BINSWEEP: Reliably Restricting Untrusted Instruction Streams with Static Binary Analysis and Control-Flow Integrity

Matteo Oldani
William Blair
Lukas Stadler
Zbyněk Šlachrt
Matthias Neugschwandtner
Oracle America Inc.
Austin, Texas, United States

Abstract

Restricting an application's instruction stream is necessary to ensure the absence of certain functionality, which in turn is a requirement for lightweight sandboxing of untrusted code in cloud environments. Doing so at the lowest possible level, (i.e., machine code), is safest as it does not assume trusted or bug-free build toolchains. However, resolving indirect branches and instruction set architectures (ISA) with variable-length instructions are a challenge for reliable and exhaustive machine code analysis.

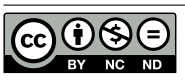
In this paper, we present BINSWEEP, a system that ensures complete analysis of variable-length ISA applications in machine code. The key enabling concept is a restricted form of Control Flow Integrity (CFI) that BINSWEEP enforces, called BINSWEEP$_{CFI}$. We implement BINSWEEP$_{CFI}$ as a compiler pass within the LLVM toolchain. Our evaluation over SPECint benchmarks in SPEC CPU 2017, and widely used binary programs, including the `NGINX` webserver, Micronaut service, and Python interpreters, demonstrates that BINSWEEP can verify real world programs, and BINSWEEP$_{CFI}$ can protect programs with manageable (6.55% in the worst case) performance overhead. Furthermore, we show BINSWEEP can verify these programs' CFGs much faster than a state of the art binary analysis tool, angr, can recover CFGs. These results demonstrate BINSWEEP can efficiently support admitting untrusted code buffers, hundreds of megabytes in size, to cloud sandboxes.

CCS Concepts

• **Security and privacy → Virtualization and security**; **Information flow control**; **Software security engineering**; • **Software and its engineering → Compilers**.

Keywords

Intra-Process Isolation, Control-Flow Integrity, Static Binary Analysis, Cloud Security



CCSW '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1234-0/24/10
https://doi.org/10.1145/3689938.3694778

ACM Reference Format:
Matteo Oldani, William Blair, Lukas Stadler, Zbyněk Šlachrt, and Matthias Neugschwandtner. 2024. BINSWEEP: Reliably Restricting Untrusted Instruction Streams with Static Binary Analysis and Control-Flow Integrity. In *Proceedings of the 2024 Cloud Computing Security Workshop (CCSW '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3689938.3694778

1 Introduction

A significant amount of past research deals with static analysis of applications in form of binary executables for malware analysis and reverse engineering [6, 9, 18]. However, static analysis is also used as a gatekeeper for runtime environments:

Both cloud computing environments as well as mobile platforms have an interest to tightly control the application code they are running to avoid malicious applications from compromising the system. To this end, applications published on the corresponding official application distribution platforms undergo an automated review process that inspects applications for malicious behavior. Restrictions on higher-level constructs such as source code or intermediate representations such as JVM bytecode or LLVM IR is insufficient, however, as the final lowering step to machine code can still introduce unintended code patterns or unknown bugs [7]. This leaves validation at the machine code level as the final option. However, in the presence of malicious input this is a non-trivial task.

So-called Jekyll apps [31, 34] have shown to be able to circumvent the review process by hiding malicious functionality in a seemingly benign instruction stream by exploiting the main challenges in static analysis of ISAs with variable-length instructions: First, an instruction stream decodes to different instructions depending on the offset where decoding is started. If the instruction stream has a two-byte instruction at offset zero and decoding starts at offset one, a misaligned, alternate instruction stream is decoded. Second, indirect branches can have targets that are derived based on the runtime state and may depend on input data. Combined, these two challenges allow hiding misaligned instructions in a seemingly benign instruction stream that are only reachable through an indirect branch executed at runtime, similar to return-oriented-programming (ROP) gadgets.

This attack vector is particularly relevant in light of the emerging interest in lightweight user-mode native code sandboxing techniques for the cloud, enabled through Intel Memory Protection Keys (MPK) [26, 27] and their ARM counterpart Permission Overlay Extensions (POE) [20]. Systems such as Erim [33], Hodor [15], Faastlane [17], and Jenny [28] leverage these hardware mechanisms to

isolate workloads. A common aspect of using these hardware mechanisms for workload isolation is that the instruction to change memory access permissions cannot be issued by the workload itself, (i.e., it is considered privileged and cannot be part of the workloads instruction stream). Research has demonstrated that ensuring this is a challenge in itself [12].

A naive solution to this issue would be decoding the instruction stream at byte-by-byte offsets. However, this would likely lead to numerous false positives as many instructions, which are actually unreachable, are mistakenly disassembled and filtered. This naive approach cannot scale to analyze real-world applications in general.

Control-flow integrity (CFI) [4] limits the number of legitimate targets for indirect branches. This naturally reduces the number of executable paths a successful exploit can take in a victim program. CFI has gained enough momentum over the years that CPU manufacturers even introduced silicon support in the form of ISA extensions. This has taken the form of Control-Flow Enforcement Technology (CET) on Intel and AMD CPUs [16, 29] as well as branch target identification (BTI) and guarded control stack (GCS) on ARM [22, 35]. CFI has mostly been positioned as a means for exploit mitigation, with its effectiveness largely depending on the accuracy of the underlying control-flow information [10]. In the case of binary executables, recovering information on legitimate control flows from the instruction stream represents a challenging static analysis problem.

With Binsweep we turn the tables: by making CFI a precondition for static analysis, we can reliably traverse all possible execution paths. Compared to the use of CFI in preventing exploits, we only require a relaxed form that indicates the targets of all indirect branches in general. At the same time, Binsweep also enforces CFI to ensure that our analysis is indeed complete.

In this paper, we make the following contributions:

- Binsweep, a novel approach to static binary analysis that ensures completeness when used in combination with control-flow integrity.
- A software implementation of CFI in LLVM for the x86_64 architecture that satisfies the CFI requirements for Binsweep.
- A performance evaluation of Binsweep and our CFI implementation on real world executables, including `NGINX`, a Micronaut service, and Python interpreters. We include a comparison with angr, a state of the art binary analysis framework [30], and demonstrate that Binsweep recovers CFGs substantially faster than angr, and often can verify executables hundreds of megabytes in size in less than 20 seconds.

2 Background and Threat Model

In this section, we provide background on binary analysis and control-flow integrity, two of the key techniques used by Binsweep. Furthermore, we present the threat model we assume in this work.

2.1 Control-Flow Integrity

Control-flow integrity (CFI) was first proposed as a technique to ensure all execution paths for a running program conform to the paths given in the program's original representation. That is, if an adversary were to subvert a program's control-flow, either by exploiting a corrupted stored instruction pointer on the stack or overwriting function pointers stored in objects to hijack forward-edges, the CFI mechanism could detect the subversion and terminate the program before an adversary could gain control of the running program.

This has spawned numerous prototypes in software that have historically suffered high performance penalties. However, hardware manufacturers such as Intel and ARM have recently introduced CFI features into their CPUs to implement both backward-edge (i.e., returns) and forward-edge (i.e., indirect jumps) CFI. Unfortunately, even though CFI features may be present on hardware, adopting new hardware models at scale comes at a significant cost. Furthermore, the entire toolchain, including compilers and operating systems, must add support for new features in order for programs to benefit for them. Our software based approach, described in Section 3 strikes a balance between a fully software CFI solution and the new hardware CFI features that may not be readily available or supported yet by commodity operating systems.

For example, the latest version of Linux, as of this writing, still does not support indirect branch tracking (IBT) available on Intel CPUs. This is not a fault of Linux, but rather an example of how much consideration and effort must go into properly configuring security features in hardware before they can be readily used in commodity computing environments. In this work, we adopt a similar approach to IBT to implement a software CFI solution. Like IBT, we mark all valid indirect branch targets with a special `ENDBR64` instruction, which executes as a wide `NOP` on unsupported CPUs. We then introduce a toolchain that emits a software CFI check for the branch target at every indirect branch. This emitted software check halts the process if the destination of the indirect branch does not contain a `ENDBR64` instruction.

2.2 Binary Analysis

Binary analysis tools seek to determine properties of programs represented as the binary executable programs produced by compilers and linkers, interpreted by dynamic loaders and operating systems, and executed directly on CPUs. The low level of abstraction used by binary programs complicates even the simplest program analysis tasks. For example, simply recovering the control-flow graph (CFG) of executables that run on commodity CPUs (i.e., x86), is intractable in general, due to the inability to define a bijection between CFGs and machine code sequences. A CFG is a graph data-structure where individual nodes represent basic blocks, or instruction sequences that either end with, or are referred to by a jump instruction. The edges in the CFG denote control flow transfers between basic blocks, such as a direct jump, indirect jump, or return instruction. Typically, the task of recovering CFGs from executable binaries involves recognizing function definitions within the binary, recovering CFGs at each routine and the executable's entrypoint, and then analyzing the lifted CFGs.

Despite the undecidable nature of the domain, practical binary analysis tools are able to reconstruct accurate CFGs from binary programs to be consumed manually by a human analyst or automatically by analysis passes implemented in software. In this work, we adopt static binary analysis for the latter use case to detect, and reject, malicious instruction sequences that may be concealed within the arbitrary machine code held by an executable's code section (see Section 3). To circumvent the intractability of this problem in general, we enforce our proposed software CFI security check on every executable snippet while enumerating the CFGs stored within an executable (see Section 3). In our approach, the task of recognizing

a valid entrypoint is simplified by our software CFI approach (see Section 3.2). This allows us to reject non-conforming executables, and, furthermore, allows operators to design flexible security policies that enable admitting untrusted programs to a variety of use cases (e.g., an intra-process isolation sandbox). The policy for an intra-process isolation sandbox could deny access to all system resources by rejecting all syscalls within an untrusted code buffer to be run inside the sandbox.

2.3 Threat Model

In this work, we assume the following threat model:

- An adversary can pass arbitrary machine code to BINSWEEP. This machine code is entirely untrusted, no assumptions are made about its content, i.e., it also does not matter if the machine code conforms to our software CFI approach. By default, BINSWEEP rejects code that does not conform to our software CFI approach.
- All code that is reachable through indirect branches is available to BINSWEEP at analysis time. This requirement is trivially fulfilled if the code does not contain any indirect branches. Otherwise, orthogonal security mechanisms can be employed to limit reachability. In case such mechanisms are software-based, they can in turn be expressed as a policy checked by BINSWEEP.
- BINSWEEP rejects code that does not comply with the specified security policies. This includes CFI enforcement as well as any operator-specified policy.
- The adversary's goal is to bypass the security policies enforced by BINSWEEP. A fundamental goal is to circumvent the CFI enforcement. Other goals may be deployment-specific, for example, an adversary may want to change memory protection settings or invoke a malicious system call when the code is run within a restricted sandbox.

The definition of the environment where untrusted code is run is up to the operators that rely on BINSWEEP. That is, any use case that requires running code of varying trust can benefit from BINSWEEP. The constraints for a given level of trust can be expressed as a policy, up to the limits of the policy language, and enforced by BINSWEEP. Following a scan of all the basic blocks in a code buffer's CFGs, BINSWEEP "admits" a code buffer if no policy violations are observed. Otherwise, the code buffer is rejected. Like many security features, the notion of what can be trusted is naturally dependent on a given use case. We argue that the policy language provided by BINSWEEP combined with its exhaustive scanning of executable CFGs with CFI, provides a flexible platform for vetting code with varying degrees of trustworthiness.

In a real deployment of BINSWEEP, orthogonal isolation techniques may be employed to limit the code reachable at runtime to code analyzed by BINSWEEP. This allows collocating both trusted and untrusted code in the same operating system process within an intra-process isolation sandbox. Figure 1 provides an example of such a sandbox where BINSWEEP, paired with a memory isolation mechanism (e.g., memory protection keys (MPK)), implements horizontal (between individual tenants) and vertical (between tenants and a trusted sandbox visor) security boundaries within a shared operating system process. That is, BINSWEEP rejects any code buffer

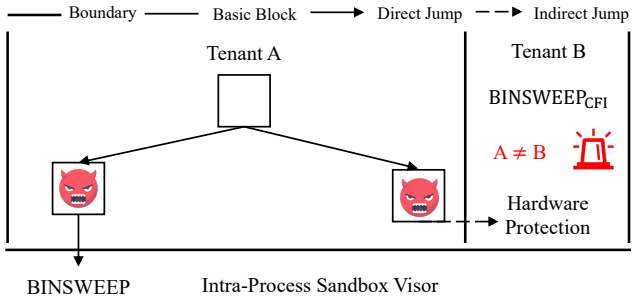


Figure 1: Example deployment of BINSWEEP to enforce horizontal and vertical security boundaries in an intra-process isolation sandbox with memory protection keys (MPK) or a similar hardware mechanism. BINSWEEP can both ensure the absence of `wrpkru` in an untrusted instruction stream as well as deny jumps outside tenant A's domain.

that directly jumps outside of the buffer or tries to change the code's assigned memory protection settings. Furthermore, the sandbox terminates any thread that attempts to indirectly jump to a valid branch within another tenant.

Each tenant contains one or more operating system threads, which execute the code mapped into each tenant. The domain associated with each tenant is given in the lower right hand corner of each tenant in Figure 1. The protection against direct jumps is provided by scanning code buffers with BINSWEEP before loading the code into the sandbox. Rejecting indirect jumps to different tenants, or the trusted visor, follows from using BINSWEEP$_{CFI}$ with a memory isolation mechanism like MPKs. When MPKs are available, the sandbox visor can assign each tenant an MPK domain, bind all tenant code pages to the tenant's domain, and limit each tenant thread's memory access to the tenant's domain. Such a combination of BINSWEEP$_{CFI}$ and MPKs will cause the CFI check emitted by BINSWEEP (see Section 3.2) to trigger a segmentation violation whenever an indirect jump occurs to another tenant's code region. If this occurs, the MPK hardware will observe a deviation from the current thread's permissions and the domain of the accessed pages (i.e., domain $A \neq$ domain B in Figure 1), raise a segmentation violation, and the sandbox visor will terminate the offending thread in response. Note that this scheme requires that BINSWEEP reject any untrusted code buffer that contains a path to a `wrpkru` instruction, which allows a thread to alter its domain. This can be trivially implemented as a BINSWEEP policy.

3 System Overview

In this section, we provide a high level overview of BINSWEEP. This includes a recursive descent sweeping procedure to enforce individual security policies. We further propose a software control-flow integrity (CFI) technique (referred to as BINSWEEP$_{CFI}$) to be used in conjunction with BINSWEEP.

Figure 2 provides a high-level overview of BINSWEEP. BINSWEEP begins by scanning an untrusted input buffer. This input buffer represents a byte stream. The contents of this byte stream are individual instructions in the host's instruction set architecture (ISA).

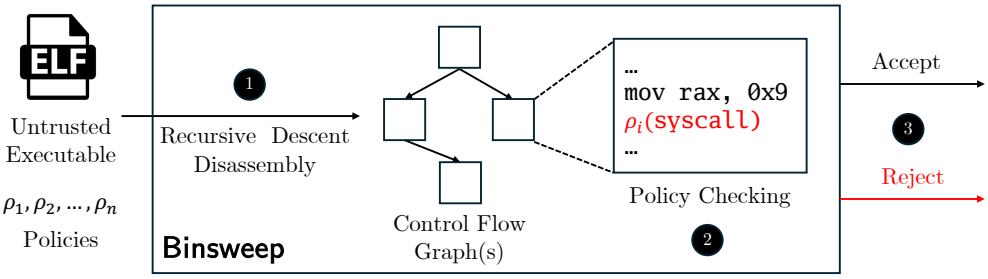


Figure 2: Architectural overview of Binsweep scanning an executable for policy violations.

Within this stream, a set of possible entrypoints must be identified. After these entry points are found, Binsweep performs recursive descent disassembly until all possible code paths are traversed (step 1). During traversal, Binsweep applies a configurable set of security policies to each instruction. This allows Binsweep to restrict the contents of the instruction stream in a modular way (i.e., new restrictions can be imposed by authoring orthogonal policies). Note each policy is allowed to maintain its own memory (i.e., keep a snapshot of the instruction stream). This is helpful for stateful policies that may require more context than enforcing a simple allowlist to reject illegal instructions (step 2). After all basic blocks have been scanned, the executable is accepted if no policy violations occur, and rejected otherwise (step 3).

3.1 Recursive Descent Sweeping

After a program has been compiled to use Binsweep$_{\text{CFI}}$, we enforce security policies by performing recursive descent sweeping over the code buffers' CFGs. At a high level, "sweeping" the binary in this way allows Binsweep to vet every basic block reachable in these CFGs when the code buffer is executed under Binsweep$_{\text{CFI}}$. The disassembly process is divided into two steps and follows a combination of two algorithms: recursive descent and linear sweeping. The final disassembled program is organized into one or more control-flow graph (CFG) structures. Within the CFGs, each instruction is saved within a map. The address of the instruction acts as the index for each saved instruction. Moreover, each decoded instruction maintains a reference to the next instruction and one or more previous instructions in the sequence. This allows Binsweep (and individual policies) a simple way to walk forward or backward while "sweeping" the buffer. Exceptions to this rule are given by the following:

- Entrypoints, which either contain the end branch instruction or are referred to by direct jumps, do not have a previous instruction.
- Terminating instructions, such as returns or interrupts as well as the instruction stream's final instruction, do not have a next instruction.

Additionally, branching instructions may also have a target instruction. To be precise:

- Conditional branches contain two references to next instructions. These include the branch target if the condition is satisfied and the next instruction if the condition is false.
- Call instructions refer to the branch target and the instruction following the call, which acts as a return address.

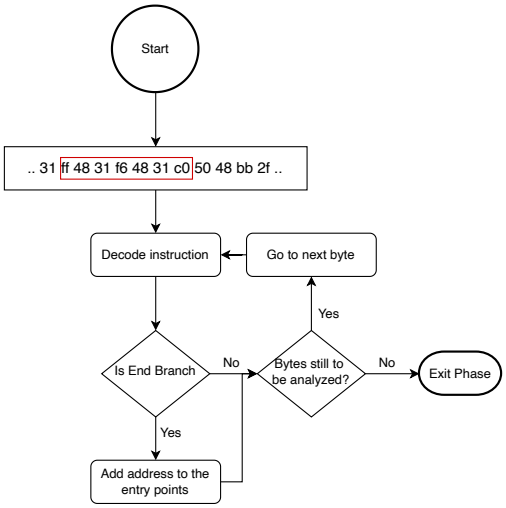


Figure 3: Detecting entrypoints with Binsweep.

- Direct jumps simply refer to the jump's target.

Figure 3 visualizes the first step in the disassembly process. This scan finds all possible entrypoints which begin with the end branch instruction. Recall the end branch instruction represents a valid branch target in the CFI mechanism used with Binsweep. A naive approach for discovering entrypoints may simply take the first instruction of the code buffer, identify individual functions within the buffer, or begin recursive descent from a known starting entrypoint. However, this naive approach fails to detect valid entrypoints concealed within data, such as those stored within 64-bit immediate values. Adversaries can use these hidden entrypoints to evade a naive scanner and execute malicious code. For this reason, Binsweep attempts to decode an instruction at every byte offset in the untrusted code buffer to detect any intended and unintended entrypoints. This ensures that, even if an attack redirects the control-flow to jump to an arbitrary entrypoint, this execution path is vetted by Binsweep's static analysis. The executed instructions are permitted if they are allowed by the policies given to Binsweep.

Figure 5 defines a simplified ISA relevant to verifying instruction buffers. In addition, the context maintained by Binsweep is defined as well. Note that this syntax is not bound to a particular ISA. Instead, the syntax captures streams of "straight-line" instructions which,

when executed, cause the CPU to increment the program counter (pc) and execute the next instruction, given by the "successor" of the program counter *succ pc*. During this initial scan, each entrypoint is added to the queue Q that represents the entrypoints from which to begin recursive descent sweeping.

This recursive descent detects illegal instruction sequences embedded within the code buffer's CFGs. Figure 6 provides the operational semantics of recursive descent sweeping over a binary's code section. This vets the instructions reachable in all the CFGs contained within the code buffer's entrypoints under a set of policies given by $\mathcal{P}$. Note that this procedure is represented incrementally (i.e., small-step semantics), with big-step semantics represented by $\Downarrow$. For example, when verifying a conditional jump, the result for the conditional jump is denoted by the conjunction of Binsweep's findings for both possible branches (i.e., both must be valid under $\mathcal{P}$).

A simple example policy can be given by $\rho_{\text{Allowlist}}$ which simply implements an instruction allowlist to reject undesired instructions (e.g., `syscall`). $\rho_{\text{Allowlist}}$ could be implemented as simply checking whether the instruction given at the program counter $\iota(pc)$ belongs to the set of allowed instructions ($\iota(pc) \in \mathcal{A}$) where $\mathcal{A}$ is a set that represents the allowlist. Since predecessor and successor instructions are accessible via the program counter pc, more complex policies can be readily implemented and provided to Binsweep. The context maintained during recursive descent sweeping is represented by a 4-tuple ctx where pc is an address ℓ within the code buffer, Q is the queue of addresses to be disassembled, Λ is the set of all addresses in the code buffer visited by recursive descent (in order to avoid loops), and $\mathcal{P}$ denotes the set of policies enforced by Binsweep. Note that the CFI policy ρ_{CFI} is always enabled.

Figure 4 visualizes recursive descent sweeping's operational semantics. This performs recursive descent scanning over all the entrypoints given within Q which is defined in the algorithm's first step.

Binsweep recursively scans each address ℓ in Q until Q is empty (i.e., the "Last Return" case in the semantics). Note that **ret** may be substituted for another terminating instruction that represents the end of a basic block (i.e., a halting instruction). Moreover, this queue of addresses is updated each time the instruction at an address is decoded (or "visited") for the first time (see "Allowed Op" in the semantics). Thus, in the end, Q contains not only the canonical entrypoints, represented by the chosen end branch instruction, but also the addresses of all direct jumps observed during recursive descent. For each address ℓ in Q, Binsweep obtains the corresponding instruction at the address while consulting the map of visited instructions *visited* to process the instruction. The processing consists of the following cases:

- The instruction is out of range, meaning that it was impossible to decode the instruction since it is located outside the code buffer. In this case, a special placeholder instruction has already been created. Thus, Binsweep simply continues to the next starting position to be processed. The detection of this kind of instruction is delegated to the verification phase.
- The instruction is a terminating instruction, for example a `RET`, `INT3`, or `HLT` in x86. In this case, Binsweep simply proceeds with the next address in the queue. We do this because terminating instructions require no further processing since they lack a reference to a successor instruction by definition. We consider instructions that trigger an exception (i.e., `INT3` and `HLT` in x86) as terminating since they will always raise a signal when executed in user mode. Whether these instructions are valid is determined by the verification phase (i.e., the security policies $\mathcal{P}$ passed to Binsweep).
- The instruction is a branch. This can be split into the following sub-cases:
  - An indirect unconditional branch simply causes Binsweep to go to the next address in Q since we cannot compute the target address. In the unconditional branch case, the next address to process is simply the instruction's successor. Note that these instructions are restricted by $\text{Binsweep}_{\text{CFI}}$. Thus, they may only jump to one of the entrypoints discovered by the initial phase.
  - An indirect call is considered a normal non-branching instruction. As with unconditional indirect branches, we cannot compute the target of an indirect call. However, CFI guarantees that the call will land on a known entrypoint.
  - A conditional branch will have its target address ℓ decoded. The target address will be added to Q. Next, Binsweep continues with the computation of the conditional branch's successor instruction (i.e., the branch not taken).

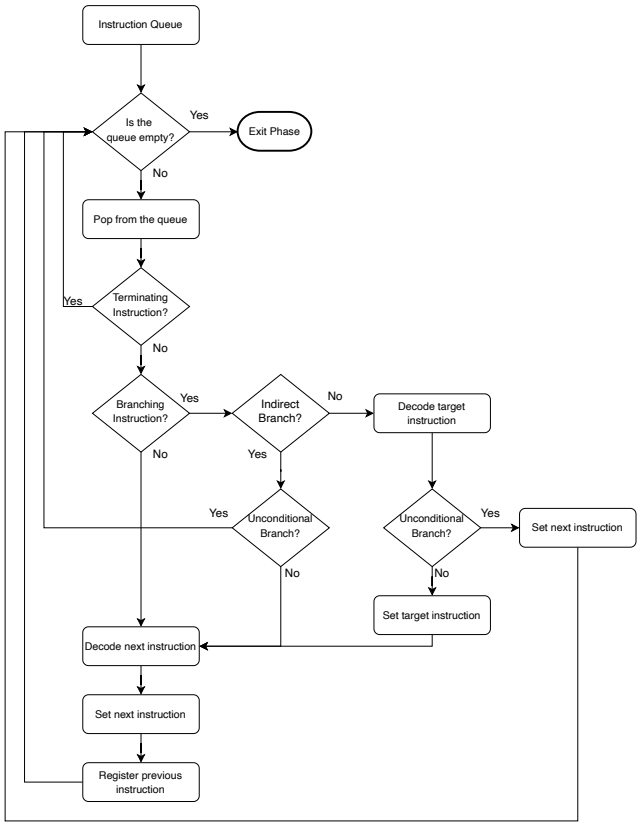


Figure 4: Processing instruction streams with Binsweep.

program	$P ::= \bar{\iota}$
instruction	$\iota ::= \mathbf{op} \mid \mathbf{endbranch} \mid \mathbf{jmp}\ \ell \mid \mathbf{jz}\ \ell \mid \mathbf{ret}$
op	$\mathbf{op} ::= \mathbf{mov}\ \sigma, \sigma \mid \mathbf{add}\ \sigma, \sigma \mid \mathbf{cmp}\ \sigma, \sigma \mid \mathbf{jmp}\ r \mid \mathbf{halt}$
operand	$\sigma ::= r \mid \ell \mid v$
locations	$\ell ::=$ memory addresses
registers	r
values	v
program counter	pc
policies	$\mathcal{P} : \{\rho_1, \rho_2, ..., \rho_n\}$
queue	$Q : \ell_1 :: \ell_2 :: ... :: \ell_n$
visited	$\Lambda : \{\ell\}$

Figure 5: A syntax for a simple instruction set architecture (ISA).

  - An unconditional direct branch has its target address decoded and set as the successor instruction. Once a terminating instruction is encountered, Binsweep continues disassembling the address at the top of Q (i.e., τ_1).
- In all the other cases, the instruction's successor is decoded. The address of the successor is computed by adding the length of the current instruction to its address in the code buffer. Once a terminating instruction is encountered, Binsweep recursively runs this phase by processing the address τ_1 at the top of Q.

In addition to the disassembling algorithm, there are two additional checks on the code buffer to ensure the disassembly is sound. Binsweep verifies that no end branch instruction is placed outside of the executable section of an executable file. This ensures that unaligned memory mappings at boundaries cannot produce valid unintended entrypoints.

3.2 Software Control-Flow Integrity

Binsweep assumes that all valid entrypoints to the code are marked with a unique instruction (i.e., a specific byte pattern). In addition, all indirect branch targets are also marked with this unique instruction. Note that the same instruction can be used for both entrypoints (i.e., valid targets that begin executing the instruction stream as well as indirect branch targets within the stream).

Binsweep assumes that a CFI mechanism is present when executing the untrusted code buffer. This CFI mechanism ensures that indirect branches can only land at locations that contain the unique instruction that denotes a branch target. This CFI assumption can be enforced either by hardware-assisted technologies, such as Intel Control-Flow Enforcing Technology (CET), or using a software implementation. The former marks all the forward-edge entrypoints (i.e., the destination of a call) with a special instruction, `ENDBR64`. All the the backward-edge targets (i.e., valid return locations) are guarded by a privileged shadow stack. In case of a mismatch, either due to a missed ENDBR64 or divergence of the stack and shadow stack, the CPU traps and the kernel kills the process.

While Binsweep can verify code buffers using an existing CFI mechanism that implements both backward and forward-edge CFI, we propose Binsweep$_{\text{CFI}}$ as a software CFI solution. Binsweep$_{\text{CFI}}$ can also represent all valid forward-edge targets with a special end branch instruction. In our setting, we can simply reuse `ENDBR64` which evaluates to a wide `NOP` instruction on hardware that lacks support for Intel Control-Flow Enforcement Technology (CET).

When hardware lacks support for a shadow stack, software CFI can substitute all return instructions with indirect jumps back to the stored instruction pointer. Finally, before each indirect jump, software CFI looks ahead at the memory contents of the branch target to ensure the presence of the unique instruction that represents the beginning of a valid branch target (i.e., `ENDBR64`). The advantage of using Binsweep$_{\text{CFI}}$ is that Binsweep can check the validity, via static analysis, of the software CFI's implementation. Binsweep does this by enforcing a policy ρ_{CFI}, enabled by default, to confirm that required control-flow patterns are correctly generated in the code buffer and not vulnerable to tampering with interleaved instructions (see Section 3.3).

This is accomplished by analyzing both forward and backward-edge jumps during an LLVM compiler pass. Binsweep$_{\text{CFI}}$ works by rewriting all return instructions as forward jumps to a register holding the return address for a function call. This is convenient because the compiler transformation reduces the problem of enforcing both backward and forward-edge CFI to simply enforcing forward-edge CFI. That is, every return instruction is rewritten as an indirect branch to the contents of the stored instruction pointer located on the stack. In our implementation, return instructions are still identifiable since all return instructions jump directly to a special "return thunk" which performs the indirect branch (see Section 4).

Next, Binsweep$_{\text{CFI}}$ rewrites all indirect jumps (i.e., forward edges) through registers to implement a check for the unique instruction that represents the end of a branch (see Figure 7). In this figure, a simple indirect branch (i.e., `jmp %rax`) is transformed into the sequence of instructions given in the figure. Due to how LLVM schedules individual instructions during code generation, potentially malicious instructions may become interleaved within this CFI sequence. In this example, such an instruction provided by the adversary in the untrusted buffer is highlighted in red. Binsweep$_{\text{CFI}}$'s principal goal is to generate CFI patterns without checking for these instructions. However, the default Binsweep policy, ρ_{CFI}, checks all indirect branches in a code buffer to confirm the CFI sequence shown in Figure 7) is followed correctly and reject any buffers where malicious instructions are interleaved in a CFI sequence (see Section 3.3).

The generated CFI sequence consists of the following operations. First, an instruction examines the memory contents of the branch target (i.e., the target register). To confirm that the branch target refers to the endbranch instruction, the CFI sequence performs a 32-bit dereference from the target register and stores the contents of memory within a 32-bit compare register. (The choice of these registers is up to the compiler or a developer in the case a given program contains handwritten assembly). To check that the compare register is equal to the end branch instruction, the sequence performs an unsigned 32-bit addition on the compare register with the additive inverse of

$$\frac{pc,Q,\Lambda,\mathcal{P} \vdash \rho_1 \iota(pc) \wedge \rho_2 \iota(pc) \wedge \ldots \wedge \rho_n \iota(pc)}{pc,Q,\Lambda,\mathcal{P} \vdash \langle succ\ pc, Q, \{pc\} \cup \Lambda, \mathcal{P} \rangle}\ \textsc{Allowed Op} \qquad \frac{pc,Q,\Lambda,\mathcal{P} \vdash \neg\rho_1 \iota(pc) \vee \neg\rho_2 \iota(pc) \vee \ldots \vee \neg\rho_n \iota(pc)}{\bot}\ \textsc{Illegal Op}$$

$$\frac{pc,Q,\Lambda,\mathcal{P} \vdash \textbf{jmp}\ \ell \quad \Lambda\ell = false}{pc,Q,\Lambda,\mathcal{P} \vdash \langle \ell, Q, \{pc\} \cup \Lambda, \mathcal{P} \rangle}\ \textsc{Jump} \qquad \frac{pc,Q,\Lambda,\mathcal{P} \vdash \textbf{jz}\ \ell \quad \Lambda\ell = false}{pc,Q,\Lambda,\mathcal{P} \vdash \langle \ell, Q, \{pc\} \cup \Lambda, \mathcal{P} \rangle \Downarrow b_t \wedge \langle succ\ pc, Q, \{pc\} \cup \Lambda, \mathcal{P} \rangle \Downarrow b_f}\ \textsc{Conditional Jump}$$

$$\frac{pc,Q,\Lambda,\mathcal{P} \vdash \Lambda\ pc = true}{true}\ \textsc{Seen} \qquad \frac{pc,Q,\Lambda,\mathcal{P} \vdash \iota(pc) = \text{out of range}, \{\tau_1, \tau_2, \ldots, \tau_n\}}{pc,Q,\Lambda,\mathcal{P} \vdash \langle \tau_1, \{\tau_2, \ldots, \tau_n\}, \Lambda, \mathcal{P} \rangle}\ \textsc{Out of Range}$$

$$\frac{pc,Q,\Lambda,\mathcal{P} \vdash \textbf{ret}, \{\tau_1, \tau_2, \ldots, \tau_n\}}{pc,Q,\Lambda,\mathcal{P} \vdash \langle \tau_1, \{\tau_2, \ldots, \tau_n\}, \Lambda, \mathcal{P} \rangle}\ \textsc{Return} \qquad \frac{pc,Q,\Lambda,\mathcal{P} \vdash \textbf{ret}, \{\}}{true}\ \textsc{Last Return}$$

Figure 6: Operational semantics of BINSWEEP's recursive descent sweeping procedure on instructions in a high level ISA.

```
1  mov          r_target ,r
2  mov_32       r_compare ,(r_target )
3  add_32       r_compare ,endbranch^-1
4  add          r_compare ,0x1000
5  cmp_32       r_compare ,0
6  jz           succeed
7  halt
8  succeed:
9  jmp                    r
```

Figure 7: An indirect branch transformed by BINSWEEP$_{CFI}$ (Intel syntax). This code is vulnerable to tampering due to an interleaved instruction highlighted in red (line 4). This instruction alters $r_{compare}$ to make an invalid branch target pass the CFI check. However, the ρ_{CFI} policy, enabled by default, rejects this snippet and any pattern that tampers with r_{target} or $r_{compare}$.

the end branch instruction (i.e., **endbranch**+**endbranch**$^{-1}$ = 0 in 32-bit unsigned arithmetic).

This ensures that adding the CFI check throughout a compiled program does not inadvertently introduce valid branch targets at every CFI pattern (i.e., comparing directly to *branchend* would make the address of the operand in memory a valid branch target). If the sum of the branch target and **endbranch**$^{-1}$ equal zero, a conditional jump will set the program counter to the original indirect branch and the program will continue now that the branch target has been verified. If the sum is non-zero, then the CFI sequence falls through to a failure handler that halts the process before an adversary can take arbitrary control of the program counter. Note that an adversary can freely jump to any location directly in their untrusted code buffer's location in process memory. However, this CFI pattern prevents the adversary from jumping to arbitrary locations within their sandbox.

3.3 BINSWEEP CFI Policy

To verify the validity of each CFI sequence within an untrusted code buffer, we implement the ρ_{CFI}, which is enabled by default, in the following way. During policy checking, if we encounter an indirect branch, this triggers a backward linear scan for every instruction shown in black given in Figure 7. That is, assume we detect the indirect branch to a register r on line 9. If this is a valid indirect branch, we must see a conditional jump to a failure routine before the indirect jump (line 6). This must be preceded by a 32-bit unsigned comparison instruction that compares a register to the immediate value of 0 (line 5). Once found, the register encountered in this comparison is referred to as the compare register $r_{compare}$.

Next, a 32-bit add instruction on $r_{compare}$ must occur before the compare, and the additive inverse of **endbranch**, **endbranch**$^{-1}$, must be an instruction operand in this instruction (line 3). This must also be preceded by a 32-bit move instruction from a register into $r_{compare}$ (line 2). The source register of this move instruction is referred to as the target register r_{target}.

Finally, a 64-bit move instruction must occur from the destination of the originally detected indirect jump and r_{target} (line 1). If any of these conditions are not met, the state machine that implements this policy cannot advance and reach an accepting state. Thus, ρ_{CFI} will reject the detected indirect jump after encountering either an **endbranch** or after scanning a fixed amount of instructions backward from the instruction, whichever occurs first.

In practice, LLVM will often interleave other instructions from the untrusted buffer located around the indirect branch. Instead of imposing restrictions on the LLVM toolchain and its use for generating code buffers for BINSWEEP, we simply permit interleaved instructions to be placed within the CFI pattern. This requires that ρ_{CFI} must ensure that each interleaved instruction preserves the contents of r_{target} and $r_{compare}$, as their modification can easily allow an adversary to jump to invalid target branches and evade BINSWEEP. For example, in line 4 in Figure 7, an adversary could jump to an arbitrary memory location by crafting *reg* to refer to a memory location that is almost like **endbranch**.

That is, suppose r refers to a memory location that contains the bytes `F3 0f 0e fa`. This is not equal to **endbranch**, and hence BINSWEEP will not enqueue the address within Q during recursive descent sweeping. If this is located within a data section, then it is unlikely to be part of any other basic block scanned by BINSWEEP. Therefore, an adversary could store a shell code that breaks out of a

sandbox or issues system calls following this byte pattern, and then do an indirect jump to the location of the pattern in memory. When the program counter reaches line 5 in the CFI pattern, observe that $r_{compare} + 0x1000 + \textbf{endbranch}^{-1} = 0$. This causes the compare to succeed and jump to r, bypass the CFI check, and execute an unswept payload. However, BINSWEEP always rejects any code buffer that contains this snippet, since the interleaved malicious instruction alters $r_{compare}$ and ρ_{CFI} always rejects such instructions while scanning CFI patterns.

A simple approach to rejecting invalid interleaved instructions may rely on a limited allowlist of instructions and check during scanning that the target and compare registers never appear as operands within an interleaved instruction. Though this can prevent the simplest attacks, the semantics of x86 instructions are complex, with different prefixes used to modify instruction behavior and switch into different modes (e.g., VEX for vector extensions).

To prevent the use of exotic extensions from corrupting the CFI pattern, we use our disassembler's support for enumerating registers modified by each decoded instruction (see Section 4). This general approach allows ρ_{CFI} to examine any valid instruction and confirm, up to the correctness of our chosen disassembler, that an interleaved instruction preserves the contents of the target and compare registers, and hence preserve the semantics of the CFI sequence. In our implementation, we use a combination of allowlists and instruction effect tracking to reject dangerous interleaved instructions.

Effect tracking individual instructions allows BINSWEEP to determine whether a given instruction can threaten the integrity of BINSWEEP$_{\text{CFI}}$'s emitted CFI check. For example, recall the offending instruction in Figure 7 (line 4) which increments the contents of the branch target stored in $r_{compare}$ to make an invalid branch target appear valid. For every interleaved instruction, BINSWEEP computes a set of modified registers $\overline{r_{modified}}$. If either r_{target} or $r_{compare}$ is in $\overline{r_{modified}}$, then the CFI check fails and a policy violation is raised, as in Figure 7. Otherwise, the instruction is permitted, and the state machine continues on towards the final accepting state that completes recognizing the full CFI pattern.

4 Implementation

In this section, we describe our prototype implementation of BINSWEEP. This prototype consists of 9,971 lines of Java code which is compiled to an executable program with GraalVM Native Image [37, 40]. This prototype implements the recursive descent sweeping algorithm described in Section 3 to scan untrusted instruction streams for policy violations. The CFI component, BINSWEEP$_{\text{CFI}}$, is implemented as an LLVM compiler pass in 237 lines of C++ code, and verified as a BINSWEEP policy.

4.1 Disassembly

We utilize Intel XED to decode individual x86 instructions. XED is a robust disassembler used by the popular Intel PIN dynamic binary instrumentation framework [21]. In addition, XED has been used by virus scanners, which often analyze untrusted code. Since XED is a C library, we make use of Native Image's foreign function interface (FFI) features to easily call out to C code from Java. This provides a convenient, and memory safe way to analyze the contents of entire executable files.

Note that, during disassembly, BINSWEEP need not explicitly construct CFGs. Instead, BINSWEEP walks the paths of individual CFGs given within the code buffer, and avoids the overhead of maintaining a large collection of CFGs on which to perform analysis. This keeps the resource overhead of BINSWEEP low, both in terms of memory and CPU time, when compared to other, more full featured binary analysis frameworks, which tackle the harder problem of lifting CFGs while making fewer assumptions on the underlying executable. In our setting, the assumption that an executable conforms to CFI simplifies BINSWEEP's main task of vetting code buffers for individual policies. Towards this end, the set of visited instructions Λ can be maintained using a simple Java `Set` along with the queue of entrypoints Q.

4.2 Software Control-Flow Integrity with LLVM

The software CFI approach outlined in Section 3 utilizes the `ENDBR64` instruction as CFI's end branch instruction on x86. This instruction has the benefit that the instruction sequence that implements it was previously used as a wide `NOP` pattern on earlier CPUs. This implies that BINSWEEP$_{\text{CFI}}$ can support existing CPUs that may lack support for Intel's indirect branch tracking (IBT) which is a part of Intel's Control-Flow Enforcement Technology (CET). Furthermore, on operating environments where CET support may be incomplete, our software CFI approach can provide an easy to use solution for CFI in software.

Rewriting every backward edge into a forward edge is accomplished using an LLVM compiler pass. Furthermore, enumerating all forward edges after the transformation enables our compiler pass to harden each forward edge with a CFI check. Our compiler pass considers forward edges represented as either calls or indirect jumps (i.e., a jump through a register value). This CFI check first computes the target of the branch. The compiler pass then emits code that loads the contents of memory at the branch target into a 32-bit register. This is necessary to ensure the the sum of the contents of the branch target and the additive inverse of `ENDBR64` equal 0 (i.e., overflow the 32-bit register to 0). This allows us to check for the end branch instruction throughout the executable without inadvertently introducing valid entrypoints at every CFI check. If the CFI check passes, the emitted CFI code then jumps to the branch target as usual. If the check fails, then the program halts, and execution stops.

4.3 BINSWEEP CFI Policy

Recall that the CFI policy is always enabled in BINSWEEP. This ensures that untrusted code buffers adhere to our CFI scheme, and provides confidence in BINSWEEP's verification results. Each forward edge, be it an indirect jump or call instruction, is verified using an automaton based BINSWEEP policy. This policy uses the ability to traverse instruction predecessors to walk backward from each indirect jump to verify the completeness of the CFI pattern.

The initial state begins by examining the jump to some location, possibly a register or absolute address in the case of a call. To reach the automaton's accepting state, the policy must work backwards in the current instruction sequence from the program counter *pc* to confirm that the jump is made after a conditional branch, that the conditional branch is taken if and only if the previously defined CFI check succeeds, and the end branch instruction is detected from the register referred to in the checked branch.

Observe that this procedure can be accomplished by examining predecessors in the visited instruction stream until either the accepting state is reached, or the end branch instruction that started the instruction sequence is reached. If the latter occurs, then the automaton ends in a rejecting state, and the CFI policy rejects the sequence as invalid. As discussed in Section 3, LLVM may interleave instructions from the program within the CFI check for an indirect branch. Hence, hardening approaches were required to ensure that interleaved code cannot interfere with the fidelity of the CFI check (i.e., clobbering the contents of the branch target stored within a register). This can be easily accomplished by using XED's introspection features on individual instructions. For each visited instruction, we reject any instruction whose operands are modified and those operands are either r_{target} or $r_{compare}$. In addition, instruction prefixes must be heavily vetted to prevent modifying these sensitive registers, or altering the flow of execution to an unverified instruction stream (e.g., jumping into the middle of the next instruction). XED allows querying the categories of prefixes present on each instruction, and this allows us to remove common prefixes to disable attacks by modifying regular instructions, while also preserving the ability to execute extensions, such as REX, and VEX.

4.4 Hardening Policies for x86

We have found that restricting untrusted code buffers for real ISAs as extensive as x86 presents a challenge for authoring policies. Nonetheless, the flexibility provided by Intel XED in dissecting individual instructions (e.g., rejecting instructions based on explicit and implicit operands), allows a policy to detect subtle unwanted effects in individual instructions and reject them as a result.

5 Evaluation

In this section, we evaluate BINSWEEP to answer the following research questions.

- Can BINSWEEP detect malicious, and concealed, code paths embedded in executables?
- Is BINSWEEP's performance overhead, both while verifying binaries and enforcing software CFI at runtime, acceptable for real world applications?

To answer these questions, we run BINSWEEP on code buffers that conceal malicious instruction sequences within the buffer's CFG. That is, a naive linear scan of the buffer's disassembly would mistakenly flag these buffers as safe (i.e., false negatives). Furthermore, we evaluate BINSWEEP on both benchmarks (SPEC CPU 2017) and real world applications. Our results show that BINSWEEP can both verify the instructions within complex real world binarys' CFGs and protect the integrity of their execution with manageable performance overhead.

5.1 Experimental Setup

All of our experiments were run on a Red Hat Derived Linux distribution on a Intel Core i9-13900K machine with 64GB of RAM.

5.2 Attack Case Studies

In this section, we present case studies where BINSWEEP detects malicious instructions, including attempts to break out of an intra-process sandbox, issue system calls, set up a return oriented programming (ROP) chain, or bypass CFI. In addition to these attack case studies, we fuzz tested the XED instruction decoder used by BINSWEEP for over three CPU months with AFL++ [14], and detected no crashes.

Benchmark	BINSWEEP$_{CFI}$ Overhead	Protected Edges
500.perlbench	+0.02 %	4,532
502.gcc	+1.96 %	24,301
505.mcf	+1.46 %	144
520.omnetpp	+6.25 %	22,905
523.xalancbmk	+0.00 %	25,167
531.deepsjeng	+1.15 %	240
541.leela	+0.00 %	1,693
557.xz	+4.21 %	877

Table 1: SPECint benchmarks from SPEC CPU2017 with average observed overhead and number of edges protected by BINSWEEP$_{CFI}$.

Memory Protection Key Escape. Within an intra-process application sandbox hardened with Intel memory protection keys (MPK), untrusted guest applications must not be able to issue the sensitive write protection key register for user pages `wrpkru` instruction. This user-mode instruction effectively allows an adversary to drop all restrictions enforced by MPK and break out of the sandbox. BINSWEEP can detect attempts to execute `wrpkru`, even while it is hidden within an untrusted code buffer's data (i.e., just after a function or stored within the immediate value of an instruction).

In our evaluation, we defined a code buffer that concealed a `wrpkru` instruction into an immediate value for a `mov` instruction. Later on in the instruction sequence, a relative jump places the program counter in the middle of the `mov` instruction, and breaks out of the sandbox by writing all zeros to the PKRU (i.e., escalates read and write access to all MPK domains). BINSWEEP rejected this code buffer, by recursively sweeping the instructions reachable from the jump to the middle of the instruction. Note that this approach removes the possibility for false positives, since the instruction embedded in the intermediate value would be deemed unreachable if the relative jump is absent. In contrast to more naive approaches like linear instruction scanning, recursive descent sweeping can explore all program behaviors reachable from a code buffer's CFG.

Concealed Syscalls. The situation above can also be extended to reject syscalls, to extend a security policy for an intra-process sandbox. In this setting, allowing an adversary to interact directly with the kernel can have undesired consequences, including interfering with other hosts on the sandbox via confused deputy attacks [12]. In our evaluation, we demonstrated that attempts to reach a `syscall` instruction embedded directly within the code buffer, was always detected by BINSWEEP. This includes scenarios where the code buffer jumps into the middle of instructions, or directly to data located outside the function.

Return Oriented Programming (ROP) Chain. A common attack vector to hijack a process is to place a return oriented programming (ROP) chain somewhere in attacker controlled memory and perform

Program	Binary Size (MB)	Binsweep Verification Time (s)	CFG Recovery Time in angr (s)	Basic Blocks	Protected Edges
NGINX	1.3	0.087	29	25,198	3,199
CPython	16.0	0.258	116	106,095	9,261
Micronaut	77.0	4.625	–	578,047	107,787
GraalPy	362.0	18.131	–	2,474,585	384,159

Table 2: Statistics for verifying binaries for widely used programs with Binsweep, the amount of time needed to verify the entire binary (including building the CFG) with Binsweep running substantially faster compared to only recovering CFGs with angr, a state of the art binary analysis framework. Also shown are the no. of basic blocks in the binary's CFGs and the number of forward edges protected by Binsweep$_{CFI}$. For both the Micronaut `helloworld` and GraalPy interpreter, angr failed due to memory exhaustion after 70 and 100 minutes, respectively.

a stack pivot to change the stack pointer to attacker controlled memory. Since Binsweep$_{CFI}$ enforces backward-edge CFI by rewriting all return instructions as jumps to the stored instruction pointer, any attempt to `return` to a stack pivot will be rejected by the runtime. In our evaluation, we defined a ROP policy ρ_{ROP} that rejects stack pivots in the code buffer (i.e., `xchg rsp, rax`). Furthermore, the always enabled CFI policy rejected any indirect jumps that lack the CFI code pattern described in Section 4, which would be used to jump to the pivoted stack pointer and start the ROP chain.

CFI Bypass. A significant effort went into investigating approaches to defeat the CFI pattern introduced by Binsweep$_{CFI}$ since this is the simplest approach to gain arbitrary code execution within a process protected by Binsweep. The simplest approach would be to use prefixes on jump instructions to land in the middle of swept instructions and thus start executing an unverified instruction stream. Other attacks can take advantage of other general purpose prefixes (i.e., `REP` and `REPZ`) to modify the sensitive r_{target} and $r_{compare}$ registers when assigned registers modified by certain instructions.

Furthermore, different extensions to the x86 instruction set pose problems, such as the vector extensions (VEX) that enable working on wide operand types. To mitigate CFI escape from either direct or indirect modifications to r_{target} and $r_{compare}$ we implemented a combination of a restricted allowlist for interleaved instructions, rejected common instruction prefixes (aside from those that enable useful modes), and implemented operand tracking using XED's available APIs. This allowed us to build a corpus of test programs that defeat a simpler implementation of ρ_{CFI} (i.e., based solely on instruction allowlists and limited operand checking) and gain confidence in our current implementation.

5.3 Performance Case Studies

In this section, we first evaluate the performance overhead of enforcing CFI using Binsweep$_{CFI}$, the compiler pass that rewrites programs to use our software CFI implementation. We present the performance overhead of Binsweep$_{CFI}$ on the SPECint benchmarks from SPEC CPU 2017, and examine several case studies in real world applications vulnerable to exploits from remote adversaries (i.e., web servers, language interpreters, and databases). We then compare the performance of Binsweep with angr, a state of the art binary analysis framework.

SPEC CPU 2017. To establish a baseline performance overhead we can expect for running programs with Binsweep's software CFI, we run the SPECint benchmarks from the SPEC CPU 2017 suite. These applications implement a broad range of functionality and provide insight into Binsweep's performance across domains, from parsing compressed files, video, XML to scientific computing applications. Focusing on the SPECint benchmarks allows us to test Binsweep's performance across a variety of domains, as opposed to programs in SPECfp that primarily pressure a CPU's floating point arithmetic capabilities. Furthermore, the Binsweep software CFI toolchain does not support Fortran, since such programs are rarely accessible to adversaries.

Table 1 summarizes the performance overhead of using Binsweep's software CFI on the SPECint benchmarks. Overall, Binsweep incurs 6.25% performance overhead in the worst case. Note that the performance overhead of Binsweep can vary for both small and large programs. That is, an application with many protected edges will not necessarily incur the worst case overhead. Instead, an application that exercises more of its branches will incur more overhead. Throughout the benchmarks, the performance overhead stays low, and only rises to a manageable overhead for a few benchmarks.

Real World Programs. Since the SPEC CPU benchmarks typically contain programs that process trusted inputs (e.g., simulating Shor's algorithm), we measure the performance overhead of Binsweep on applications most vulnerable to control-flow hijacking exploits from remote attackers. Furthermore, we measure Binsweep's runtime while verifying binaries for real world programs. Table 2 summarizes the time required for verifying real world software with Binsweep, along with statistics on the programs under analysis, such as the number of basic blocks and edges protected by Binsweep$_{CFI}$.

To showcase Binsweep's runtime scalability on binaries with increasing size, we included a Micronaut `helloworld`[1] application built with Oracle GraalVM Native Image [37, 40], and GraalPy, the GraalVM implementation of Python[2].

Here, "protected edges" denote the number of indirect jumps restricted by Binsweep$_{CFI}$, whereas the number of verified basic blocks include the basic blocks to which indirect branches cannot jump, due to the restrictions of Binsweep$_{CFI}$. Overall, Binsweep processes real world binaries very efficiently, especially when compared to full-fledged static analysis tools that require constructing

[1] https://guides.micronaut.io/latest/creating-your-first-micronaut-app.html
[2] https://www.graalvm.org/python/

knowledge bases in memory or disk for a binary before performing analysis. In contrast, BINSWEEP iteratively scans all basic blocks once while applying all relevant policies to each encountered instruction. This makes BINSWEEP especially suitable for verifying untrusted code buffers as a part of vetting code for inclusion in a production environment. In addition, our results demonstrate that BINSWEEP can efficiently analyze large, complex executables that can be hundreds of megabytes in size.

Comparison to angr. The angr framework is a state of the art binary analysis framework [30] which supports a myriad of security related binary analysis workflows, including recovering CFGs, decompilation, and symbolic execution. Furthermore, angr placed in the DARPA Cyber Grand Challenge (CGC) program [2] and is actively used in the DARPA AI Cyber Challenge (AIxCC) program [3]. In contrast, BINSWEEP is designed to simply discover and exhaustively search CFGs in untrusted code buffers to prove the absence of malicious instruction sequences and conformance to our CFI technique. As a result, BINSWEEP performs verification on CFGs much faster than angr by default, as shown by our performance comparison visualized in Table 2. This experiment showed that BINSWEEP often runs orders of magnitude faster than angr. In addition, angr was unable to recover CFGs for the large executables included in our evaluation (i.e, a Micronaut application and the GraalPy interpreter) within an hour. Contrast this with BINSWEEP analyzing these executables in seconds.

In this experiment, we used angr version 9.2.113. While running angr, we simply recovered CFGs of our evaluation's real world programs by using default parameters for angr, except we disabled the automatic discovery of library dependencies. To recover CFGs, we used angr's `CFGFast` routine. This mode uses static analysis to recover a CFG, in contrast to more advanced, and slower modes, that rely on symbolic execution to recover the destination of indirect branches. Furthermore, we excluded the amount of time needed to create the angr project for each executable, whereas BINSWEEP's verification time includes loading and verifying the executable. All analysis was restricted to the main executable. Likewise, the policies applied by BINSWEEP are not specialized for the programs under analysis, they are the default policies automatically applied to every program.

Our results indicate that BINSWEEP can provide a performant static binary analysis for vetting the security of untrusted code buffers in production sandbox systems, as opposed to supporting general purpose binary analysis tasks. In practice, BINSWEEP, like angr, easily verifies both executables and all supporting libraries, including the C Standard Library. In this experiment, BINSWEEP also scanned the recovered CFGs for illegal instructions and for valid CFI sequences. We expect that scanning recovered CFGs in angr can be done efficiently with a more specialized configuration. We only recover CFGs in this experiment to highlight BINSWEEP's efficiency for recovering CFGs and enforcing policies simultaneously. The large executables in our evaluation emphasize BINSWEEP's efficiency. As the size of the executable reaches hundreds of megabytes, the speed of performing analysis stays within tens of seconds. In contrast, angr configured with default parameters takes more than an hour to recover these executable's CFGs before exiting with an out of memory error. We emphasize that this is not a deficiency of angr, but rather the benefit of enumerating, as opposed to explicitly constructing, CFGs to enforce security policies.

No. of Clients	Performance Overhead
512	+4.33 %
1,024	+0.15 %
2,048	+0.87 %
4,096	+1.58 %
8,192	+6.51 %

Table 3: Performance overhead (measured by decrease of request throughput) of running `NGINX` with BINSWEEP$_{CFI}$ and a baseline while handling 1,000,000 total requests generated from a varying number of clients. As the number of clients increases, the performance overhead stays manageable.

NGINX. The `NGINX` web server is one of the most widely used web servers that powers 34.2% of the Internet's publicly reachable hosts. In our evaluation, we built `NGINX` version 1.25.4 with BINSWEEP$_{CFI}$, and stressed both the CFI hardened and baseline server using the Apache Bench `ab` tool. Table 3 summarizes the performance overhead, in terms of decreased request throughput, when using `NGINX` protected by BINSWEEP$_{CFI}$. In this experiment, eight `NGINX` worker processes, which run on 16 available CPU cores, accept requests from an increasing number of clients. As the number of clients increases, the performance overhead stays manageable at 6.55%. These results show that BINSWEEP$_{CFI}$ can protect web servers during times of high load with a manageable amount of performance overhead.

Microservice. Micronaut is a Java framework for developing cloud microservices. The GraalVM Native Image compiler can emit efficient native executables from Micronaut services compiled to Java. During our evaluation, we used Oracle GraalVM Native Image version 23 to compile a native executable for a simple `helloworld` Micronaut service. We verify this Micronaut executable with BINSWEEP, and demonstrate how our approach can scale to large complex binaries. Note that, the executables emitted by Native Image represent Java programs, and all their dependencies, in the target platform's instruction set architecture (ISA). In addition to the microservice, the entire Micronaut framework used by the microservice, all Java library dependencies, and a runtime to support components like a garbage collected heap all compiled into the microservice's executable. This demonstrates BINSWEEP's scalability and robustness for verifying real world microservices and highly complex binaries.

Python Interpreters. The CPython interpreter is the most used implementation of the popular Python programming language. Web applications that regularly interact with untrusted remote clients are often implemented in Python. This makes the CPython interpreter an attractive attack surface shared amongst all python web services. For this reason, securing these vulnerable web applications' language runtime a vital and ongoing security concern. In our evaluation, we used CPython version 3.12.3. When running CPython protected with BINSWEEP$_{CFI}$ on standard Python benchmarks [1], we observed a performance overhead of 0.87%. To further demonstrate BINSWEEP's ability to verify real world interpreters, we verified a native executable for GraalPy, an implementation of the Python programming language in GraalVM Truffle [38, 39], a framework for implementing

efficient interpreted languages in the GraalVM [41]. For our evaluation, we built GraalPy version 23 using the same GraalVM Native Image builder as Micronaut. We observed that Binsweep was able to efficiently verify the resulting executable, which is hundreds of megabytes in size, in less than 20 seconds. This quick verification time enables operators to admit untrusted Python interpreters into an intra-process sandbox and drastically limit the amount of trusted code shared between tenants (i.e., the blast radius of a compromised interpreter is restricted to a single tenant).

6 Related Work

Binsweep relates to prior work in two main categories, control-flow integrity (CFI) and static binary analysis.

6.1 Control-Flow Integrity

Control-flow integrity (CFI) restricts the paths a program may execute to the set known at compile time [5]. Without a loss of generality, the notion of "compile time" could be extended to code dynamically executed on the fly, as in a just-in-time (JIT) compiler [23]. CFI is often implemented by a toolchain that inserts checks within machine code that control-flow transitions (i.e., indirect jumps and returns) are performed properly, and can be applied to both user space programs [32] and kernels [42]. This complicates an adversary's task of hijacking control of a program, either by overwriting the stored instruction pointer on the stack, or hijacking C++ vtables [43] to trick programs into jumping to attacker controlled pointers.

Though software CFI implementations have traditionally incurred high performance overhead, hardware manufacturers have recently provided CFI primitives within hardware [19]. This allows programs to verify backward-edge transitions by either checking the integrity of stored instruction pointers, or relying on a shadow stack to detect corrupted return values. Furthermore, techniques like indirect branch tracking (IBT) allow the hardware to only perform indirect jumps to locations with a designated end branch instruction (i.e., `ENDBR64`). Forward edges can also be verified by performing integrity checks on register values before an indirect jump. While these approaches have limitations (i.e., all branch targets are grouped within the same equivalence class), their implementation in hardware implies that CFI can be efficiently enforced with minimal changes made to the underlying program.

In this work, we adopt the approach taken by IBT by verifying branch targets in software before performing indirect jumps. Furthermore, care is taken to avoid inadvertently introducing unintended basic blocks within our CFI checks (i.e., excluding the `ENDBR64` constant from emitted CFI code). This allows us to statically confirm untrusted programs conform to our CFI technique using a Binsweep policy.

6.2 Static Binary Analysis

Static binary analysis can trace its roots back to decompilation frameworks that recovered control-flow graphs (CFGs) by recursively disassembling the contents of executable files[11]. Despite the intractability of the problem in general, binary analysis has flourished into a diverse ecosystem of tools for reverse engineering binaries [25] in addition an active research topic to address limitations found in practice, such as maintaining fidelity in decompilation results [13], and evaluating important metrics to measure decompilation quality (e.g., difference from an executable's source code) [8]. These tools enable performing static analysis passes over CFGs recovered from binary executables [9, 30], recover source code from the CFG in combination with machine learning techniques [24], and deploy security hardening techniques via binary rewriting [36, 44].

In this work, we restrict the problem of statically analyzing untrusted code to executables that conform to a CFI scheme. By confirming that CFI patterns are always present for every indirect jump, we can restrict the behavior of an executable to those instructions reachable via recursive descent. In contrast to more general purpose static binary analysis tools, which are designed to support a variety of analyses over arbitrary executables, Binsweep verifies both that malicious instructions are not present, and actively checks for conforming to our CFI technique. Any deviation from the chosen CFI technique causes Binsweep to reject the binary. Otherwise, Binsweep statically traverses all basic blocks within the CFGs contained within the binary, and rejects all instructions forbidden by the chosen security policies. In our evaluation, we found that our approach of scanning CFGs without explicitly constructing them is able to outperform angr in recovering CFGs. This result is somewhat expected: Binsweep is a highly optimized analysis intended to ensure the security of untrusted code buffers, not a general purpose binary analysis framework like angr.

7 Conclusion

In this work, we introduced Binsweep, a lightweight static binary analysis tool for verifying untrusted instruction streams with recursive descent sweeping, described by formal semantics. We described $\text{Binsweep}_{\text{CFI}}$, our software based CFI approach that allows Binsweep to restrict analysis to CFGs reachable from special end branch instructions. We evaluated Binsweep over SPEC CPU 2017 benchmarks, along with widely used programs vulnerable to exploitation, including the popular `NGINX` web server, a Micronaut microservice, and Python interpreters. Our results show that Binsweep can efficiently verify production executables, hundreds of megabytes in size, much faster than angr, a state of the art binary analysis tool, and that $\text{Binsweep}_{\text{CFI}}$ has manageable performance overhead (6.25% in the worst case).

References

[1] [n. d.]. Python Interpreter Benchmarks. https://pybenchmarks.com.
[2] 2016. Cyber Grand Challenge (CGC). https://www.darpa.mil/program/cyber-grand-challenge.
[3] 2024. AI Cyber Challenge (AIxCC). https://www.darpa.mil/program/ai-cyber-challenge.
[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *ACM Conference on Computer and Communications Security*.
[5] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and Systems Security* (2009).
[6] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *USENIX Security Symposium*.
[7] Gogul Balakrishnan and Thomas Reps. 2010. WYSINWYX: What You See is Not What You EXecute. *ACM Transactions on Programming Languages and Systems* (2010).
[8] Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. 2024. Ahoy SAILR! There is No Need to DREAM of C: A Compiler-Aware Structuring Algorithm for Binary Decompilation. In *USENIX Security Symposium*.
[9] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *International Conference on Computer-Aided*

Verification.

[10] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *Comput. Surveys* (2017).

[11] Cristina Cifuentes and K John Gough. 1995. Decompilation of binary programs. *Software: Practice and Experience* (1995).

[12] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *USENIX Security Symposium*.

[13] Luke Dramko, Jeremy Lacomis, Edward J Schwartz, Bogdan Vasilescu, and Claire Le Goues. 2024. A taxonomy of c decompiler fidelity issues. In *USENIX Security Symposium*.

[14] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *USENIX Workshop on Offensive Technologies*.

[15] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *USENIX Annual Technical Conference*.

[16] Intel. [n. d.]. A Technical Look at Intel's Control-flow Enforcement Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html.

[17] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *USENIX Annual Technical Conference*.

[18] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static Disassembly of Obfuscated Binaries. In *USENIX Security Symposium*.

[19] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. 2019. PAC it up: Towards pointer integrity using ARM pointer authentication. In *USENIX Security Symposium*.

[20] Arm Ltd. [n. d.]. ARM Permission indirection and permission overlay extensions. https://developer.arm.com/documentation/102376/0200/Permission-indirection-and-permission-overlay-extensions.

[21] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices* (2005).

[22] Alan Mujumdar. [n. d.]. Armv8.1-M Pointer Authentication and Branch Target Identification Extension. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension.

[23] Ben Niu and Gang Tan. 2014. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *ACM Conference on Computer and Communications Security*.

[24] Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, et al. 2024. "len or index or count, anything but v1": Predicting variable names in decompilation output with transfer learning. In *IEEE Symposium on Security and Privacy*.

[25] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *IEEE Symposium on Security and Privacy*.

[26] Soyeon Park, Sangho Lee, and Taesoo Kim. 2023. Memory Protection Keys: Facts, Key Extension Perspectives, and Discussions. *IEEE Security and Privacy* (2023).

[27] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software abstraction for Intel memory protection keys. In *USENIX Annual Technical Conference*.

[28] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *USENIX Security Symposium*.

[29] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *International Workshop on Hardware and Architectural Support for Security and Privacy*.

[30] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy*.

[31] Chilik Tamir. [n. d.]. Dr. Jekyll and Mr. "Hide" – How Covert Malware Made it into Apple's App Store. https://www.zimperium.com/blog/dr-jekyll-and-mr-hide-how-covert-malware-made-it-into-apples-app-store/.

[32] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow integrity in GCC and LLVM. In *USENIX Security Symposium*.

[33] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security Symposium*.

[34] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. 2013. Jekyll on iOS: When Benign Apps Become Evil. In *USENIX Security Symposium*.

[35] Martin Weidmann. [n. d.]. Arm A-Profile Architecture Developments 2022. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-a-profile-architecture-2022.

[36] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. 2020. Egalito: Layout-agnostic binary recompilation. In *ACM Conference on Architectural Support for Progamming Languages and Operating Systems*.

[37] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize once, start fast: application initialization at build time. *ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity* (2019).

[38] Christian Wimmer and Thomas Würthinger. 2012. Truffle: a self-optimizing runtime system. In *ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*.

[39] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical partial evaluation for high-performance dynamic language runtimes. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*.

[40] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*.

[41] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. In *Symposium on Dynamic Languages*.

[42] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. 2022. In-KernelControl-Flow integrity on commodity OSes using ARM pointer authentication. In *USENIX Security Symposium*.

[43] Chao Zhang, Dawn Song, Scott A Carr, Mathias Payer, Tongxin Li, Yu Ding, and Chengyu Song. 2016. VTrust: Regaining Trust on Virtual Calls.. In *ISOC Network and Distributed System Security Symposium*.

[44] Haotian Zhang, Mengfei Ren, Yu Lei, and Jiang Ming. 2022. One size does not fit all: security hardening of mips embedded systems via static binary debloating for shared libraries. In *ACM Conference on Architectural Support for Progamming Languages and Operating Systems*.