBIUS reduces the indirect targets for 99.944%, slightly more than GCC-BTI (about 99.578%). The reason is that the land positions in the programs are determined by MOBIUS and GCC-BTI in the under-approximated and over-approximated ways, respectively. In detail, MOBIUS and GCC-BTI averagely instrument 44 and 453 bti instructions on each program, respectively. In addition to the destinations of indirect jumps, GCC-BTI instruments bti instructions at the entry of each function, even if the function is not indirectly invoked. In contrast, MOBIUS only instruments the land positions determined by profiling or inter-module calls. From the results of AIR, we can conclude that the ability of MOBIUS in reducing the indirect targets is approximately to (even slightly stronger than) that of GCC-BTI.

We also report the AIR of MOBIUS before and after obfuscation. On redis-server and speedtest, the AIR increases from 99.978% to 99.995% and 99.934% to 99.993%, respectively. This is due to the code section expansion during obfuscation.

We further evaluate the file size of the 44 binaries. The space overhead primarily comes from instrumentation code, where reduced code yields both smaller attack surfaces and lower memory overhead. Compared to rewriting-based CFI methods like BinCFI [33] (139% overhead), our rewriter only incurs a 0.9% increment of space, near to GCC-BTI.

2) False Positives and Negatives: It should be noted that due to profiling false negatives, MOBIUS makes certain compromises in BTI exception handling. Actually, MOBIUS allows the process to take an indirect branch to non-BTI instructions, which could potentially become a vulnerability in our protection. To mitigate this, MOBIUS employs heuristic frequency analysis of BTI exceptions, where threshold selection critically impacts JOP attack resistance and false positive rates.

We examine the false positives of MOBIUS across varying coverage levels under the default thresholds in Section IV-C. By analysing the log, we find that only MOBIUS-90%, MOBIUS-99%, and MOBIUS-99.99% triggered false alarms in 3, 7, and 10 benchmarks of SPEC2006, respectively. MOBIUS did not produce any false positives across all test cases. Therefore, it can be seen that the coverage has a certain effect on false positives at the given thresholds. Generally, the higher the coverage, the higher the detection accuracy of MOBIUS.

We also analyse the distributions of BTI_ins_interval and BTI_branch_interval during testing Redis. We record the number of BTI exceptions across the executed instructions during 1-minute testing of Redis. Then, we calculate the interval of the executed instructions and branches between two adjacent exceptions. Fig. 6 presents the results. From Fig. 6,

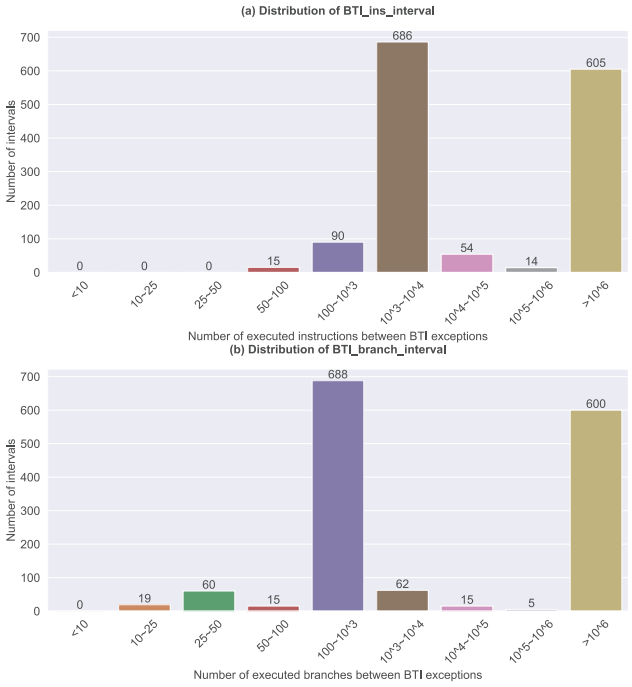


Fig. 6. The distribution of intervals between two alarms.

all intervals are longer than the default thresholds of MOBIUS. Particularly, for BTI_branch_interval, all of the intervals are much longer than the default threshold (i.e., 2 branches), which provides opportunities to further enhance MOBIUS in protecting the Redis. For BTI_ins_interval, only 15 intervals are shorter than 100. After analysing the log, we find that the exceptions during these 15 intervals occurred in the period leading up to the termination of the process, where the land positions were not recorded by MOBIUS. *After our testing, by manually adding these positions, the instruction threshold in the diagnosis function can be increased to 100 to enhance the security of* MOBIUS.

Since some CFI solutions [37], [38] detect the JOP attacks by heuristically detecting the suspicious control flows, we qualitatively compare the checking strategy in our diagnosis function with these methods. JOP-alarm [37] examines the gadget length and distance of indirect branches and uses a scoring-based approach to detect JOP. SCRAP [38] relies on a state machine which employs a counter to record the current gadget length and a comparator to decide whether the counter exceeds the threshold. These methods check all of the indirect branches. Unlike them, MOBIUS filters most indirect branches by instrumenting bti instructions. Only the unintended or missed branches will be examined. Theoretically, MOBIUS may generate fewer false positives in regular execution than them.

Moreover, according to our design, once the interval of two alarms is within the threshold, MOBIUS will terminate the process. We refer to the gadget that does not start with bti instruction as N-BTI gadget. A JOP attack which can bypass MOBIUS must satisfy one of the following conditions: 1) *There is at most one N-BTI gadget in the JOP chain;* 2) *Between any two N-BTI gadgets, the numbers of both executed instructions and branches are within the thresholds.* Generally, satisfying the first condition is similar to conducting specific attacks [42] that can bypass the coarse-grain CFI and more difficult than the second condition. For achieving the second one, it usually requires that the instructions and branches contained in the preceding N-BTI gadget are less than the thresholds of MOBIUS. From the analysis of glibc in [38], there are usually no gadgets with 8 or more instructions that update less than two states. Hence, these conditions make MOBIUS examine the JOP attacks in a strict way.

Finally, we use the attack example in Section II-A to demonstrate the ability of MOBIUS in resisting real-world attacks. Although this example is very simple, it is significantly capable of exploiting a stack overflow vulnerability to launch the unauthorized shell with hijacking only two indirect branches and utilizing only one gadget. We utilize a file which contains a character "0" as the input to implement the profiling. Then, we use MOBIUS to rewrite the binary and conduct the attack under the protection of MOBIUS. MOBIUS can detect and resist the JOP attack when the control flow is transferred to the entry of —libc_system function.

RQ3: MOBIUS reduces 99.944% indirect targets, slightly higher than GCC-BTI. Moreover, MOBIUS can detect specific JOP attacks in a high precision and strict way.

V. DISCUSSION AND LIMITATIONS

A. Coarse-Grain CFI Policy of Mobius

Though BTI provides efficient checking for the land positions of indirect branches, it would not check the sources of indirect transfers. This means the GCC-BTI and MOBIUS implement the coarse-grain CFI policy. Sophisticated attackers can still counterfeit the "legitimate" targets by selecting the gadgets starting with BTI instructions to conduct powerful attacks (e.g., COOP [42]). Therefore, in line with other coarse-grain CFI techniques, MOBIUS is weakened in resisting specific and powerful attacks, including control-flow bending and out-of-control attacks [34], [40], [41], [42].

However, this does not mean that our attempts in this work are meaningless. Generally, a stricter CFI policy usually incurs higher overhead and implementation complexity. We believe that the binaries, particularly AArch64 binaries, can benefit from the protection provided by MOBIUS cheaply and practically, which can drastically reduce the freedom of an attacker. Moreover, instrumentation-based binary-level CFI techniques may require balancing granularity against applicability. Finer-grained CFI policy demands more precise binary analysis and additional instrumentation for runtime checks, which may limit the applicability to specific binaries, such as obfuscated code. For instance, TypeArmor [14] achieves finer-grained protection with 4-bit tags than BinCFI [33] and CCFIR [9] by statically analysing and checking the number of function arguments. However, TypeArmor only analyses the register-passed arguments and cannot handle obfuscated code, hand-crafted binaries, or floating-point arguments [14]. Though recent work [7] enhances the IBT technology, it is conducted at the source level, which can provide more precise information than the binary level. Therefore, when adapting it to improve the granularity of MOBIUS, we should (1) improve binary analysis accuracy and profiling coverage, and (2) deliberate the granularity/applicability trade-offs when required.

B. Accuracy of Profiling

Generally, the false positives in profiling pose negligible concerns in our framework. In contrast, false negatives pose greater challenges. As discussed in Section III-D, the legitimate targets missed in profiling introduce unnecessary alarms during the normal execution of programs. To resolve this, we use the runtime monitor to capture and diagnose the exceptions. However, excessive missed positions may incur significant runtime overhead and bring heavy pressure to MOBIUS for distinguishing the illegal branches. Moreover, there may be insufficient testcases valid when hardening COTS binaries, particularly for network applications that take the network requests as the input. Fortunately, MOBIUS allows the users to manually add land positions after analysing the alarms. Most importantly, a fundamental capability of our analyzer is to provide CFGs for the binary rewriter, which is not limited to dynamic analysis. By integrating advanced pointer analysis techniques [10], [69], we can improve the coverage of indirect targets, which is our future work.

C. Unsupported Code

Our rewriter may not handle dynamic code such as JIT code, self-modifying code, or dynamic obfuscated code. This is also a common open issue with most binary-rewriting CFI solutions [8], [9], [12], [14], [33], [35]. Since JIT code is popular in real-world applications, trying to employ BTI technology to protect the JIT code is one of our future directions. Moreover, since our runtime monitor raises the SIGCONT signal to resume the process, MOBIUS may not support those binaries that specially register the signal handlers for the SIGCONT signal. Fortunately, in our investigation, those binaries are very rare.

D. Mobius on Other Platforms

Though the design and implementation of MOBIUS focus on the AArch64 platform, MOBIUS can be ported to other architectures, such as X86-64. Actually, Intel has proposed the IBT technology. By replacing the instructions at land positions with endbr (like bti instruction), we can try to implement MOBIUS on the X86-64 platforms. However, due to the variable-length ISA of X86-64, some instrumentations should be designed carefully, which we leave for future work.

E. Protecting the Backward Edges

As illustrated in Section I, it may be more difficult to protect the forward indirect transfers than the backward ones due to the imprecision of binary analysis. And the backward transfers usually return to the next points of function calls, which can be determined and protected by the specific stateful methods efficiently, such as the shadow stack [11]. This is one of the reasons that this paper focuses on protecting the forward edges rather than the backward ones and assumes that the backward edges have been protected in Section II-C.

Actually, Arm has proposed the PA technology to sign and verify pointers by the message authentication codes (MACs), which can be achieved by the paciasp and retaa instructions, respectively [43]. This technology usually prevents the return addresses from being modified [70]. Inspired by PA, we can extend MOBIUS to protect the backward edges on COTS binaries by rewriting them with PA instructions. A possible method is shown in Fig. 7. We can redirect all of the function calls to the trampolines, where we sign the return addresses and branch to the called functions, and verify the addresses by replacing the return instructions with the return and authentication instructions (e.g., retaa).

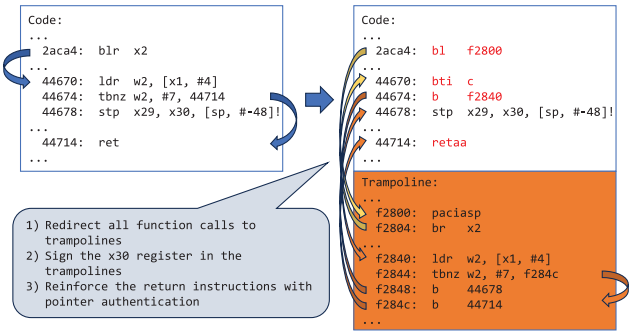


Fig. 7. A rewriting method to protect the backward edges by PA.

This extension may require additional work. Moreover, to the best of our knowledge, as of the time we completed this paper, there was no real-world device that supports both BTI and PA simultaneously. Hence, it may be difficult to test and evaluate the effectiveness and efficiency of this extension, which we leave for future work.

VI. RELATED WORK

A. Software-Based Binary-Only CFI

Binary-only CFI techniques based on software have been developed for many years. Early works were primarily based on (static or dynamic) binary rewriting [8], [9], [12], [14], [33], [35]. CCFIR [9] redirects the indirect transfers to the trampoline sections (i.e., "springboard") to protect the Windows x86 executables in a coarse-grain policy. However, its reliance on relocation information for branch target analysis and strong binary assumptions limit the applicability [71]. Furthermore, it instruments the check code in the original code sections. Compared to CCFIR, MOBIUS relies on a few assumptions regarding the binaries and modifies the original code more robustly due to the selective instruction replacement. BinCFI [33] instruments indirect branches and redirects them to address translation routines for implementing the coarse-grain CFI policy. It reserves the original code section and puts the instrumented code into a new code section, which may introduce higher space and runtime overhead than MOBIUS. To support the modular CFI policy, BinCFI modifies the source code of the loader, while MOBIUS avoids this way. Moreover, BinCFI is based on an over-approximated CFG. In contrast, MOBIUS utilizes the profiling to generate the under-approximated CFG and leaves fewer attack surfaces than BinCFI. Specific binary-only tools [8], [14] require precise binary analysis and cannot handle the obfuscated code, which may be protected by MOBIUS. LockDown [12] avoids the static analysis by using the DBI technique to enforce a dynamically constructed modular CFI policy but introduces much higher overhead than MOBIUS.

B. Hardware-Assisted CFI

With the development of hardware tracing and hardware security technologies, hardware-assisted CFI techniques have been proposed [17], [18], [19], [20], [21], [22], [70]. Some works utilize the hardware tracing technique rather than instrumentation to monitor and check the control-flow transfers, avoiding modifying or breaking the original binaries [17], [18], [19], [20], [21], [22]. However, as illustrated in Section I, post-processing hardware tracing techniques are not designed for online security protection and incur heavy decoding workloads [25]. Existing techniques based on them [18], [20], [22] have to offload the decoding and validation onto idles cores, which may be unavailable in many scenarios [26]. For instance, the evaluation in [25] has shown that the PT-based CFI tool incurs up to 652% overhead on a single core for h264ref. To avoid the heavy overhead, several works use lightweight tracing technologies (e.g., LBR) to record the transfers [17], [28]. However, since the LBR's recording capacity is limited (usually recording no more than 32 branches), specific endpoint-driven works only enforce the CFI policies at predetermined sensitive points (e.g., mprotect), which may be vulnerable to the endpoint-pruning attacks or history flushing attacks [17]. Moreover, due to the imprecise binary analysis, some hardware-assisted CFI techniques do not support the forward CFI policies [19], [28]. Compared to these works, MOBIUS incurs negligible overhead.

Several works propose new hardware designs to support CFI techniques [72], [73], [74]. However, specific tools modify the ISA to arm the core with CFI-related instructions, which makes them unable to be deployed in production environments [72], [73]. MOBIUS is more practical than them as we utilize the BTI technology existing in modern Arm processors without additional hardware features.

Since more and more processors have supported the security instructions (e.g., Intel CET, Arm PA, and Arm BTI), instruction-based protections have experienced rapid advancement. PACStack [70] presents an authenticated call stack by chained message authentication codes based on Arm PA, without requiring additional hardware. It achieves security comparable to hardware-assisted shadow stacks. Through instrumenting program code, FineIBT [7] improves the coarse-grain policies provided by the original IBT to support fine-grain CFI, which can also apply to improving Arm BTI. However, *all of the above solutions require the available source code*. Even for the binary rewriter Egalito [44], its implementation of Intel IBT is incomplete. To the best of our knowledge, **MOBIUS is the first complete implementation of binary-level CFI technique that uses security instructions within the commercial processor**.

VII. CONCLUSION

This paper presents MOBIUS, the first complete implementation of binary-level CFI technique based on the security instructions of commercial processors. MOBIUS generates the under-approximated CFGs for the binaries and shared libraries by combining the static analysis with profiling. It employs the BTI-based additive rewriting technique to correctly and efficiently instrument the programs. Then, MOBIUS monitors the BTI exceptions in runtime to diagnose the malicious indirect branches. Our evaluation shows that MOBIUS correctly rewrites many real-world applications and protects them efficiently.

REFERENCES

[1] T. Kornau et al., "Return oriented programming for the arm architecture," Master's thesis, Ruhr-Universität Bochum, Bochum, Germany, 2010.

[2] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 1–34, Mar. 2012.

[3] 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.*, Mar. 2011, pp. 30–40.

[4] S. Andersen and V. Abella, "Data execution prevention. changes to functionality in Microsoft windows XP service pack 2, part 3: Memory protection technologies," Microsoft TechNet Library, 2004. [Online]. Available: https://dl.acm.org/doi/10.1145/2857705.2857722

[5] P. Team. (2003). *PaX Address Space Layout Randomization (ASLR)*. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[6] 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.

[7] A. J. Gaidis, J. Moreira, K. Sun, A. Milburn, V. Atlidakis, and V. P. Kemerlis, "FineIBT: Fine-grain control-flow enforcement with indirect branch tracking," 2023, *arXiv:2303.16353*.

[8] M. Wang, H. Yin, A. V. Bhaskar, P. Su, and D. Feng, "Binary code continent: Finer-grained control flow integrity for stripped binaries," in *Proc. 31st Annu. Comput. Secur. Appl. Conf.*, Dec. 2015, pp. 331–340.

[9] C. Zhang et al., "Practical control flow integrity and randomization for binary executables," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 559–573.

[10] S. H. Kim, C. Sun, D. Zeng, and G. Tan, "Refining indirect call targets at the binary level," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021, pp. 1–18.

[11] 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.

[12] 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. Cham, Switzerland: Springer, 2015, pp. 144–164.

[13] 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.

[14] V. van der Veen et al., "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 934–953.

[15] S. Dinesh, N. Burow, D. Xu, and M. Payer, "RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1497–1511.

[16] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 1683–1700.

[17] V. Van der Veen et al., "Practical context-sensitive CFI," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 927–940.

[18] 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.

[19] 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.

[20] 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.

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

[22] H. Hu et al., "Enforcing unique code target property for control-flow integrity," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 1470–1486.

[23] A. Kleen and B. Strong, "Intel processor trace on Linux," *Tracing Summit 2015*, Montreal, QC, Canada, 2015. [Online]. Available: https://tracingsummit.org/ts/2015/

[24] Arm.(2016). *Arm CoreSight SoC-400 Technical Reference Manual*. [Online]. Available: https://developer.arm.com/documentation/100536/latest/

[25] S. Canakci, L. Delshadtehrani, B. Zhou, A. Joshi, and M. Egele, "Efficient context-sensitive CFI enforcement through a hardware monitor," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, Jan. 2020, pp. 259–279.

[26] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, "Adaptive call-site sensitive control flow integrity," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Jun. 2019, pp. 95–110.

[27] Intel.(2024). *Intel 64 and Ia-32 Architectures Software Developer Manuals*. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[28] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *Proc. 22nd USENIX Secur. Symp. (USENIX Secur. 13)*, Aug. 2013, pp. 447–462.

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

[30] Z. Ning and F. Zhang, "Understanding the security of ARM debugging features," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 602–619.

[31] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur. 17)*, Aug. 2017, pp. 557–574.

[32] Z. Zhang, S. Natarajan, and A. Banerjee, "Detecting process hijacking and software supply chain attacks using Intel threat detection technology," Intel, Santa Clara, CA, USA, Tech. Rep. 222.

[33] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proc. 22nd USENIX Secur. Symp.*, 2013, pp. 337–352.

[34] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. Groß, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proc. 24th USENIX Secur. Symp. (USENIX Secur. 15)*, Aug. 2015, pp. 161–176.

[35] A. Prakash, X. Hu, and H. Yin, "VfGuard: Strict protection for virtual function calls in COTS C++ binaries," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.

[36] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert, "CFI: Type-assisted control flow integrity for x86–64 binaries," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*. Cham, Switzerland: Springer, 2018, pp. 423–444.

[37] F. Yao, J. Chen, and G. Venkataramani, "JOP-alarm: Detecting jump-oriented programming-based anomalies in applications," in *Proc. IEEE 31st Int. Conf. Comput. Design (ICCD)*, Oct. 2013, pp. 467–470.

[38] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "SCRAP: Architecture for signature-based protection from code reuse attacks," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2013, pp. 258–269.

[39] Z. Shen and J. Criswell, "InversOS: Efficient control-flow protection for AArch64 applications with privilege inversion," 2023, *arXiv:2304.08717*.

[40] M. Conti et al., "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.

[41] 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.

[42] 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*, 2015, pp. 745–762.

[43] Arm.(2023). *Arm Architecture Reference Manual for A-profile Architecture*. [Online]. Available: https://developer.arm.com/documentation/ddi0487/ja/?lang=en

[44] D. Williams-King et al., "Egalito: Layout-agnostic binary recompilation," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, New York, NY, USA, Mar. 2020, pp. 133–147.

[45] A. Altinay et al., "BinRec: Dynamic binary lifting and recompilation," in *Proc. 15th Eur. Conf. Comput. Syst.*, Apr. 2020, pp. 1–16.

[46] Github.(2025). *The CFI Function of Etharden is Incomplete*. [Online]. Available: https://github.com/columbia/egalito/issues/44

[47] A. Flores-Montoya and E. Schulte, "Datalog disassembly," in *Proc. 29th USENIX Secur. Symp.*, Jun. 2019, pp. 1075–1092.

[48] E. Schulte, M. D. Brown, and V. Folts, "A broad comparative evaluation of x86–64 binary rewriters," in *Proc. 15th Workshop Cyber Secur. Experimentation Test*, Aug. 2022, pp. 129–144.

[49] T. Kim et al., "RevARM: A platform-agnostic ARM binary rewriter for security applications," in *Proc. 33rd Annu. Comput. Secur. Appl. Conf.*, Dec. 2017, pp. 412–424.

[50] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2020, pp. 151–163.

[51] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attacks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, 2014, pp. 1–16.

[52] 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.

[53] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++: Combining incremental steps of fuzzing research," in *Proc. 14th USENIX Workshop Offensive Technol. (WOOT 20)*, 2020, pp. 1–12.

[54] T. Yue et al., "EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit," in *Proc. 29th USENIX Secur. Symp.*, Aug. 2020, pp. 2307–2324.

[55] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. OSDI*, vol. 8, Dec. 2008, pp. 209–224.

[56] Y. Shoshitaishvili et al., "SOK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 138–157.

[57] C.-K. Luk, "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, 2005.

[58] D. L. Bruening and S. Amarasinghe, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Inst. Technol., Cambridge, MA, USA, 2004.

[59] M. Jiang, Y. Zhou, X. Luo, R. Wang, Y. Liu, and K. Ren, "An empirical study on ARM disassembly tools," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2020, pp. 401–414.

[60] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: Debloating binary shared libraries," in *Proc. 35th Annu. Comput. Secur. Appl. Conf.*, Dec. 2019, pp. 70–83.

[61] L. Di Bartolomeo, H. Moghaddas, and M. Payer, "Armore: Pushing love back into binaries," in *Proc. 32nd USENIX Secur. Symp.*, 2023, pp. 6311–6328.

[62] A. Sadeghi, S. Niksefat, and M. Rostamipour, "Pure-call oriented programming (PCOP): Chaining the gadgets using call instructions," *J. Comput. Virol. Hacking Techn.*, vol. 14, no. 2, pp. 139–156, May 2018.

[63] U. F. Mayer. (2003). *Linux/Unix nbench*. [Online]. Available: https://www.math.utah.edu/~mayer/linux/bmark.html

[64] 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 *Proc. 28th USENIX Conf. Secur. Symp.*, 2019, pp. 177–194.

[65] Y. Deng et al., "StrongBox: A GPU TEE on arm endpoints," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2022, pp. 769–783.

[66] A. Davanian, Z. Qi, Y. Qu, and H. Yin, "DECAF++: Elastic whole-system dynamic taint analysis," in *Proc. 22nd Int. Symp. Res. Attacks, Intrusions Defenses (RAID)*, Jan. 2019, pp. 31–45.

[67] Y. Jia et al., "HyperEnclave: An open and cross-platform trusted execution environment," in *Proc. USENIX Annu. Tech. Conf.*, Jan. 2022, pp. 437–454.

[68] Y. Ye, Z. Zhang, Q. Shi, Y. Aafer, and X. Zhang, "D-ARM: Disassembling ARM binaries by lightweight superset instruction interpretation and graph modeling," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2023, pp. 2391–2408.

[69] Q. Shen et al., "An improved method on static binary analysis to enhance the context-sensitive CFI," *Int. J. Comput. Inf. Eng.*, vol. 11, no. 7, pp. 848–854, Jun. 2017.

[70] H. Liljestrand, T. Nyman, L. J. Gunn, J. Ekberg, and N. Asokan, "PACStack: An authenticated call stack," in *Proc. 30th USENIX Secur. Symp.*, Aug. 2021, pp. 357–374.
[71] S. Priyadarshan, "A study of binary instrumentation techniques," Stony Brook Univ., Res. Proficiency Rep., 2019. [Online]. Available: http://seclab.cs.sunysb.edu/seclab/rareasdesc__bintx.html
[72] L. Davi et al., "HAFIX: Hardware-assisted flow integrity eXtension," in *Proc. 52nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2015, pp. 1–6.
[73] Y. Wang, J. Wu, T. Yue, Z. Ning, and F. Zhang, "RetTag: Hardware-assisted return address integrity on RISC-V," in *Proc. 15th Eur. Workshop Syst. Secur.*, New York, NY, USA, Apr. 2022, pp. 50–56.
[74] 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.

Tai Yue received the B.S. degree from the Department of Mathematics, Nanjing University, Nanjing, in 2017, and the M.S. and Ph.D. degrees from the College of Computer, National University of Defense Technology, Changsha, in 2019 and 2024, respectively. He is currently an Assistant Researcher at the Academy of Military Science. His research interests include system security, software security, and hardware-assisted security.

Kai Lu received the B.S. and Ph.D. degrees in computer science and technology from the College of Computer, National University of Defense Technology, Changsha, in 1995 and 1999, respectively. He is currently a Professor with the College of Computer, National University of Defense Technology. His research interests include operating systems, parallel computing, and security.

Zhenyu Ning received the Ph.D. degree in computer science from Wayne State University in 2020. He is currently an Associate Professor at Hunan University. His research interests include security and privacy, such as system security, mobile security, the IoT security, trusted execution environment, and hardware-assisted security.

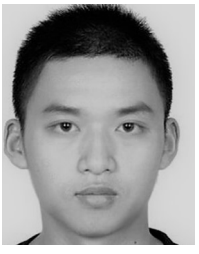
Pengfei Wang received the B.S., M.S., and Ph.D. degrees in computer science and technology from the College of Computer, National University of Defense Technology, Changsha, in 2011, 2013, and 2018 respectively. He is currently an Associate Professor with the College of Computer, National University of Defense Technology. His research interests include operating systems and software testing.

Lei Zhou received the Ph.D. degree in computer science from Central South University. He is currently a Research Associate with the College of Computer, National University of Defense Technology. His primary research interests include systems security, such as trustworthy execution, hardware-assisted security, and memory forensics.

Xu Zhou received the B.S., M.S., and Ph.D. degree from the School of Computer Science, National University of Defense Technology, China, in 2007, 2009, and 2013, respectively. He is currently an Associate Professor with the School of Computer Science, National University of Defense Technology. His research interests include operating systems and security.

Yaohua Wang is currently a Professor with the College of Computer Science, National University of Defense Technology. His research interests include computer architecture and machine learning and security. His work spans and stretches the boundaries of computer architecture. He is especially excited about novel, fundamentally-efficient computation, and memory/storage paradigms, applied to emerging machine learning applications.

Fengwei Zhang (Senior Member, IEEE) received the Ph.D. degree in computer science from George Mason University. He is currently an Associate Professor with the Department of Computer Science and Engineering, Southern University of Science and Technology (SUSTech). His primary research interests include systems security, with a focus on trustworthy execution, hardware-assisted security, debugging transparency, transportation security, and plausible deniability encryption.



Gen Zhang received the Ph.D. degree in computer science and technology from NUDT in 2022. His research interests include fuzzing and testing.