Jinn: Hijacking Safe Programs with Trojans

Author Names Ommitted for Blind Review

Abstract

Untrusted hardware supply chains enable *malicious*, *powerful*, and *permanent* alterations to processors known as hardware trojans. Such hardware trojans can undermine any *software-enforced* security policies deployed on top of the hardware. Existing defenses target a select set of hardware components, specifically those that implement *hardware-enforced* security mechanisms such as cryptographic cores, user/kernel privilege isolation, and memory protections.

We observe that computing systems exercise general purpose processor logic to implement *software-enforced* security policies. This makes general purpose logic security critical since tampering with it could violate software-based security policies. Leveraging this insight, we develop a novel class of hardware trojans, which we dub *Jinn* trojans, that corrupt general-purpose hardware and can hide in many places within a processor to enable flexible and powerful high level attacks. Jinn trojans deactivate compiler-based security-enforcement mechanisms, making type-safe software vulnerable to memory-safety by compromising a *single* bit of architectural state. We show that Jinn trojans are effective even when planted in general purpose hardware, disjoint from any hardware-enforced security components. We show that protecting hardware-enforced security logic is insufficient to keep a system secure from hardware trojans.

1 Introduction

The increasing complexity of modern System-on-chips (SoCs) incentivises companies to outsource the designs of hardware design blocks [19, 33]. Similar to code reuse via software libraries, sourcing third-party intellectual property (3PIP) allows system integrators to benefit from highly optimized or specialized designs. The increased dependence on 3PIP exposes the hardware supply chain to the danger of malicious logic planted at design-time [32, 39, 42, 45, 83]. Rogue designers or malicious design houses can inject *design-time trojans* that permanently hide in an otherwise functional hardware component. Such trojans compromise the security of SoCs assembled by system integrators. Maliciously designed components masquerade as benign functional units, but alter critical signals under a stealthy set of run-time conditions that allows the trojan to persist undetected until deployment. When the run-time conditions *trigger* the trojan [63, 83], the malicious hardware modifies the processor's behavior to enable attacks against the system. Such trojans can leak cryptographic keys [29, 46, 49] or cause application code to execute within the processor's privileged mode [42].

Existing hardware trojans typically target processor mechanisms that implement *hardware-enforced* security policies (e.g., the user-kernel mode bit [79]). Such trojans limited to attacking hardware-enforced security logic, and invite increased scrutiny of the hardware components implementing those security features. Such hardware components may be subjected to verification or simply implemented in-house [77]. Further, prior work on defense mechanisms emphasizes the dependence of annotated hardware designs or security-critical hardware invariants that are generally decided at design-time, and generic to any particular software workloads [34, 70, 81, 84]. From a malicious hardware-designer's perspective, it seems challenging to place trojans at design-time in components used to implement hardware-enforced security policies that are highly analyzed and verified.

Many security policies are *enforced by software* that utilizes processor features *not typically associated with security*. For example, array bounds checks inserted by a compiler for a type-safe programming language protect programs from buffer overflow attacks [56, 68] using simple comparison and conditional branch instructions. We find that trojans that tamper with hardware used to implement software-enforced security policies can deliver payloads with comparable effects on system security while evading modern trojan detection schemes. Such trojans can indiscriminately hijack software running in multiple processor modes, affecting application code, operating system kernel code, and software running within a trusted execution environment (TEE).

Such trojans are not isolated to specific processor components typically associated with security enforcement; *any* part

of the processor that is used to implement instructions used by software to enforce security policies can be tampered to implement these trojans. Therefore, there is no singular self-contained component that requires increased scrutiny. In short, we observe that *general-purpose hardware used to implement instructions involved in compiler-injected safety-checks is* ***security-critical***.

Despite their advantages, designing a trojan that thwarts software-enforced security policies poses several challenges:

- Trojans must precisely distinguish between instructions from software that is enforcing security and the same instructions used elsewhere.
- Malicious hardware designers may be limited to hardware modifications within a *single* non-security-critical IP block.
- Trojans can't be tightly tailored to specific software, and must remain useful across patches to application software.

We prototype *Jinn* trojans, a new class of trojans that attacks in-flight memory safety checks injected by type-safe programming language compilers. Such compilers add auxiliary instructions into a program during compilation to maintain that program's safety properties (in this case, type-safety and memory-safety) during execution. Our Jinn trojans hide in general-purpose hardware IP blocks of a CPU core and tamper with the execution of safety checks, allowing attackers to exploit the *now vulnerable* memory accesses within type-safe software. In effect, Jinn trojans make type-safe software vulnerable to return-to-libc [69], return-oriented programming [56], and other memory safety attacks.

In summary, this paper makes the following contributions:

- We show that building a Jinn trojan is possible. Using gem5, we build end-to-end Jinn trojans that successfully launch code-reuse attacks on Rust programs to spawn a shell.
- We implement a Jinn trojan prototype in a large out-of-order RISC-V core and evaluate it's complexity and power consumption.
- We design two trigger mechanisms necessary to deliver Jinn payloads, and demonstrate trade-offs between versatility, precision, and attacker-effort.

The rest of the paper is organized as follows. Section 2 provides background material on hardware trojans and memory safety. Section 3 describes our threat model. Section 4 describes the Jinn attack methodology. Section 5 describes the design of Jinn hardware trojans, and Section 6 describes the attacker steps and corresponding malicious software that exercises the trojans. Section 7 describes our end-to-end attack implementations. Section 8 describes our RTL implementation. Section 9 presents the complexity of our gem5 and RTL trojan implementations. Section 10 discusses potential mitigations against Jinn trojans. Section 11 compares Jinn-style attacks to related work. Finally, Section 12 presents our conclusions.

2 Background

2.1 An Untrusted Hardware Supply Chain

The hardware supply chain broadly comprises three stages: design, fabrication, and deployment. The growing complexity of hardware designs and market deadline requirements encourage heavy re-use of hardware component designs, similar to the reuse of software libraries. Hardware engineers at the design stage use Hardware Description Languages (HDLs) like Chisel [18], Bluespec [52], Verilog and VHDL to specify the behavior of hardware components. Proprietary hardware designs are commonly implemented and integrated at this stage; they may be distributed as black-boxes, protecting third-party intellectual property (3PIP) to maintain profitability of highly specialized and optimized hardware designs [33].

The design stage presents an opportunity for powerful attackers such as nation-states or significant stakeholders in the hardware design space to inject malicious functionality into hardware designs [67]. Such malicious alterations are called *design-time* hardware trojans [32, 39, 42, 45, 83]. An attacker attempting to place design-time trojans will have access to the HDL-level implementation of the hardware design, allowing attackers to implement trojans that affect relatively high level behavior of the hardware. In comparison, *fabrication-time* trojans [21, 29, 46, 54, 79] are injected through alterations at the layout level of a hardware design and, consequently, rely on the sophistication of reverse-engineering techniques [55] to infer high level behavior.

2.1.1 Hardware Trojan Construction

The design of hardware trojans can be separated into two logical components: a *trigger* and *payload* [67].

The trigger defines the mechanism for activating the trojan's malicious behavior; until the trigger activates the trojan, the processor exhibits no malicious behavior. Effective trojans must successfully evade verification testing – when hardware blocks are tested for their functionality. Trojans that erroneously activate on benign workloads will be caught during verification testing and fail to reach deployment as the processor's behavior will deviate from expected benign behavior. Trojans must therefore be *stealthy*. They can evade detection by hiding in the enormous state space of modern integrated circuits (ICs) and snooping for an attacker-defined secret value or waiting on a set of conditions that have a low probability of occurring during benign workloads.

The trojan's payload defines the malicious function that is delivered when the trigger conditions are met – commonly overriding a benign signal. Payloads typically target security-critical control signals that lead to breaches in confidentiality (e.g. leaking cryptographic keys) [21, 46], integrity (e.g. flipping the user/kernel privilege bit) [42, 79], or availability (e.g. a kill-switch crashing the system) [13].

2.2 Memory Safety Guarantees

Programs written in safe programming languages like Java [31] and Rust [43] are guaranteed to be type-safe. Their type-safety ensures that accesses to memory via pointers always access the correct memory object (called the *referent* memory object [58]). The compiler for such languages may insert checks into the code to enforce type-safety at run-time; this is most notably done for array bounds checks [9, 44]. Because of these bounds-checks, type-safe programs are invulnerable to memory safety attacks such as code injection [53], return-to-libc [69], and return-oriented programming (ROP) [56] attacks which corrupt control data (such as function pointers and return addresses) to divert control flow to code of the attacker's choosing.

To maintain safe operation, compilers for type-safe languages rely on hardware to correctly implement the machine instructions (also referred to as machine code and native code) that implement the run-time checks. If the processor somehow modifies the behavior of these instructions, the run-time checks no longer work, and the program, even though its source code is type-safe, is now vulnerable to memory safety attacks.

3 Threat Model

We adopt a well-studied *design-time trojan* threat model [29, 32, 36, 63, 67, 71, 73, 82, 83]. The attacker's goal is to place a design-time trojan in a system-on-chip (SoC) that remains undetected through verification testing and is placed on a deployed machine. This machine deploys software written in type-safe programming languages such as Rust, Java, Go, C#, and Kotlin; the software is therefore protected by run-time checks inserted by the type-safe language compiler.

Attackers attempting to place design-time trojans have (or can reverse-engineer) high-level behavior of hardware components such as branch-predictors, decoders, computational-execution units, load-store queues, reorder buffers (ROBs), etc. Consistent with prior work [29, 32, 42, 50], we assume that attackers can modify the HDL-level, RTL-level, and netlist-level hardware designs. Consistent with modern IP reuse protection trends, 3rd-party IP (3PIP) hardware designs are shared as closed-source (or black-box) designs [26, 30, 77].

We assume that attackers forgo modifications to security-oriented hardware components (such as user/kernel privilege separation logic, memory-protection units (MPUs), trusted execution environment (TEE) logic, cryptographic cores, etc), and instead target general-purpose hardware (such as branch-predictors, ROBs, etc.). The attackers' goal when maliciously altering hardware designs is to inject hardware footholds which enable malicious software to compromise high-level security guarantees.

Attackers then attempt to interface with software running on the deployed system. An attacker can exercise the trojan in a wide variety of use cases including: victim software co-located with malicious software as separate guests on a single virtual machine server host; interfacing with a victim web server over the internet; or a disguised driver software running within the sandboxed constraints. We assume that all victim software is written in safe programming languages. The compilers of these safe programming languages inject the appropriate run-time checks.

Further, we adopt a threat model identical to those assumed in memory-safety research [11, 12, 53, 57, 62, 65, 80]. We assume that software exposes attacker-controllable variables that control the contents of run-time checked regions of memory. An attacker aims to undermine a compiler-injected run-time check and consequently exploit a memory-safety error to deviate the program's control flow. We assume that the attacker has some knowledge of the memory layout of the victim program and can further identify gadgets necessary to launch control-flow hijacking attacks. An attacker would use this information to prepare a payload to deliver to the deployed victim software.

4 Attacking Safe Programming Languages

Modern hardware trojans typically target processor components that implement hardware-enforced security policies (such as user/kernel privilege separation logic [42, 79], memory protection logic [50], cryptographic cores [46, 49, 54], and trusted execution logic [32]). However, we observe that compilers for type-safe programming languages utilize "general-purpose" hardware to implement memory-safety run-time checks.

Straightforward attacks on application-implemented security policies from within hardware are highly inflexible; if the malware hard-coded application-specific information within the processor, the malware would likely break when the application is updated. Jinn trojans attack compiler-based enforcement mechanisms, which use instruction sequences that remain identical across changes to an application's implementation.

Compiler-injected safety checks prevent programs from performing unsafe operations at run-time; without these run-time checks, programs may have exploitable memory safety bugs. *We recognize these injected run-time checks as* ***repetitive*** *instruction sequences that* ***implement security-enforcement mechanisms***. Instruction sequences that implement these run-time checks are structured very similarly, if

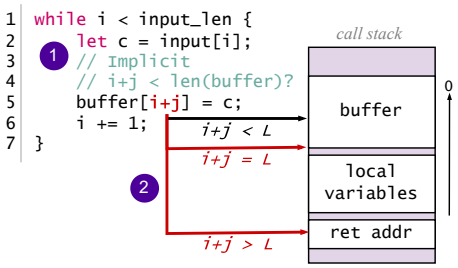

Figure 1: Steps for Software Exploitation

```
1  # i+j
2  add    0x460(%rsp), %rax
3  # Store i+j on the stack for later
4  mov    %rax, 0x8(%rsp)
5  # Compare i+j with the buffer length
6  cmp    $0x400, %rax
7  # Move status flags into GP register
8  setb   %al
9  # Conditionally Set Zero-flag (ZF) to
10 # set jump direction
11 test   $0x1, %al
12 # Jump to error handler if the index
13 # is out-of-bounds
14 jne    0x40a2df <pass>
15 jmp    0x40a305 <error>
```

Figure 2: Bounds Check Machine Code

not identically, throughout a program. For example, Kotlin injects its bounds checks [15] as a procedure-call (prior to optimizations), and Rust [43] does this by in-lining code; any array access will therefore execute a *predictable* code pattern. These strict patterns make the bounds checks easily and flexibly identifiable by a hardware trojan snooping on in-flight instructions. We show in Section 7 that such bounds-checks are sufficiently consistent to be encoded in hardware as a trigger to reliably attack critical bounds-checks in diverse software contexts. By tampering with run-time checks, trojans enable software to violate the safety properties such as memory-safety. The vulnerabilities induced by trojans that tamper with run-time checks expose programs written in type-safe languages to the wide scope of memory-safety attacks [65] such as sophisticated control-flow hijacking [25, 27, 57, 61, 68, 76] and data leaks [48, 62]..

A trojan that attempts to tamper with these safety checks must recognize the particular instructions in the dynamic instruction stream flowing through the processor's pipeline that correspond to the compiler-injected bounds check. The trojan must usefully tamper with instructions, delivering a *hardware payload* that *forces a safety-check to fail*, and *stable execution to resume*. After the trojan disables the safety-check, an attacker must exploit the memory-safety error. Figure 1 illustrates this process. First, malicious hardware recognizes the operation of an imminent bounds-check and delivers a payload that causes the bounds-check to pass (when it should fail when the processor acts benignly); then, attacker-controllable and maliciously crafted inputs cause the victim program to access memory outside the bounds of the buffer. In Figure 1, step 2 illustrates a software payload that overwrites the return address of the current frame on the call stack to initiate a control-flow hijacking attack; this attack is launched upon execution of the next return instruction.

Machine code lacks the high-level information about the purpose of each instruction; this is often referred to as the *semantic gap*. A trojan must distinguish between the `cmp` instructions that implement a bounds check from any others, such as those that implement `if` statements or looping conditions. To overcome this challenge, we have designed two different triggers that allow Jinn trojans to identify which `cmp` instruction to tamper; one such trigger is novel.

5 Jinn Trojans

We characterize the class of Jinn trojans by the payload that they deliver. Jinn trojans thwart the software-level security of SoCs by tampering with the native code of compiler-injected run-time checks.

Figure 2 illustrates a bounds check generated by the Rust compiler for the x86 instruction set. The bounds check is comprised of several instructions that compute the index of buffer being accessed (line 2), compare the index and the length (line 6), and jump to (or past) the error handler (lines 14 and 15). By tampering with bounds checks, Jinn trojans thwart the language-level safety guarantees provided by the compiler and open programs up to a broad range of memory-safety exploitation techniques.

5.1 Hardware Payloads

To tamper with the bounds check, the trojan may inject a variety of payloads. Considering the instruction sequence in Figure 2, multiple hardware payload designs can disable the array bounds-check. For example, a trojan may tamper with immediate operand of the compare instruction (`cmp`) on line 8 to represent a much larger number; this can cause the bounds-check to operate as if the buffer is much larger than it really is. Another payload may tamper with the status flags set by the `cmp` instruction that that are subsequently read by the `setb` instruction (line 8). This will cause the conditional jump (`jne`) instruction to behave as if the index (stored in `%rax`) was within the bounds of the buffer. Further, the `jne` (jump-not-equal) instruction opcode could be tampered with to behave like a `je` (jump-equal) instruction, then incorrectly transferring control to the buffer-access code. Each of these payload designs allows an index that is out of bounds to proceed to the buffer access, and equivalently expose a memory safety vulnerability. As a consequence, the completed indexing operation will read or write memory outside the buffer, allowing input to drive the program into a state in which it reads and leaks sensitive information or writes and corrupts

critical program state such as return addresses or function pointers.

5.2 Hardware Trigger

Ideally, the trojan payload should only be delivered when a bounds checking routine is executing. Consider a payload that corrupts a `cmp` instruction's immediate operand. The trojan's trigger must therefore distinguish between a *bounds checking* `cmp` and `cmp` instructions used in other parts of the same program or within other programs and the operating system.

Jinn trojans significantly benefit from trigger designs that precisely distinguish tampered instructions in the context of run-time checks from other non-critical occurrences of the same instructions.

Several trigger mechanisms enable trojans to make this distinction between execution contexts with various trade-offs, and we discuss two. We first discuss an interactive trigger design that loads identifying information for the victim run-time check's execution context, and second, a trigger that encodes a signature of the run-time check instruction sequence. Both these triggers provide the trojan with the precision necessary to identify instructions belonging to a bounds check.

5.2.1 Interactive Trigger

A trojan can use the memory address of the code stored in memory to precisely identify a particular bounds checking routine. As the threat model (Section 3) explains, we assume that an attacker is able to learn information about the memory layout of a program and that the attacker can learn the address of a specific bounds checking instruction.

The trojan can implement a covert interface for accepting this address by tampering with an additional instruction (for example, an `add` instruction) that takes two 64-bit operands where the first operand accepts the tampered bounds checking instruction address value, and the second operand accepts a secret value that tells the trojan to store the first operand into internal state that records the program counter value on which to trigger the payload. Due to the interactivity of this trigger, it's easier to reason about its capabilities with an attacker that can *run arbitrary code* with *local access* to the tampered machine. For example, an attacker that's operating within a guest VM on a cloud server may want to attack other guest VMs dispatched to the same server.

This interactive trigger implements a two-stage design, where the initial state of the trigger waits for a tampered instruction's address (the `add` instruction in the previous example), and the secondary phase waits for the program counter to match the internally stored address, and for the instruction at this address to match the instruction opcode of the instruction to be tampered (e.g., the `cmp` instruction in the example in Figure 2).

5.2.2 Run-time Check Encoded Trigger

Alternatively, a trojan can recognize an incoming bounds check by observing the data flow pattern corresponding to a bounds check. We observe that the compiler predictably generates bounds checking instruction sequences that trojans can reliably tamper with to hijack the program. By encoding logic to recognize the dataflow path of key data used to decide the jump target for the bounds check, and the associated instruction signatures, trojans can anticipate incoming bounds checks. Section 7.1.2 discusses this in further detail, and Section 9.3 presents our experiments to empirically verify the sensitivity and resilience of such a trigger.

This trigger obviates the necessity of the additional reconnaissance step of identifying a target instruction address. Rather, this trigger identifies and tampers all executions of the encoded bounds checks. We observe that under benign workloads for safely written code, bounds checks are expected to pass. Consequently, delivering the payload on a bounds check that passes induces no malicious behavior, leaving the trojan undetected. However, code that intentionally causes a bounds-check to fail, such as compiler test suites, may detect this trojan; this trojan would be detected in the unlikely scenario where both the victim software and compiler's test suite are run on the same deployed system. If an attacker anticipates this risk, this trigger design can be augmented with additional conditions such as a counter for unlikely workload events, such as floating point exceptions [73,82,83] or another metric-based characterization of the victim program.

6 Launching the Software Attack

To successfully exercise a Jinn trojan, software must first perform two tasks. First, it must prime the victim program's state to productively and immediately exploit the vulnerable software state produced by the hardware trojan; this process constitutes the reconnaissance of gadgets used in the software payload, and an attacker delivering the corresponding malicious inputs to the victim program. Next, as the victim program executes, the trojan's trigger recognizes a targeted safety check (as Section 5.2 previously discussed) and delivers the hardware payload. Finally, the previously software-injected input launches following the memory corruption yielded by the disabled safety check.

Figure 1 illustrates a victim program's call stack. The `buffer` access in the snippet of code is vulnerable to exploitation with *Jinn* trojans. When `i+j` is greater than the length of the array, dereferencing the buffer would access memory outside the bounds of the `buffer`; bounds checks prevent the invalid run-time accesses (denoted with red dashed lines). An attacker attempting to hijack this victim program must exercise the Jinn trojan to corrupt memory outside the `buffer`.

Step 0: ***Reconnaissance***

The attacker must perform reconnaissance steps to construct the payload that will be injected in to the program. The first step is for the attacker to identify an exploitable buffer access, which is characterized by a couple properties. The access must depend on a compiler-injected bounds check[1].

Further, the buffer access must operate on attacker-controllable data. Depending on the intended software payloads for attacks, either the index, or both the index and the data, must be attacker-controllable to launch memory-corruption attacks, such as buffer overflows and overreads; this step is reminiscent of traditional reconnaissance steps towards exploiting traditional memory-safety vulnerabilities [65].

As Section 3 explains, we assume that an attacker is capable of performing the necessary reconnaissance by using tools like GDB [2], `angrop` [1] ROPgadget [59] to learn information about a binary's memory layout and identify gadgets to launch control-flow hijacking attacks.

Address Space Layout Randomization (ASLR) [66] deployment may complicate a Jinn attack. ASLR is a common defense that randomizes the base addresses of several data locations (such as the stack, heap and code). A randomized the location of a bounds-checking instruction's address necessitates an additional reconnaissance step when using the interactive trigger (Section 5.2.1). ASLR is often thwarted by memory-disclosure attacks [62] which commonly rely on other memory safety vulnerabilities (such as buffer overreads [62]) to disclose the layout of the victim process. Unfortunately, within the domain of safe programming languages, this reliance is not feasible. However, attackers that can run arbitrary code on the deployed machine can launch cache-side channel attacks to learn the memory-layout of a program [20, 40, 60]. For example, if the victim is a type-safe operating system (e.g. Redox [4]), an attacker would run timing-based cache side-channel attacks to reveal code location offsets for kernel code (e.g. system-call handlers) [40], and consequently expose the ASLR offset.

Step 1: ***Priming the Victim Program***

Following identification of an exploitable buffer access and gadgets to construct the control-flow hijacking attack, the attacker must construct the corresponding software payload bytes to be delivered to the victim process. The software payload bytes are delivered preemptively, anticipating that the Jinn trojan will induce a memory-safety vulnerability.

Step 2: ***Trigger Sequence***

As the victim processes the software payload bytes, the attacker must concurrently run a triggering sequence to engage the trojan on the upcoming bounds check. Depending on the trigger design, this may include steps like priming a trojan-implemented counter or delivering a secret activation value to the trojan, as discussed in Section 5.2.

Step 3: ***Delivering Software Payloads***

As Figure 1 illustrates, an attacker that exploits a faulty bounds checks can corrupt memory locations that are reachable through the `buffer[i+j]`-dereferencing expression. An attacker with influence over the indexed location and data (as discussed in Section 3) can launch an attack that hijacks the control flow of the victim program. A simple example of this would require corrupting the return address with a chain of gadget addresses to launch a ROP attack that spawns a shell program [57].

7 Attack Implementations

We implemented three prototype Jinn trojans on the x86 out-of-order core (O3CPU) running a full-system simulation on the gem5 simulator [24], and attacked Rust programs running on Ubuntu Base 20.04.

We compiled victim Rust applications for 64-bit x86. To ease the construction of a software payload that implements a generic ROP attack [57], we use a static relocation model that produces a *non*-position-independent executable.

Rust 1.58.1 [10] supports neither stack smashing protection [72] nor backward-edge control-flow protections such as shadow stacks [12]. Therefore, the memory-corruptions enabled by Jinn trojans attacking Rust executables are not inhibited by mitigation schemes commonly used to defend against memory-safety attacks in type-unsafe languages.

We implement our prototypes in the Issue-Execute-Writeback (IEW) stage of gem5's O3CPU. The IEW stage comprises Issue, Execute, and Writeback routines; however, we model our code to isolate modifications to the core within the *Execute* logic. We limit all modifications to the core to the C++ function that defines the execute semantics of each instruction. We expect that the changes necessary to implement the Jinn trojan resemble modifications isolated to a single IP.

7.1 Variant 1: Attacking Indexed Buffers

Our first prototype attacks an indexed buffer that relies on an implicit bounds-check injected by the Rust compiler. We implement a trojan that uses the run-time check encoded trigger discussed in Section 5.2.2 and delivers a payload that corrupts the status flags set by a bounds-checking `cmp` instruction. We then exploit the memory safety error induced by our Jinn trojan to overwrite a return address on the call stack, hijacking the control flow of the victim Rust program and spawning a shell.

[1] This needs special consideration since compilers can optimize away redundant bounds checks.

```
1  # Compare length (0x400) and index (%rax)
2  cmp     $0x400, %rax
3  # EFLAGS = [PF IF CF]
4  # Load status flags into register (%al)
5  setb    %al
6  # %al = 0x1
7  # Conditionally set Zero-flag (ZF)
8  # to control jump direction
9  test    $0x1, %al
10 # EFLAGS = [IF] (unset ZF)
11 # Jump to buffer access
12 jnz     0x40a2df # jump if ZF is unset
13 # Jump to error handler
14 jmp     0x40a305
```

Figure 3: Tampering with a Bounds Check

7.1.1 Payload

We choose a hardware payload that tampers with the status flags controlled by the `cmp` instruction within the bounds check. This payload implementation allows us to set a single bit, minimizing the payload's complexity. The payload *sets* the carry flag in the EFLAGS register regardless of the result of the `cmp` instruction.

We observe that a payload that triggers on a `cmp` instruction can thwart the bounds checks of other languages, allowing our trojan to defeat the bounds checks of other programming languages. In addition to Rust, Go implements bounds checks with a critical `cmp` instruction on x86. Consequently, interactive trigger designs (such the one discussed in Section 5.2.1) allow the trojan to deliver a payload to flexibly undermine both Rust and Go bounds-checks since they implement identically structured bounds checks.

Corrupting the carry-flag causes the bounds-check to transfer control to the code which accesses the array as opposed to the `panic` handler which should be executed when a bounds-check fails.

Figure 3 illustrates the effects of our trojan. Assuming that the register `%rax` holds an index value greater than the length of the accessed buffer (`0x400`), an untampered `cmp` instruction (line 2) would clear the carry-flag (i.e. set it to `0`). By setting the carry-flag to `1`, the subsequent `setb` instruction (line 5) writes 1 to the register operand `%al`. When `%al` is 1, the following `test` instruction (line 9) clears the zero-flag (`ZF`) (sets to 0), and the subsequent `jnz` transfers control flow to the buffer-access code, *passing* the bounds check.

The Gem5 O3CPU divides the `cmp` instruction into two micro-ops:

- Load-immediate (`li`), which loads the statically identified buffer-length into a physical register
- Sub-flags (`sub`), which subtracts the value in the immediate-loaded physical register from a renamed architectural register that contains the index offset into the memory buffer. The subtraction's result is then used to update the status flags.

```
1  ADD_R_M  : ld   t1, DS:[t0 + rsp + 0x4c8]
2  MOV_R_M  : ld   rcx, DS:[t0 + rsp + 0x18]
3  MOV_R_M  : ld   cl, DS:[rcx]
4  ADD_R_M  : add  rax, rax, t1
5  MOV_M_R  : st   rax, DS:[t0 + rsp + 0x10]
6  SETB_R   : movi  al, al, 0x1
7  TEST_R_I : limm  t1b, 0x1
8  JNZ_I    : rdip  t1, t1
9  JNZ_I    : limm  t2, 0x14
10 CMP_R_I  : limm  t1, 0x400
11 SETB_R   : movi  al, al, 0
12 MOV_R_M  : ld   rax, DS:[t0 + rsp + 0x10]
13 TEST_R_I : limm  t1b, 0x1
14 JNZ_I    : rdip  t1, t1
15 JNZ_I    : limm  t2, 0x1a
16 MOV_M_R  : st   cl, DS:[t0 + rsp + 0xf]
17 TEST_R_I : and   t0b, al, t1b
18 MOV_M_R  : st   cl, DS:[t0 + rsp + 0x4ff]
19 JNZ_I    : wrip  t1, t2
20 CMP_R_I  : sub   t0, rax, t1
```

Figure 4: OoO Micro-ops Observed During Bounds Check

This payload is delivered as part of the `SubFlags` micro-op. This temporary alteration to the behavior of the `cmp` instruction doesn't corrupt any extra architectural state, obviating any risks to stable execution after delivering a payload.

7.1.2 Trigger

We implement the trigger design discussed in Section 5.2.2 that encodes the sequence of micro-ops that precede the targeted `cmp` instruction in a bounds check.

Figure 4 lists the sequence of in-flight micro-ops used to implement the bounds check and array access as they pass through the IEW stage. We implement a trojan trigger that recognizes this sequence of micro-ops and then triggers the malicious logic that delivers the payload on the final micro-op of the sequence (`sub`).

This listing represents one of many observed micro-op execution sequences observed during execution of the same Rust statement (line 4). These sequences vary significantly due to extra-microarchitecural state that is not observable from a trojan isolated within the IEW stage. These performance optimizations challenge the design for a reliable trojan trigger; however, we recognize that the data dependence between values calculated to perform the bounds check must be maintained. The arrows in Figure 4 point from a physical or architectural register operand to a preceding operation's

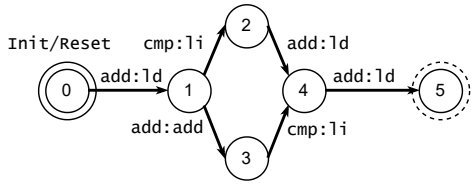

Figure 5: Trigger States

output on which it depends.

Figure 5 shows a partial illustration of the finite state machine that our trigger implements. The edges depicting state transitions correspond to the in-flight instructions (formatted as **`macro-op:micro-op`**) that the trojan observes. The trigger implementation encodes the micro-ops for instructions that compute critical values along the dataflow path to the payload-targeted sub-flags micro-op. In its inital state, the trigger snoops for an `add` instruction's micro-op, `ld`, that attempts to load a value from the stack into a physical register. Subsequently, the trigger then snoops for two micro-ops that may arrive out of order:

- The first micro-op of the `cmp` instruction that we wish to tamper, loading an immediate value to a physical register (the source-level buffer's length)
- The second micro-op of the `add` instruction that adds the source-level index and offset.

At its final pre-triggered state, the trigger snoops for the second micro-op of the `cmp` instruction that implements the subtraction to compute the status flags.

The trigger progressively advances its state as it observes instructions that match the bounds checking sequence. Additionally, the trigger implements a decay-counter that increments when it observes irrelevant instructions and resets the trigger state to the initial state upon reaching a threshold. These would correspond to backward edges in Figure 5 from each node back to the `Init/Reset` state; we omit these edges from Figure 5 for clarity, but we implemented them in our trigger logic. Additionally, trigger activation both delivers the payload and resets the trigger state back to the `Init/Reset` state.

In our experiments, we found that a decay threshold of seven micro-ops provided sufficient accuracy to detecting bounds-checking instruction sequences. This threshold is decided by various properties of the hardware design like presence of out-of-order execution, speculative components, superscalar operation, etc. To further improve the precision of the trigger, we implement logic that allows the trojan to recognize instructions that are contextually appropriate while executing a bounds check out of order– for example, the control-transfer instructions (`jnz` and `jmp`) that follow a bounds check to set up the buffer access; such instructions do not affect the trigger's decay counter.

7.1.3 Exploit

The victim program implements a simple procedure that copies bytes from standard input to a statically allocated buffer at a parameterized offset from the start of the buffer.

We use angrop [1] to analyze the victim binaries for gadgets to construct a return-oriented-programming (ROP) attack payload. As illustrated in Figure 1, the tampered bounds check

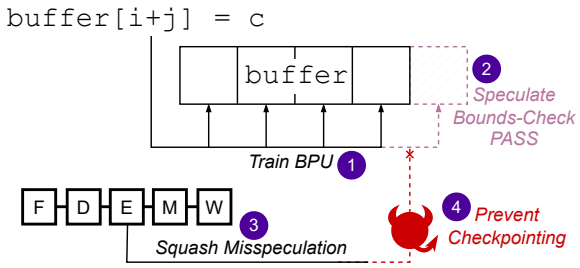

Figure 6: Attacking Mispredicted Bounds-Checks

allows indexed-buffer to write outside the bounds of the buffer. To exploit this vulnerability, we inject a sequence of bytes into the program. The program then attempts to copy as many bytes as it can to fill the `buffer`; however, this condition is never met since Jinn trojan causes the bounds check to fail. Instead, the program keeps reading bytes until the input buffer is exhausted. We input a maliciously crafted set of bytes that overwrite the return address to launch a return-oriented programming (ROP) attack [57] that launches a shell.

7.2 Variant 2: Mispredicted Bounds Checks

To demonstrate the flexibility of payloads that can implement Jinn trojans, we implement an attack that corrupts the branch resolution logic that squashes microarchitectural state on mispredicted branches. Several successful bounds-checks in sequence train micro-architectural branch predictors to transfer control to the memory-access (as opposed to the panic handler). This variant tampers with branch resolution to prevent mispredicted bounds–checks from being squashed, consequently committing memory accesses that would have failed their bounds-checks.

7.2.1 Payload

The trojan's payload targets the conditional jump instruction (`jnz`) that implements the control-flow transfer to memory access code that is guarded by a bounds-check. Microarchitectures that implement branch-predictors will speculatively execute past this branch instruction to execute either the memory-access or the bounds-check panic code. By exploiting a branch predictor state that's trained to pass bounds-checks, a payload that tampers with branch resolution/misprediction can *prevent* the processor from squashing and rewinding execution for incorrect speculatively executed bounds-checks. The payload hijacks the logic that checks for mis-speculation. When the trojan's trigger is engaged, the payload suppresses the checkpointing logic that reverts the microarchitectural state upon identifying misspeculation.

As Figure 6 illustrates, repeated writes within the bounds of a bounds-checked buffer first train the branch-prediction unit (BPU). The subsequent access outside the bounds of

the buffer is predicted to pass. Control is then speculatively transferred across the conditional jump to the buffer-access (instead of the panic handler for an out-of-bounds access). During branch-resolution, the microarchitecture discovers that it mispredicted the target of the conditional jump and incorrectly transferred control to the buffer-access code. At this point, a trojan payload prevents the microarchitecture from reverting to the checkpoint.

Following the delivery of this hardware payload, the microarchitecture continues stable execution as if the bounds-check had passed, exposing an exploitable memory-safety vulnerability.

7.2.2 Trigger

We implement the interactive trigger discussed in Section 5.2.1 to pair with the payload discussed in the previous section. We tamper with the operation of the `add` instruction to store the first operand into internal state if the second operand matches a secret hard-coded value that would be decided at design-time. Following that, the trojan snoops on in-flight instructions, searching for an instruction address that matches the stored target instruction address. Once it identifies a matching instruction, it validates that the target instruction is a `jnz` instruction and delivers the payload.

7.2.3 Exploit

The trojan-induced memory-safety attack must first train the branch predictor to speculatively pass bounds-checks and to branch to the guarded buffer-access. An attacker that can control victim software indices must therefore access several locations within the bounds of the buffer before the trojan can usefully deliver its payload. Exploiting the trained and tampered branch prediction logic, our implementation of the attack overwrites memory locations offset from the buffer. Similar to the previously discussed exploit in Section 7.1.3, we use this memory-safety vulnerability to launch a ROP attack that spawns a shell.

7.3 Variant 3: Attacking Rust Iterators

We implement a third attack that attacks Rust iterators within the same gem5 O3CPU platform and stage as discussed in Section 7.1. Rust provides *iterators* as code patterns or templates to idiomatically process sequences of data objects from a parameterized data structure. They differ from buffers that are *indexed* since they don't expose an interface for random access within the set of elements. Instead, elements are fetched from iterators sequentially, and library code ensures that an internal pointer marking the *current element* never exceeds the bounds of the data structure. This attack exercises a Jinn trojan that implements an interactive trigger (discussed in Section 5.2.1) to compromise the comparison of an iterator's internal *current element* pointer to the bounds of the structure.

```
1  let iter = buffer.iter_mut();
2  // Iterate through local buffer
3  for elem in iter {
4      // Copy elements from input iterator
5      *elem = match input_iter.next() {
6          // "x" contains a value
7          Some(x) => *x
8          // "input_iter" is empty
9          None => break
10     };
11 }
```

Figure 7: Iterator Implementation in Rust

```
1  # Load the pointer to the current element
2  mov     0x18(%rsp), %rax
3  # Load a pointer to the iterator object
4  mov     0x20(%rsp), %rcx
5  # Compare bounding pointer in the iterator
6  # to the current element pointer
7  cmp     0x8(%rcx), %rax
8  # Conditionally jump to code that handles
9  # an empty iterator
10 je      0x409d7d <empty>
11 # Otherwise, fetch the next element
12 mov     0x20(%rsp), %rax <fetch next>
13 [...]
14 # Empty iterator code
15 mov     0x0, 0x30(%rsp) <empty (0x409d7d)>
```

Figure 8: Iterator Pointer Comparison Machine Code

7.3.1 Payload

The payload of this Jinn trojan variant targets the `cmp` instruction that implements the Rust iterator's bounds-check. This "bounds-check" is a Rust source-specified bounds-check. Rust ships this iterator implementation as part of its `core` crate (set of libraries). Its implementation differs from the bounds checks discussed previously that are inserted by the compiler during code generation. However, it maintains the same properties that make it an ideal candidate for tampering by a trojan: it's consistently structured and used across a variety of applications.

Figure 7 shows a source-level implementation of an iterator [8] in Rust. Source-line 1 declares and instantiates the mutable iterator (`iter`) with which the Jinn trojan tampers. At line 3, a `for` loop statement iterates over elements in the iterator. The body of the loop (lines 5-9) consists of a statement that copies elements from another iterator (`input_iter`) into the local `buffer`. A trojan that tampers with the `for` loop can cause the assignment at line 5 to write to elements *beyond* the bounds of the `buffer`.

Figure 8 shows the machine code that implements a portion of the iterator code, defined in Rust's *core* library, and called at source-line 3 in Figure 7. The `mov` instructions, which load the current-element pointer (line 2) and bounding iterator pointer (line 4), prepare operands for the `cmp` instruction at line 7. This comparison sets condition flags that are used in the following instruction to transfer control flow depending on if the iterator still contains elements. Iterators will update internal data each time an element is extracted, and increment

the current-element pointer that's stored in the architectural register, `%rax`. Upon reaching the end of the structure, the condition codes set by the `cmp` instruction (line 7) will cause the following jump-equal (`je` at line 10) to transfer control to code that handles an empty iterator.

We implement a Jinn payload that tampers with the condition code for the `cmp` instruction. The payload clears the zero-flag (ZF) used by the `je` instruction, and this allows the iterator to increment its current-element pointer to memory addresses that are beyond the bounds of the `buffer`. To exploit this vulnerability induced by the trojan, an attacker that interfaces with the victim program can deliver inputs and use the Jinn-tampered `for` loop to write maliciously crafted exploit-bytes to corrupt critical memory locations such as a frame's return address.

7.3.2 Trigger

We implement the interactive trigger discussed previously (Sections 5.2 and 7.2.2). An attacker interfaces with the trojan by executing an `add` instruction with a hardcoded operand, and the trigger stores the second operand as the target instruction to be tampered. This trigger intends to target a `cmp` to deliver the payload discussed in the previous section.

7.3.3 Exploit

The vulnerability exposed by this Jinn trojan variant resembles our previous variants (Sections 7.1.3 and 7.2.3) but with a slight difference. Since the iterator doesn't expose an indexing interface, the attack must overwrite *all* bytes in memory that are located between the buffer and the return address. As Figure 1 illustrates, other program data for the current frame, such as local variables and function arguments, may be located between the `buffer` and the return address.

In our experiments, the Rust compiler typically places the iterator (`iter` in Figure 7) in this region; this challenges the attack since a naive approach of corrupting all data between the buffer and return address would corrupt the internal data of the iterator. We overcome this challenge by intelligently corrupting the internals of the iterator to cause it to point to the return address on the stack. We do this by corrupting the internal pointer to the "next" element that would be served by the iterator. Subsequent operations on elements served by the iterator then corrupt the return address. Finally, similar to the exploit discussed in previously, we construct an ROP attack that launches a shell.

8 RTL Prototype

To get a more accurate representation of the complexity of Jinn trojans, we evaluate an RTL implementation of the *Variant 1* payload (Section 7.1) using the interactive trigger (Section 5.2.1). We implemented this on the RISC-V Berkeley Out-of-Order Machine (BOOM) core [5].

Variant	Component	C++ SLOC (*count*)
Baseline	Gem5 O3CPU	16,626
Variant 1	Carry-flag Payload	2
	RTC-encoded Trigger	63
Variant 2	BPU-Payload	1
	Interactive Trigger	30
Variant 3	Carry-Flag Payload	2
	Interactive Trigger	30

Table 1: Jinn Trojan Source Lines of Code (SLOC)

We implement this trojan by adding *three* lines of Chisel [18] code to the ALU within the core's execute stage. These three lines comprise logic that: (1) declares a 64-bit register; (2) implements the *trigger* by snooping for a secret value as the first, operand to an `add` instruction, storing the second operand to the declared register if the first operand matched; and (3) delivers the payload to a target RISC-V `bltu` instruction when the target instruction's address matches the value stored by the trigger.

We run a bare-metal application atop the tampered core to verify the functionality of the implementation. The application comprises a C program that calls to linked Rust functions that implement bounds-checks. We verify our RTL Jinn trojan successfully allows an attacker to escape inject a memory-safety guarantee in the Rust functions, and maliciously divert control flow to a maliciously invoked function.

9 Evaluation

We evaluate the complexity of our Jinn trojan prototypes implemented within the gem5 out-of-order core (O3CPU) and RISC-V BOOM core implementations, and the sensitivity of the run-time check encoded trigger against other large codebases. Further, we discuss a couple potential real-world attack vectors in third-party code.

9.1 Gem5 Evaluation

To measure the complexity of our prototype gem5 trojans, we counted source-lines of code (SLOC) using SLOCCount 2.26 [7]. Table 1[2] shows the results; Table 2 lists the complexity of the internal state of our trojan implementations.

Variant 1, which implements the run-time check encoded trigger, trades storage complexity for increased logical complexity in comparison with Variants 2 and 3. This is because Variant 1 requires manual encoding of each of the instructions

[2]The Run-time check encoded trigger is abbreviated to RTC-trigger.

Variant	Stateful Component	Complexity (*bits*)
Variant 1	Trigger Stages	4
	Decay Counter	4
	Total	8
Variant 2	Trigger Stages	1
	Target Address	64
	Total	65
Variant 3	Trigger Stages	1
	Target Address	64
	Total	65

Table 2: Gem5 State Complexity

Category	Baseline (*count*)	Jinn (*count*)	Overhead (%)
Gates	303,092	303,217	< 0.001

Table 3: BOOM Core Gate-count Comparison

encoded within the run-time check. In contrast, Variants 2 and 3 use significantly more dynamic storage with less logical complexity. However, as Section 5.2 discusses, the attacker's capabilities dictate which trigger is best.

Untriggered operation of the trojan has no impact on architectural state and, consequently, does not affect dynamic instruction stream post-deployment. While gem5 does not expose timing perturbations due to additional logic within the core, our trojans do not impact the tampered functional unit's critical path because (1) benign operation doesn't depend on trojan snooping (and vice versa), and (2) malicious logic doesn't exceed the complexity of benign execute-stage logic. Therefore, we expect our prototype trojan to incur negligible (if any) performance overhead.

9.2 RTL Evaluation

We use the Chipyard [16] VLSI flow with BOOM core configuration defaults and Cadence plugins for Genus and Joules to measure power consumption post-synthesis. Table 4 lists the power consumption of the baseline BOOM core and the same core tampered with a Jinn trojan. We observe very low power overheads (on the order of 0.1%); these are unlikely to cause tampered cores to exceed design-time power budgets.

Category	Baseline (*mW*)	Jinn (*mW*)	Overhead (%)
Register	9.566	9.578	0.125
Logic	40.000	40.047	0.118
Total (Static)	49.567	49.625	0.117
Dynamic	5,152	5,153	< 0.001

Table 4: Power Analysis of BOOM Core

Further, Jinn trojans do not have an effect on architectural-level performance because they do not affect benign execution during untriggered operation, and are small enough to feasibly fit within existing path-delay-constraints, and therefore not incur extra operating cycles. Additionally, we use Genus to verify that our RTL implementation indeed does not introduce sufficient complexity to put it on the critical-path under the default timing constraints of the BOOM core.

Design-time trojan complexity does not directly determine a trojan's stealthiness because mitigation schemes search for typical behavioral properties of trojan logic irrespective of the size of logical blocks. Trojan complexity is therefore not a useful metric for comparing evasiveness with other design-time trojans. Complexity metrics *are* useful to demonstrate the effort an attacker must deliver to implement a trojan design. So, while power consumption and gate-counts don't influence the stealthiness of *design-time* trojans (unlike *fabrication-time* trojans [21, 46, 79])[3], we nonetheless provide these details to further (1) communicate the complexity of our Jinn trojan's implementation, (2) demonstrate that it insignificantly impacts performance requirements, and (3) show the low attacker-effort necessary to implement such trojans.

9.3 Trojan Evasiveness

A successful trojan must evade the full range of trojan mitigation schemes to successfully tamper with a deployed system. As our threat model in Section 3 explains, trojan detection schemes can analyze netlist-level and RTL designs from untrusted third-party intellectual property (3PIP) to identify suspicious circuitry. Since manual inspection techniques aren't scalable to larger designs, and attackers can exercise design-obfuscation schemes [17], we focus on automated schemes. Trojan-mitigation schemes broadly analyze hardware and search for suspicious (redundant) logic [36, 82], specialized triggering mechanisms [47, 71, 73], and information flow properties [23, 35, 37, 38, 51].

UCI [36], FANCI [73] and VeriTrust [82] are all defeated by trigger transformations [83] to avoid exhibiting suspicious properties. We rely on such transformations to Jinn trojan RTL trigger designs to evade detection from these analyses.

Bomberman [71] analyzes hardware designs for Ticking-Timebomb Triggers (TTTs); triggers that monotonically count system-events such as page-faults. Our interactive trigger (discussed in Section 5.2.1) does not implement a counter or state-machine that increments throughout execution. Likewise, our run-time check encoded trigger (discussed in Section 5.2.2) violates a property of TTTs by resetting its state machine periodically (and is therefore not a monotonic counter), thus

[3] Power-analysis schemes for trojan detection typically require a trojan-free design for comparison. This is not available within the threat-models of design-time trojans.

it avoids being flagged as suspicious. Therefore, both of the trigger designs discussed in Section 5.2 evade Bomberman.

Information-flow analysis for hardware trojan detection attempts to verify lattice properties of annotated hardware designs. To consider applying traditional information flow analyses [23, 35, 37, 38, 51], tampered signals demonstrated in this paper's attacks, such as the status flags set by `cmp` instructions, must be assigned a security label and verified against integrity properties. It's unclear how existing information flow analysis security labels and policies would applied to mitigate Jinn trojans since Jinn trojans don't tamper with conventionally high-security labels; no such clear demarcations exist yet for the logic that Jinn trojans tamper.

9.3.1 Trigger Limitations

The **interactive trigger** provides high accuracy for delivering the payload. It relies on hiding within the state-space proportional to the size of general-purpose registers. On a 64-bit system, this likely evades detection during verification testing since it is infeasible to engage the trojan initial trigger by testing the full range of 2^{64} values across all the operands of multi-operand instructions. At 1 *billion* tests per second, testing all values for a `single` 64-bit register would take approximately 507 years.

The **run-time check encoded trigger** relies on the assumption that deployment software will *not* panic. This *may* happen when the deployed system with a planted trojan is the same system that is used to test/debug victim software. Under panic'ed execution, the trojan will instead exhibit memory-safety vulnerability and will result in undefined-behavior.

9.4 Real-World Code

To verify that Jinn trojans can attack real-world software, we identified vulnerable code in third-party Rust programs. As our threat model in Section 3 discusses, memory-safety attacks typically utilize an attacker-controlled variable to induce a program to write to a memory location outside the bounds of a pointer's referent memory object.

We identify two real-world vulnerable Rust code sequences: one in the Rust-based operating system Redox [4] and another in the Rust-based web engine Servo [6].

```
  fn encrypt(&mut self, data: &mut [u8]) -> bool
1 {
2   [...]
3   for i in 0..data.len() / BLK_SIZE {
4     self.aes_blocks.push( [...]
5       &data[i * BLK_SIZE..(i + 1) * BLK_SIZE],
  ));
```

Figure 9: Vulnerable Code Sequence in Redox

Figure 9 is a code-listing from Redox source-code (simplified for brevity) that encrypts file-system blocks. We observe that `data` is attacker-controllable, and chunks are moved from this input parameter to an object-local structure (`aes_blocks`). The `push` function call will first check the length of the `aes_blocks` structure and decide to either proceed with the memory-write or first "grow" (or enlarge) the structure if it has reached capacity. Figure 10 lists the `if`-statement within library-implemented code that the trojan would tamper with; more specifically, it would tamper with the underlying comparison instruction that's generated by the compiler to implement the check. A Jinn trojan can tramper with a branch within `push` so that the size of `data` appears sufficient when, in fact, it needs to be enlarged. The remaining code then erroneously writes past the end of the `data` buffer.

```
1 pub fn push_back(&mut self, value: T) {
2   if self.is_full() {
3     self.grow();
4   }
5   [...] // Write to memory
```

Figure 10: Bounds-Check in Library Code

Figure 11 similarly lists (simplified) code from Servo [6], a web engine written in Rust. This snippet is from an HTML5 tokenizer that parses attacker-controllable HTML.

```
1 fn tokenize(input: Vec<StrTendril>, opts: TokenizerOpts)
2             -> Vec<(Token, u64)> {
3   [...]
4   let mut buffer = BufferQueue::new();
5   for chunk in input.into_iter() {
6           buffer.push_back(chunk);
7           [...]
```

Figure 11: Vulnerable Code Sequence in Servo

The write to a local `buffer` exposes a similar attack vector for a Jinn-trojan as discussed in the previous example. Both these code snippets expose attacker-controllable interfaces that can exploit a memory-safety vulnerability injected by a Jinn-trojan. Due to the structure of this victim code, however, the interactive trigger (Section 5.2.1) is likely to be more effective since the bounds-check is implemented by a Rust core library, similar to the iterator attack discussed in Section 7.3.

9.5 Bounds Checking Instruction Sequences

To determine whether existing software could accidentally activate our trojan, we developed a binary analyzer to search for the instruction sequences used to trigger our Jinn trojan. We use the `angr` [75] Python binary analysis framework to implement our binary analyzer. We then used this tool to search for the bounds-checking instruction sequences (the sequence illustrated in Figure 5, and listed in Figure 2) in real programs. When observing instructions, the analyzer abstracts some details away like particular memory offsets, constant values, and GP-register identifiers to accommodate differing memory layouts and register selection; for example, the analyzer would search for `add` instructions that load memory at an offset from

the stack pointer (`%rsp`) and store to a GP-register (illustrated in Appendix Figure 12). The analyzer found exceedingly few occurrences of the target instruction sequence in large code bases, with only a single false-positive trigger in the Apache web server (as listed in Appendix Table 5).

In our experiments through prototyping the attacks, we empirically observed no accidental triggers of the trojan from software excluding the bounds check. Thus, we expect the Jinn trojan to reliably deliver the payload during a bounds check. While the run-time check encoded triggers provide an attack vector that relieves an attacker of a reconnaissance step, they are sensitive to compiler build flags such as optimizations. Our experiments show that an attacker attempting to deploy such a trojan must ascertain the build flags used on the deployed system prior to trojan placement, however, this is likely to be the highest optimization level for performance-conscious deployments.

10 Possible Mitigations

In the short term, we believe that diversifying instruction selection for the run-time checks may help thwart Jinn trojans. Our malware leverages the fact that type-safe language compilers often emit the *same* code sequence for performing dynamic array bounds checks and type safety checks. If the compiler could insert different instruction sequences for different run-time checks, or if a dynamic loader could randomize the instructions used for run-time checks each time a program is loaded, it would be much more difficult for Jinn trojans to corrupt the execution of the run-time checks.

Longer term, we think a strong defense would be to identify the circuits within the processor that must be tamper-free to correctly implement instructions used by run-time checks. This analysis would enable more rigorous reasoning about which circuits are security-critical when accounting for *software-enforced* security policies; once identified, such security-critical IP could be designed in-house.

11 Related Work

Hardware trojans are typically organized into two broad categories: design-time trojans [32, 39, 42, 45, 83] and fabrication-time trojans [21, 29, 41, 46, 54, 79]. Design-time trojans are likely to be limited to smaller IP blocks that are typically outsourced, but they also have access to a higher-level description of the hardware at the HDL-level. Conversely, fabrication-time trojans can tamper with *any* portion of an SoC but are limited to the behavior of the hardware that can be discovered via reverse-engineering [54, 55].

While our prototype Jinn trojans are design-time trojans, we believe that creating fabrication-time Jinn trojans is also possible and relatively straightforward. Therefore, we direct our comparison of Jinn trojans to previous work by reasoning specifically about versatility of the trojan **payloads**.

Prior work on attacks against user/kernel isolation mechanisms, arbitrary privileged code execution [42, 79]. Jinn trojans similarly enable such capabilities by allowing an attacker to hijack code written in safe programming languages, including type-safe operating systems [3, 14, 22, 78] and trusted execution environments (TEEs) [74].

Privilege escalation remains a powerful capability enabled by prior work. There exist both design-time and fabrication-time defenses that are premised on identifying such critical signals [35, 37, 38, 51]. The privilege-bits implement well-studied and critical hardware-enforced security, and trojan-mitigation schemes are thus well tuned to identifying trojans that attack them; layout-hardening [70] and physical inspection [28] are two examples of such mitigation schemes. Jinn trojans in contrast *do not attack hardware-enforced security signals* and are hence more stealthy while enabling an attacker with similar capabilities. For similar reasons, payloads in cryptographic logic [21,46] are detected by detection schemes [64] that target such hardware-enforced security. Further, Jinn trojans also enable attackers attempting to leak keys from a process's address space by using a ROP attack to spill secret keys to an output vector (e.g. `stdout`).

Trojan attacks in the memory hierarchy [29, 41] enable three capabilities: fault injection, information leakage, and denial of service. The HarTBleed trojans [29] implement payloads against hardware-enforced security checks in the TLB to compromise page-table mappings. However, these attacks are limited to narrow attack scenarios with constraints of hard-coded (in hardware) physical frame location must coinciding with program-load-time secret data. In contrast, programs hijacked using Jinn trojans enable an attacker to launch a ROP attack that can arbitrarily read and write the victim's memory contents.

Jinn trojans flexibly deliver the wide range of payloads described above with a *single instantiation* since the complexities of attack logic is *pushed out of hardware* and *into the gadgets* [57] of a hijacked victim program.

12 Conclusion

We presented *Jinn* trojans, a novel class of hardware trojans characterized by their payloads that attack safety guarantees provided by typesafe programming languages. Jinn trojans induce memory-safety vulnerabilities by compromising compiler-injected safety checks. We demonstrated the efficacy of this class of trojans by implementing end-to-end attacks that exercise Jinn trojans to compromise Rust Programming Language's bounds checks to hijack a victim program's control-flow to launch a shell. With Jinn trojans, we demonstrate that software-level security policies can be flexibly compromised by a trojan placed in traditionally *non-security-critical* hardware components, that is, components

that are not responsible for implementing hardware-enforced security policies.

References

[1] angrop. https://github.com/angr/angrop. [Online; accessed 02-March-2022].

[2] GDB: The GNU Project Debugger. https://www.sourceware.org/gdb/. [Online; accessed 02-March-2022].

[3] Github Organization: Rust for Linux. https://github.com/Rust-for-Linux. [Online; accessed 14-March-2022].

[4] Redox: Your Next(Gen) OS. https://www.redox-os.org/. [Online; accessed 22-Sep-2022].

[5] RISC-V BOOM. https://boom-core.org/. [Online; accessed 22-Sep-2022].

[6] Servo. https://servo.org/. [Online; accessed 22-Sep-2022].

[7] SLOCCount. https://dwheeler.com/sloccount/. [Online; accessed 29-Mar-2022].

[8] The Rust Programming Language: Functional Language Features: Iterators and Closures. https://doc.rust-lang.org/book/ch13-02-iterators.html. [Online; accessed 31-Mar-2022].

[9] The Rust Reference: Array and slice indexing expression. https://doc.rust-lang.org/reference/expressions/array-expr.html#array-and-slice-indexing-expressions. [Online; accessed 29-Mar-2022].

[10] The rustc book: Exploit Mitigations. https://doc.rust-lang.org/beta/rustc/exploit-mitigations.html. [Online; accessed 30-Dec-2021].

[11] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control flow. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering*, pages 111–124, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[12] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information Systems Security*, 13:4:1–4:40, November 2009.

[13] Sally Adee. The hunt for the kill switch. *IEEE Spectrum*, 45(5):34–39, May 2008.

[14] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, MSPC '06, pages 1–10, New York, NY, USA, 2006. ACM.

[15] Marat Akhin and Mikhail Belyaev et al. Kotlin language specification: Kotlin/Core. JetBrains / JetBrains Research https://kotlinlang.org/spec, 2020. [Online; accessed 23-Feb-2022].

[16] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21, July 2020.

[17] Sarah Amir, Bicky Shakya, Domenic Forte, Mark Tehranipoor, and Swarup Bhunia. Comparative analysis of hardware obfuscation for ip protection. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, GLSVLSI '17, page 363–368, New York, NY, USA, 2017. Association for Computing Machinery.

[18] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 1216–1225, New York, NY, USA, 2012. Association for Computing Machinery.

[19] Brian Bailey. Incremental design breakdown. https://semiengineering.com/incremental-design-breakdown/, 2022. [Online; accessed 29-March-2022].

[20] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. CAIN: Silently breaking ASLR in the cloud. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., August 2015. USENIX Association.

[21] Georg T. Becker, Francesco Regazzoni, Christof Paar, and Wayne P. Burleson. Stealthy dopant-level hardware trojans. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, pages 197–214, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[22] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. Copper Mountain, CO, USA, 1995.

[23] Mohammad-Mahdi Bidmeshki and Yiorgos Makris. Vericoq: A verilog-to-coq converter for proof-carrying hardware automation. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 29–32, May 2015.

[24] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011.

[25] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, page 30–40, New York, NY, USA, 2011. Association for Computing Machinery.

[26] Rajat Subhra Chakraborty and Swarup Bhunia. Rtl hardware ip protection using key-based control and data flow obfuscation. In *2010 23rd International Conference on VLSI Design*, pages 405–410, Jan 2010.

[27] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, page 559–572, New York, NY, USA, 2010. Association for Computing Machinery.

[28] Franck Courbon, Philippe Loubet-Moundi, Jacques J.A. Fournier, and Assia Tria. A high efficiency hardware trojan detection technique based on fast sem imaging. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 788–793, March 2015.

[29] A. De, M. N. I. Khan, K. Nagarajan, and S. Ghosh. Hartbleed: Using hardware trojans for data leakage exploits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–12, 2020.

[30] Stephanie Drzevitzky, Uwe Kastens, and Marco Platzner. Proof-carrying hardware: Towards runtime verification of reconfigurable modules. In *2009 International Conference on Reconfigurable Computing and FPGAs*, pages 189–194, Dec 2009.

[31] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[32] Mathieu Gross, Nisha Jacob, Andreas Zankl, and Georg Sigl. Breaking trustzone memory isolation through malicious hardware on a modern fpga-soc. In *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop*, ASHES'19, page 3–12, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Ujjwal Guin, Qihang Shi, Domenic Forte, and Mark M. Tehranipoor. Fortis: A comprehensive solution for establishing forward trust for protecting ips and ics. *ACM Trans. Des. Autom. Electron. Syst.*, 21(4), may 2016.

[34] Xiaolong Guo, Raj Gautam Dutta, Jiaji He, Mark M. Tehranipoor, and Yier Jin. Qif-verilog: Quantitative information-flow based hardware description languages for pre-silicon security assessment. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 91–100, May 2019.

[35] Xiaolong Guo, Huifeng Zhu, Yier Jin, and Xuan Zhang. When capacitors attack: Formal method driven design and detection of charge-domain trojans. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1727–1732, March 2019.

[36] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *2010 IEEE Symposium on Security and Privacy*, pages 159–172, May 2010.

[37] Wei Hu, Armaiti Ardeshiricham, Mustafa S Gobulukoglu, Xinmu Wang, and Ryan Kastner. Property specific information flow analysis for hardware security verification. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, page 1–8. IEEE Press, 2018.

[38] Wei Hu, Baolei Mao, Jason Oberg, and Ryan Kastner. Detecting hardware trojans with gate-level information-flow tracking. *Computer*, 49(8):44–52, Aug 2016.

[39] Wei Hu, Lu Zhang, Armaiti Ardeshiricham, Jeremy Blackstone, Bochuan Hou, Yu Tai, and Ryan Kastner. Why you should care about don't cares: Exploiting internal don't care conditions for hardware trojans. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 707–713, Nov 2017.

[40] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205, May 2013.

[41] Mohammad Nasim Imtiaz Khan, Karthikeyan Nagarajan, and Swaroop Ghosh. Hardware trojans in emerging

non-volatile memories. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 396–401, March 2019.

[42] Samuel T. King, Joseph Tucek, Anthony Cozzie, Chris Grier, Weihang Jiang, and Yuanyuan Zhou. Designing and implementing malicious hardware. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, LEET'08, USA, 2008. USENIX Association.

[43] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, USA, 2018.

[44] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspot™ client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1), may 2008.

[45] Christian Krieg, Clifford Wolf, and Axel Jantsch. Malicious lut: A stealthy fpga trojan injected and triggered by the design flow. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Nov 2016.

[46] R. Kumar, P. Jovanovic, W. Burleson, and I. Polian. Parametric trojans for fault-injection attacks on cryptographic hardware. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 18–28, Sep. 2014.

[47] Wenchao Li, Adria Gascon, Pramod Subramanyan, Wei Yang Tan, Ashish Tiwari, Sharad Malik, Natarajan Shankar, and Sanjit A. Seshia. Wordrev: Finding word-level structures in a sea of bit-level gates. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 67–74, June 2013.

[48] MITRE. CVE-2009-1897: Linux Kernel NULL Pointer Dereference Privilege Escalation. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1897, June 2009. [Online; accessed 31-Jan-2022].

[49] Michael Muehlberghuber, Frank K. Gürkaynak, Thomas Korak, Philipp Dunst, and Michael Hutter. Red team vs. blue team hardware trojan analysis: Detection of a hardware trojan on an actual asic. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, New York, NY, USA, 2013. Association for Computing Machinery.

[50] K. Nagarajan, M. N. I. Khan, and S. Ghosh. Entt: A family of emerging nvm-based trojan triggers. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 51–60, May 2019.

[51] Adib Nahiyan, Mehdi Sadi, Rahul Vittal, Gustavo Contreras, Domenic Forte, and Mark Tehranipoor. Hardware trojan detection through information flow security verification. In *2017 IEEE International Test Conference (ITC)*, pages 1–10, 2017.

[52] R. Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, pages 69–70, June 2004.

[53] Aleph One. Smashing the Stack for Fun and Profit. *Phrack*, 7, November 1996. http://www.phrack.org/issues/49/14.html [Online; accessed 11-March-2019].

[54] Tiago Perez, Malik Imran, Pablo Vaz, and Samuel Pagliarini. Side-channel trojan insertion - a practical foundry-side attack via eco. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, May 2021.

[55] Shahed E. Quadir, Junlin Chen, Domenic Forte, Navid Asadizanjani, Sina Shahbazmohamadi, Lei Wang, John Chandy, and Mark Tehranipoor. A survey on chip to system reverse engineering. *J. Emerg. Technol. Comput. Syst.*, 13(1), April 2016.

[56] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information Systems Security (TISSEC)*, 15(1):2:1–2:34, March 2012.

[57] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1), mar 2012.

[58] O. Ruwase and M. Lam. A Practical Dynamic Buffer Overflow Detector. In *In Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pages 159–169, San Diego, CA, USA, 2004.

[59] Jonathan Salwan and Allan Wirth. Ropgadget: Gadgets finder and auto-roper, 2011. http://shell-storm.org/project/ROPgadget.

[60] Jeff Seibert, Hamed Okhravi, and Eric Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 54–65, New York, NY, USA, 2014. Association for Computing Machinery.

[61] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 552–561, New York, NY, USA, 2007. Association for Computing Machinery.

[62] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, page 1–8, New York, NY, USA, 2009. Association for Computing Machinery.

[63] Cynthia Sturton, Matthew Hicks, David Wagner, and Samuel T. King. Defeating uci: Building stealthy and malicious hardware. In *2011 IEEE Symposium on Security and Privacy*, pages 64–77, May 2011.

[64] Takeshi Sugawara, Daisuke Suzuki, Ryoichi Fujii, Shigeaki Tawa, Ryohei Hori, Mitsuru Shiozaki, and Takeshi Fujino. Reversing stealthy dopant-level circuits. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 112–126, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[65] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[66] The PaX Team. ASLR. http://pax.grsecurity.net/docs/aslr.txt. [Online; accessed 11-March-2019].

[67] M. Tehranipoor and F. Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE Design Test of Computers*, 27(1):10–25, Jan 2010.

[68] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the Expressiveness of Return-into-libc Attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)*, pages 121–141, Menlo Park, CA, 2011.

[69] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the Expressiveness of Return-into-libc Attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)*, pages 121–141, Menlo Park, CA, 2011.

[70] T. Trippel, K. G. Shin, K. B. Bush, and M. Hicks. ICAS: an extensible framework for estimating the susceptibility of ic layouts to additive trojans. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1742–1759, May 2020.

[71] Timothy Trippel, Kang G. Shin, Kevin B. Bush, and Matthew Hicks. Bomberman: Defining and defeating hardware ticking timebombs at design-time. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 970–986, May 2021.

[72] Perry Wagle, Crispin Cowan, et al. Stackguard: Simple stack smash protection for gcc. In *Proceedings of the GCC Developers Summit*, pages 243–255. Citeseer, 2003.

[73] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. Fanci: Identification of stealthy malicious logic using boolean functional analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 697–708, New York, NY, USA, 2013. ACM.

[74] Shengye Wan, Mingshen Sun, Kun Sun, Ning Zhang, and Xu He. Rustee: Developing memory-safe arm trustzone applications. In *Annual Computer Security Applications Conference*, ACSAC '20, page 442–453, New York, NY, USA, 2020. Association for Computing Machinery.

[75] Fish Wang and Yan Shoshitaishvili. Angr - the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9, Sep. 2017.

[76] Rafal Wojtczuk. The advanced return-into-lib (c) exploits: Pax case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*, 70, 2001.

[77] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor. Hardware trojans: Lessons learned after one decade of research. *ACM Trans. Des. Autom. Electron. Syst.*, 22(1), may 2016.

[78] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, New York, NY, USA, 2010. ACM.

[79] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester. A2: Analog malicious hardware. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 18–37, May 2016.

[80] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, May 2009.

[81] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. *SIGPLAN Not.*, 50(4):503–516, March 2015.

[82] J. Zhang, F. Yuan, L. Wei, Y. Liu, and Q. Xu. Veritrust: Verification for hardware trust. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(7):1148–1161, July 2015.

[83] Jie Zhang, Feng Yuan, and Qiang Xu. Detrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 153–166, New York, NY, USA, 2014. Association for Computing Machinery.

[84] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. End-to-end automated exploit generation for validating the security of processor designs. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 815–827. IEEE Press, 2018.

A Binary Analysis

Figure 12 illustrates the pattern-matching criteria searched for when analyzing binaries. As discussed in Section 9.5, we use a binary-analyzer to verify that exclusive use of the run-time check encoded trigger is unlikely to activate when executing non-bounds-checking code. Use of additional triggers and varying run-time check encodings may yield additional flexibility or precision for attackers attempting to hijack systems with more prevalent use of safe programming languages.

```
# Initial Trigger State
add    0xoffset(%rsp),%gp-reg
# Stage one
mov    %gp-reg, 0xoffset(%rsp)
# Stage two
cmp    $immediate,%gp-reg
# Deliver Payload
[...]
```

Figure 12: Pattern Matching Criteria for Binary Analysis

Table 5 lists the number of instruction sequences in built binaries for several benchmark applications from the Phoronix Test Suite. Besides the first row, the remaining applications are primarily written in C/C++. This experiment demonstrates that bounds-checks for Rust programs, as encoded in the run-time check encoded trigger are unlikely to falsely trigger on benign workloads.

Benchmark	**Matched Instruction Sequences**
Indexed-buffer victim	9
Nginx (2.0.0)	0
Apache (2.0.0)	1
Linux Kernel (5.4.0)	0
OS Bench (1.0.2)	0
OpenSSL (1.1.0)	0
Mcperf (1.1.0)	0
Memcached (1.6.9)	0
ipc-benchmark (1.0.0)	0
Leveldb (1.22)	0

Table 5: Observed Runtime-check Instruction Sequences