Fuzzilicon: A Post-Silicon Microcode-Guided x86 CPU Fuzzer

Johannes Lenzen
Technical University of Darmstadt
johannes.lenzen@stud.tu-darmstadt.de

Mohamadreza Rostami
Technical University of Darmstadt
mohamadreza.rostami@trust.tu-darmstadt.de

Lichao Wu
Technical University of Darmstadt
lichao.wu@trust.tu-darmstadt.de

Ahmad-Reza Sadeghi
Technical University of Darmstadt
ahmad.sadeghi@trust.tu-darmstadt.de

***Abstract*—Modern Central Processing Units (CPUs) are black boxes, proprietary, and increasingly characterized by sophisticated microarchitectural flaws that evade traditional analysis. While some of these critical vulnerabilities have been uncovered through cumbersome manual effort, building an automated and systematic vulnerability detection framework for real-world post-silicon processors remains a challenge.**

In this paper, we present Fuzzilicon, the first post-silicon fuzzing framework for real-world x86 CPUs that brings deep introspection into the microcode and microarchitectural layers. Fuzzilicon automates the discovery of vulnerabilities that were previously only detectable through extensive manual reverse engineering, and bridges the visibility gap by introducing microcode-level instrumentation. At the core of Fuzzilicon is a novel technique for extracting feedback directly from the processor's microarchitecture, enabled by reverse-engineering *Intel*'s proprietary microcode update interface. We develop a minimally intrusive instrumentation method and integrate it with a hypervisor-based fuzzing harness to enable precise, feedback-guided input generation, without access to Register Transfer Level (RTL) or vendor support.

Applied to *Intel*'s *Goldmont* microarchitecture, Fuzzilicon introduces 5 significant findings, including two previously unknown microcode-level speculative-execution vulnerabilities. Besides, the Fuzzilicon framework automatically rediscover the μSpectre class of vulnerabilities, which were detected manually in the previous work. Fuzzilicon reduces coverage collection overhead by up to $31\times$ compared to baseline techniques and achieves 16.27% unique microcode coverage of hookable locations, the first empirical baseline of its kind. As a practical, coverage-guided, and scalable approach to post-silicon fuzzing, Fuzzilicon establishes a new foundation to automate the discovery of complex CPU vulnerabilities.

I. Introduction

Computation and information processing form the backbone of modern society, and their security fundamentally depends on the trustworthiness of the underlying hardware. At the core of secure computing are CPUs, which are expected to faithfully implement their Instruction Set Architectures (ISAs) and enforce strict isolation between processes. However, this assumption has been increasingly challenged by the discovery of critical architectural and microarchitectural-level vulnerabilities [1]–[4]. These attacks demonstrate that flaws in CPU microarchitectural can be exploited to leak data, bypass protections, or undermine system integrity, even for secure and well-written software [5]. Indeed, modern processors, particularly in the x86 family, are highly complex, with layers of undocumented behavior implemented in proprietary μcode [6]. As designs become increasingly complex and opaque, the risk of hardware-level security flaws continues to grow [7].

To detect hardware-level vulnerabilities, researchers have traditionally relied on techniques such as formal verification [8]–[12], runtime detection [13], [14], information flow tracking [15]–[17], and hardware fuzzing [18], [19]. Among them, hardware fuzzing has emerged as a promising approach due to its scalability and adaptability to various designs [20]–[41]. Hardware fuzzing has evolved into two distinct approaches: pre-silicon fuzzing, which targets Register-Transfer Level (RTL) models during hardware development, and post-silicon fuzzing, which evaluates manufactured processors under real execution conditions [20]–[41]. While pre-silicon fuzzing is widely studied in literature thanks to the deep observability and fine-grained instrumentation within the RTL model [20], [21], [25]–[30], [33], [39]–[41], post-silicon fuzzing is rarely touched. The reason is straightforward: post-silicon fuzzers commonly target black-box or proprietary CPUs (e.g., from Intel and AMD) with visibility limited to architectural registers or crash symptoms [24]. Even worse, the internal microarchitectural state and μcode-level behavior, where many subtle bugs manifest [1], [2], [42], are largely inaccessible and undocumented. Existing hardware feedback mechanisms, such as performance counters or architectural registers, offer only coarse-grained or indirect insight. The lack of transparency and informative feedback prevents the evaluator from finding unexpected behaviors and tracing corresponding root causes.

Our Contribution. In this work, we present Fuzzilicon, the first post-silicon fuzzer for proprietary x86 CPUs with gray-

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.231486
www.ndss-symposium.org

box visibility. Fuzzilicon introduces a novel internal microarchitectural feedback channel to guide test generation. By running the CPU in a Red-unlocked mode [43] and leveraging undocumented debugging and instrumentation capabilities in Intel processors [44], we gain access to the μcode engine interface. We re-purpose this interface, which is typically used to deploy μcode patches, as a programmable introspection layer, inserting lightweight instrumentation directly into the processor. Through careful reverse engineering, we construct μcode patches that instrument internal μcode execution paths. This turns a proprietary CPU into a gray box, enabling observation of internal execution states (e.g., μcode path transitions) at runtime, without RTL access or specialized hardware. To ensure safe and deterministic execution of fuzzing workloads on the target CPU, we build a bare-metal, hypervisor-based fuzzing framework that isolates the device under test (DUT), controls its environment, and continuously monitors execution. We further introduce a serialization oracle that synthesizes semantically equivalent variants of instruction sequences, improving fuzzing reproducibility and enabling reliable detection of vulnerabilities and divergences across microarchitectural implementations. Together, these capabilities enable feedback-driven fuzzing of real, post-silicon x86 processors with microarchitectural visibility, uncovering rare execution paths and vulnerabilities.

Our contributions are listed as follows:

- We introduce Fuzzilicon, the first post-silicon x86 CPU fuzzer that leverages internal microarchitectural feedback by injecting runtime instrumentation using μcode patches, enabling introspection on proprietary silicon.
- We introduce a new μcode coverage feedback for post-silicon CPU fuzzing, enabling visibility into CPU's internal execution at the granularity of μcode operations.
- We design an optimized μcode instrumentation strategy that reduces patching overhead by 31 times compared to the baseline instrumentation [45].
- We demonstrate a novel μcode-level speculative execution fuzzing use case, enabling Fuzzilicon to detect μcode-level speculative vulnerabilities and reveal undocumented leakage paths.
- On *Intel N3350 (Goldmont)* CPU, Fuzzilicon uncovered 5 significant findings, including two previously unknown μcode-level speculative-execution vulnerabilities.
- The Fuzzilicon framework automatically rediscover the μSpectre class of vulnerabilities [42], which previously was detected manually.
- We extend prior reverse engineering of Intel *Goldmont*'s μcode patching infrastructure to enable, for the first time, custom instrumentation at arbitrary micro-op entry points.
- We design a low-overhead and bare metal-hypervisor for CPU fuzzing that safely isolates arbitrary x86 programs and preserves determinism and system stability even under malformed instruction sequences.
- We introduce a serialization oracle that generates semantically equivalent instruction sequences, enabling robust cross-platform divergence detection without any ground-truth architectural oracle.

The complete source code for the Fuzzilicon framework is open-sourced at https://github.com/0xCCF4/ufuzz and permanently publicly available at https://doi.org/10.5281/zenodo.17012971. For detailed instructions on setting up and using the framework, please refer to Appendix D.

The rest of this paper is organized as follows. Section II provides background on key concepts necessary for understanding Fuzzilicon, including x86 μcode execution, instruction decoding, *Red-Unlock* mode, and microarchitectural introspection techniques. Section III details the core technical challenges of applying coverage-guided fuzzing to commercial x86 CPUs. Section IV outlines the design of Fuzzilicon, while Section V describes its framework implementation, including μcode instrumentation and control infrastructure. Section VI evaluates Fuzzilicon's effectiveness in terms of discovered vulnerabilities, coverage, and performance. Section VII discusses the Fuzzilicon with more insights. Section VIII discusses related works. Section IX concludes this work.

II. Background

A. Microcode and Instruction Decoding

Intel x86 instructions range from simple arithmetic operations to highly complex instructions involving system state, memory ordering, or cryptographic operations. Implementing each *x86* instruction entirely in hardware would be infeasible due to silicon cost, complexity, and the need for update flexibility [6]. Instead, modern Intel CPUs use a μcode engine to decode complex instructions into sequences of simpler μoperations, which are then executed by the processor [46]. Each x86 instruction could be mapped to one or more μoperations. Simple instructions often decode statically into μoperations [6]; complex ones invoke the μcode engine [6]. The μcode engine expands each *x86* instruction into a sequence of μoperations, scheduled and executed under the structured control-flow primitive known as a *triad* [43], [45]. A *triad* contains three μoperations and a *sequence word* that controls static branching, ordering, and instruction termination.

Figure 1 presents a high-level view of the μcode system. The Read-Only Memory (ROM), ②, stores factory-installed instruction handlers, whereas the μcode Random Access Memory (RAM) stores runtime patches that manufacturers can apply software patches to even after the CPU has shipped. The CPU's μcode engine first checks the RAM for patches before executing the ROM default. A set of patch registers stores $(SRC_i \rightarrow DST_i)$ mappings, ③ when a μcode address SRC_i is fetched, execution is redirected to DST_i in RAM [45].

B. Red-Unlock Mode and Microarchitectural Access

Intel CPUs support multiple debug access levels [43], [45]: 1) *Green-locked:* default user/OS-visible mode on retail CPUs, all debugging features are disabled; 2) *Orange-unlocked:* Intended for Original Equipment Manufacturers (OEMs), enables a subset of debug features, and 3) *Red-unlocked:* Engineering mode used internally by Intel; enables unrestricted

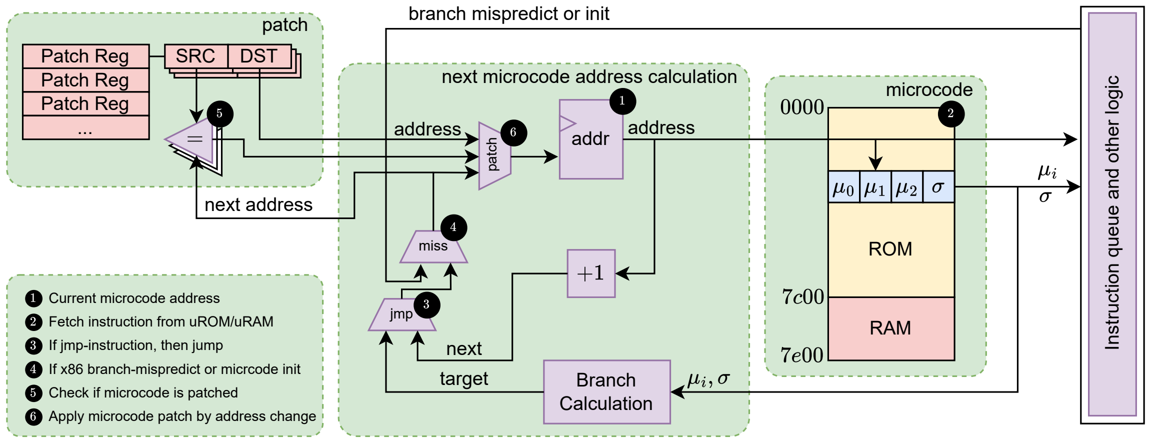
patch
Patch Reg
Patch Reg
Patch Reg
...
SRC
DST
=
branch mispredict or init
next microcode address calculation
address
patch
addr
address
microcode
0000
μ_0 μ_1 μ_2 σ
ROM
7c00
RAM
7e00
μ_i
σ
Instruction queue and other logic
next address
miss
+1
jmp
next
target
Branch Calculation
μ_i, σ
1 Current microcode address
2 Fetch instruction from uROM/uRAM
3 If jmp-instruction, then jump
4 If x86 branch-mispredict or mircrode init
5 Check if microcode is patched
6 Apply microcode patch by address change

Fig. 1. Simplified view of the Intel μcode engine [46]. Red components indicate runtime reconfigurable structures.

access to microarchitectural components. Specifically, in *Red-unlocked* mode, two undocumented new instructions become available: `udbgrd` and `udbgwr` [43]. These enable reading and writing internal CPU memory regions, including the μcode RAM. By writing a sequence of μcode *triads* and triggering execution at a target address, arbitrary μcode can be executed. This capability allows bypassing the normal and protected update procedure for μcode patches, i.e., first signature validation, then RAM writing, finally patch register configuration, permits direct manipulation of the currently deployed μcode update [45].

C. Fuzzing

Software fuzzing involves generating random and unexpected inputs to explore different execution path and uncover bugs [19], [47], [48]. Since exhaustive input enumeration is infeasible, fuzzers rely on heuristics, commonly guided by code coverage, to mutate inputs in ways that maximize program exploration [32]. Bugs are typically detected through observable runtime violations, such as crashes, memory access errors, or failed assertions.

Hardware fuzzing adapts these core ideas to the context of processor testing. Instead of application-level inputs, hardware fuzzers generate short programs, i.e., instruction sequences, that are executed directly on the target CPU. As in software fuzzing, heuristics are required to guide test generation. These may include RTL signal toggling in simulation-based setups or microarchitectural activity in post-silicon environments. However, detecting bugs in hardware poses unique challenges: CPUs are not considered “buggy” because they fault or access invalid memory regions or trigger exceptions. This is often correct behavior under certain conditions. As a result, post-silicon hardware fuzzing typically depends on two main strategies for bug detection: 1) Assertion checking, which is only feasible when RTL is available; and 2) Differential testing, which compares execution results across different CPUs or hardware configurations to identify inconsistencies [19].

Further, post-silicon fuzzing is constrained by restricted visibility into the CPU internal state of the processor, making it difficult to detect complex errors. In this work, we address this limitation by introducing custom instrumentation into the μcode layer. This allows us to collect CPU internal execution data at runtime, enabling a new class of post-silicon fuzzing that observes microarchitectural behavior.

III. Breaking the Post-Silicon Barrier

Despite dominating modern computing, commercial x86 CPUs have not been the target of coverage-guided fuzzing, a proven method for exposing complex software bugs. This gap is driven by four core challenges: *microarchitectural invisibility*, *absence of bug detection oracle*, *non-deterministic execution*, and *fault containment*. We detail each challenge in the following and explain how Fuzzilicon overcomes them.

Challenge 1: Microarchitectural Invisibility

x86 processors are complex and opaque, with undocumented μcode and speculative behaviors. This lack of visibility severely limits the feedback needed for the fuzzer.

Coverage-guided fuzzing depends on informative feedback to guide test generation. However, unlike open architectures such as RISC-V, where pre-silicon emulation allows fine-grained instrumentation, x86 processors are closed-source and proprietary; thus, valuable sources of feedback are limited. While prior work [22], [24] has used coarse-grained feedback, such as general-purpose register values or performance counters, these signals reveal only surface-level execution behavior. They offer limited guidance for exploring the deep microarchitectural behaviors that carry critical hardware vulnerabilities. Consequently, rich and informative feedback is lacking to drive effective fuzzing campaigns for *x86* processors.

Our Solution. Fuzzilicon introduces the first microarchitectural feedback mechanism for post-silicon fuzzing: *μcode-level coverage*. By instrumenting μoperations, Fuzzilicon collects fine-grained microarchitectural traces during execution. This coverage metric enables deep exploration of all CPU's microarchitectural behavior (μoperations) space without requiring RTL models. We detail the μcode-level coverage in Section V-B.

However, enabling μcode instrumentation is not straightforward. First, the μcode architecture is proprietary and undocumented. As a result, all public knowledge about Intel's μcode originates from reverse engineering, and is neither complete nor guaranteed to be correct. This makes deploying custom μcode updates inherently risky, as unsound patches may corrupt the internal state of the CPU. To overcome this, Fuzzilicon minimizes the state changes originating from μcode instrumentation, described in Section V-B. Second, prior work [45] records only whether a given μcode address executed, ignoring execution multiplicity. Consequently, two x86 instructions that invoke the same μoperations but with different iteration counts (e.g., once vs. four times) appear indistinguishable, and one may be discarded for not increasing coverage, despite exploring different microarchitectural behaviors. This coarse signal misses substantial exploration space and prunes valuable inputs. Fuzzilicon addresses this by counting executions per μcode handler, revealing loops, and deeply nested paths within a single test. On our target, *Intel Apollo Lake (Celeron, Goldmont) N3350 (CPUID[1].EAX=0x506ca)*, we repurpose 16 internal registers to track 32 μcode addresses concurrently. Given a $32k$ μcode address space, this requires $1k$ instrumentation points. However, we developed coverage-scheduling optimizations to keep the overhead practical (Section V-C).

Challenge 2: Absence of Bug Detection Oracle

Detecting x86 CPU bugs is difficult without formal microarchitectural specifications. ISA-level models miss undocumented/speculative behaviors, limiting bug detection.

Detecting incorrect behaviors in commercial x86 CPUs is inherently difficult due to the absence of formal microarchitectural specifications. While ISA-level simulators offer partial reference behavior, they fail to capture microarchitectural effects such as speculative execution, undocumented instructions, or μcode operations. As a result, it's often unclear whether observed differences reflect genuine vulnerabilities or benign implementation variations. In the absence of a reliable oracle, fuzzers rely on differential detection: input-driven, which requires known outputs (rare in fuzzing), and output-driven, which compares results across different CPU implementations. While general, output-driven methods suffer from 1) False Positives (FP), due to architectural differences, and 2) False Negatives (FN), when different CPUs exhibit the same flawed behavior. A few solutions have been proposed to address this challenge; for instance, Reversi [49] proposes generating a reverse instruction for each instruction. However, for complex *x86* instruction, automatic generation of such reverse instructions is challenging [1].
Our Solution. Inspired by Zenbleed [1], Fuzzilicon introduces *Serialization Oracles* (Section V-F). For each test case P, Fuzzilicon synthesizes a semantically equivalent variant Q by inserting serialization fences to suppress speculation and reordering. Divergences between P and Q potentially flag microarchitectural bugs; no reference model is required [1].

Challenge 3: Fault Containment

Applying custom μcode patches in the CPU is inherently risky: malformed sequences can corrupt architectural state or block forward progress, stalling or terminating the fuzzing campaign.

Since μcode documentation is proprietary and our knowledge originates from reverse engineering, deploying custom μcode sequences carries nontrivial risk. Exercising undocumented behaviors or injecting malformed μcode can induce hangs, crashes, or persistent/transient lockups requiring a hardware reset. Without proper fault containment, such failures crash the entire fuzzing infrastructure, resulting in lost execution state and requiring manual recovery.
Our Solution. Fuzzilicon decouples test execution from control logic using a two-part architecture. The *Fuzzer Agent*, running on the target CPU, is responsible for setting up the hypervisor, executing fuzzing test cases, applying μcode instrumentation, and collecting internal coverage feedback. The *Fuzzer Controller*, on a separate host, handles test case generation and mutation, coverage analysis, feedback-driven scheduling, and vulnerability triage. If the agent crashes or stalls, the controller resets the system and resumes testing. This architecture ensures that crashes in the system under test do not affect the fuzzer's control logic or analytics pipeline. Furthermore, this architecture enables scalable deployment across multiple agents. We detail our solution in Section IV.

Challenge 4: Non-deterministic Execution

Fuzzing depends on reproducible execution to analyze crashes, however stateful instructions, Operating System (OS) noise, and microarchitectural states introduce Non-determinism during CPU fuzzing.

For sound evaluation and triage, fuzzing results must be reproducible [50]. Achieving reproducibility on post-silicon x86 CPUs is challenging due to: stateful instructions (e.g., `RDRAND`, `RDTSC`, `RDPMC`), OS/system noise, and microarchitectural state all introduce nondeterminism. Moreover, fuzzing inputs can corrupt global state or leave persistent side effects that contaminate subsequent tests: unconstrained programs may modify control registers, I/O ports, or memory mappings.
Our Solution. Fuzzilicon executes each test case in a bare-metal hypervisor environment that provides strong isolation from the host system, described in Section IV-D. It resets memory and CPU state between tests, suppresses asynchronous events, and prevents residual side effects. This ensures repeatable execution even for fuzzing inputs, while preserving access to advanced CPU features like speculation,

virtual memory, and μcode engine.

IV. Design

In this section, we begin with a high-level overview of Fuzzilicon's design. We then introduce the μcode coverage metric and explain how it exposes microarchitectural execution for effective exploration. Next, we describe our low-overhead bare-metal hypervisor that isolates execution while supporting efficient μcode coverage collection. Finally, we present our *Serialization Oracle* for vulnerability detection and discuss how we address its practical challenges.

A. High-level Overview

The high-level overview of Fuzzilicon is shown in Figure 2, structured into three high-level components: *Fuzzer Controller*, *Fuzzer Agent*, and *Watchdog*. The *Fuzzer Controller* runs off-target CPU and orchestrates the fuzzing campaign: it maintains the test corpus, mutates test cases, generates serialized variants for each mutated input, and dispatches both the original and serialized tests to the *Fuzzer Agent*.

The *Fuzzer Agent* runs on the target CPU. It pulls test cases from the *Fuzzer controller*, applies μcode instrumentation that records μcode coverage into RAM, provisions isolated execution using Fuzzilicon's bare-metal hypervisor, and executes each test both as-is and as its serialized variant in separate Virtual Machines (VMs). After each run, the *Fuzzer Agent* collects the final architectural state (general-purpose and control registers) via the hypervisor interface and forwards it to the *Early Bug Evaluation* stage. If the final architectural state of two executions diverges, a tracer replays the testcase instruction-by-instruction, snapshotting post-instruction architectural state to localize the root cause. The suspicious trace is then sent to the *Fuzzer Controller* for triage and archival. In parallel with *Early Bug Detection*, the *Fuzzer Agent* exports post-execution μcode coverage to the controller to guide mutations and records it for final reporting.

Fuzzilicon decouples the fuzzer from the DUT. This separation ensures that unexpected target behavior (e.g., hangs, crashes) does not stall the campaign: all fuzzing state, collected coverage, and detected issues are durably stored. Upon detecting such failures, the *Fuzzer Controller* triggers the *Watchdog* that hard-resets the target CPU, power-cycles the *Fuzzer Agent*, and restores the target CPU to a known good state.

B. Microcode-Level Coverage Metric

Fuzzilicon introduces a novel feedback signal, μcode coverage, which enables internal visibility into instruction execution at the μoperations level. This feedback tracks which μcode addresses are executed during test case execution, enabling the fuzzer to probe complex microarchitectural behaviors that are invisible to traditional architectural feedback.

As outlined in Section II, modern x86 processors execute complex instructions by translating them into sequences of μoperations, stored in μcode ROM. To support post-silicon reconfiguration, CPUs also includes a writable patch RAM and a redirection mechanism known as the *hook table*. Upon instruction decode, the *hook table* determines whether to use the default μcode ROM or redirect execution to a patch stored in RAM. Fuzzilicon repurposes this reconfiguration mechanism to inject lightweight probes into the CPU's μcode execution path. When an instruction executes, each associated μoperation triggers a lightweight logging operation that increments a counter located in physical memory (RAM). These counters are indexed by hook index, which can be later mapped to μcode addresses, producing a precise execution profile that records which μoperation is executed and the corresponding frequency. The μcode coverage serves as a fine-grained feedback signal, enabling Fuzzilicon to prioritize inputs that explore new microarchitectural behaviors. To ensure that μcode instrumentation does not affect CPU functionality (Challenge 2 in Section III) Fuzzilicon preserves architectural state and control flow, and leverages unused or non-critical μcode registers to avoid interfering with the instruction's intended behavior.

At runtime, the *Fuzzer Agent* installs instrumentation by writing patched triads into μcode RAM and configuring the *hook table* before execution; upon completion, it retrieves coverage from a memory-mapped region and reports it to the *Fuzzer Controller*. By integrating with the processor's native μcode patch infrastructure, Fuzzilicon provides microarchitectural introspection without emulation, turning the CPU into a gray-box fuzzing target with microarchitectural visibility.

C. Why Microcode Coverage?

Architectural feedback, e.g., register differences and instruction coverage, observes only committed instruction outcomes and overlooks semantically distinct microarchitectural behaviors. A single x86 opcode, which is implemented by a μcode routine and contains guarded control-flow (condition on *VMX*/*SMM* state, availability/health of internal units, retry/loop paths, fallbacks), can traverse different paths depending on microarchitectural state. For example, as illustrated by the μcode pseudo code for `RDRAND` in Algorithm 1. Lines 1, 2, and 5 gate behavior on these conditions: the same instruction may return a hardware random value, trap to *SMM*, raise #UD, or exit to a hypervisor while executing an identical instruction, yet architectural coverage reports "covered" after the first observation and provides no signal about which internal condition remains unexplored. In fuzzing terms, relying on ISA-level outcomes is akin to function-coverage in software fuzzing [51]: it saturates early and fails to separate interesting edges inside the implementation.

We introduce the μcode coverage to address this visibility gap (Challenge 1 in Section III) by exposing path-sensitive signals from the μcode engine. By logging which μoperation execute and with what multiplicity, it distinguishes inputs that exercise different internal paths for the same instruction. This yields a metric that guides the fuzzer to prioritize seeds that unlock new μcode edges or change execution counts. In practice, μcode coverage plays the role that edge coverage plays in software fuzzing [51]: it provides the fine-grained,

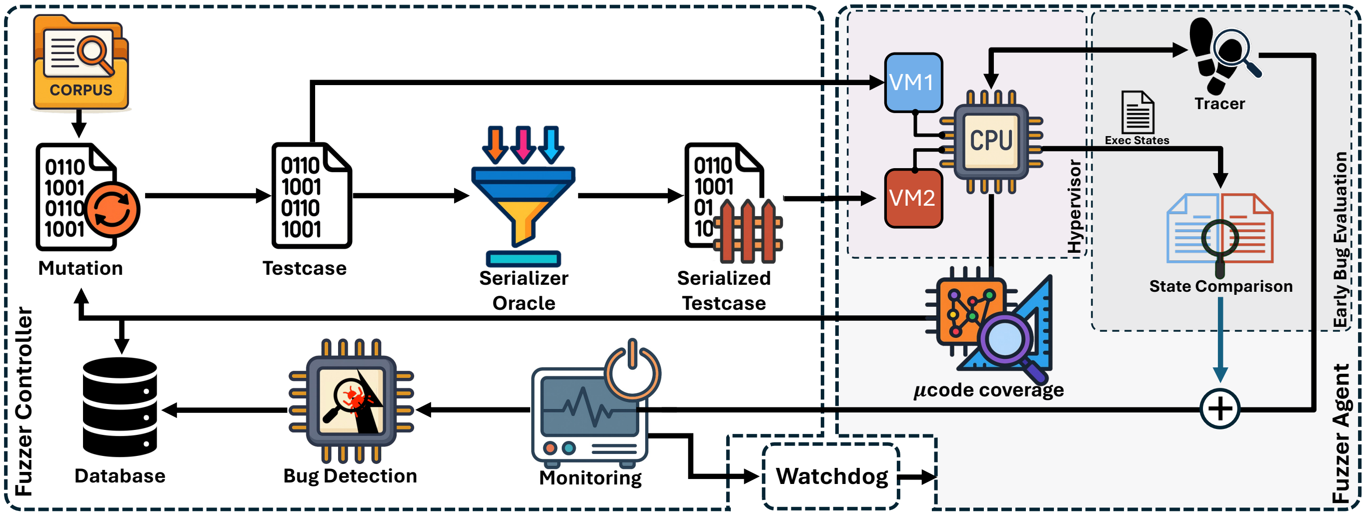

Fig. 2. High-level overview of Fuzzilicon architecture

state-sensitive feedback necessary to explore the CPU's microarchitectural functionality.

Algorithm 1 Pseudocode summary of the `RDRAND` μcode behavior.

```
 1: if not in SMM then
 2:     if LOCK prefix used then
 3:         RAISE(#UD)
 4:     end if
 5:     if VMX then
 6:         EXIT TO SMM OR VMX MONITOR
 7:     end if
 8: end if
 9: $number \leftarrow$ HARDWARERNGGEN()
10: $dst \leftarrow$ ZEROEXTEND($number$)
11: EFLAGS.CF $\leftarrow$ SET IF $number \neq 0$
```

D. Hypervisor-Based Execution Isolation

To ensure deterministic execution and full isolation of fuzzing inputs (Challenge 4 in Section III) Fuzzilicon introduces a lightweight, custom type-1 hypervisor [52] that encapsulates each test case within a dedicated virtual CPU instance. Unlike traditional virtualization systems designed for long-lived guest operating systems, Fuzzilicon's hypervisor is optimized for single-purpose, short-lived fuzzing executions with strict guarantees on state control and containment. For every test case, the fuzzer initializes a new virtual CPU configured with a fixed architectural state. This includes the general-purpose registers, control and system registers, the instruction pointer, and segment selectors. The goal is to ensure that all inputs begin execution from an identical and defined CPU state, eliminating residual effects from prior tests.

To enforce memory isolation, the hypervisor exposes a restricted, virtualized memory layout to the VM using hardware-assisted paging. Control over memory permissions ensures that critical configuration structures remain immutable and that fuzzing inputs cannot overwrite or reuse memory in unintended ways. The hypervisor is responsible for intercepting instructions that may affect host system configuration, access IO ports, result in non-deterministic execution, and intercept non-maskable interrupts. To prevent non-termination, the hypervisor enforces a maximum execution time. Tests that exceed their budget are preempted with a forced VM exit, ensuring that infinite loops cannot degrade the throughput or availability of the fuzzing campaign. After the VM terminates, the hypervisor extracts the architectural state of the virtual CPU for bug detection. The hypervisor is tightly integrated into Fuzzilicon with low overhead, detailed in Section V-D.

E. Differential Testing with Serialization Oracle

To detect bugs and vulnerabilities without a formal reference model, Fuzzilicon employs a differential strategy, inspired by the Zenbleed [1], that compares each input program P with a serialized variant Q, constructed to be semantically equivalent. The central idea is that if a processor is functionally correct, P and Q should produce the same architectural outcome. Any deviation implies a hidden inconsistency in microarchitectural behavior. The transformation that produces Q enforces instruction-level serialization by inserting *fence instructions* between every instruction to suppress speculation and reordering.

However, this transformation alters code layout, breaks relative addressing, and disrupts control and data flow. To overcome these challenges, Fuzzilicon incorporates systematic transformation strategies that preserve instruction semantics despite structural modification. For example, instruction pointer relative immediate memory accesses are adjusted by changing the opcode's operand to access the same memory location, like in the original program.

Additionally, Fuzzilicon detects misaligned or non-standard control transfers, such as jumps into instruction bodies. In these cases, the program is unrolled into isolated fragments that can be serialized independently and safely joined (see Appendix B). This Serialization Oracle lets Fuzzilicon expose bugs rooted in microarchitectural features, such as speculation, transient state retention, hidden instruction, without a reference model (Challenge 2, Section III). The implementation of this Serialization Oracle is explained in Section V-F.

V. Implementation

In this section, we detail the implementation of the Fuzzilicon framework. We first present the system architecture and deployment approach, followed by our μcode instrumentation technique. We then describe our patching and coverage collection strategy, the hypervisor-based execution environment, our fuzzing input generation and mutation methods, and finally our differential testing approach with serialization oracle.

A. System Architecture and Deployment

As outlined in Section IV, Fuzzilicon is comprised of three components: the *Fuzzer Controller*, *Fuzzer Agent*, and *Watchdog*, implemented in Rust and x86-64 assembly. For target CPU, we utilized a Gigabyte *GB-BPCE-3350C* [53] Mini-PC containing an *Intel N3350* (*Goldmont* microarchitectural, *CPUID* `0x506ca`). The target CPU does not natively expose interfaces for custom μcode patches. We first red-unlock the processor following established procedures for this processor family [43], [45], [54]–[56]. Upon successful red-unlocking, undocumented instructions become available that enable Control Register Bus (CRBUS) access (`udbgrd` and `udbgwr`), which are essential for μcode manipulation during fuzzing. This is a one-time manual procedure per target device.

Both the *Fuzzer Controller* and *Watchdog* are hosted Rust applications running on Raspberry Pi 4 platforms with *NixOS* [57]. The *Fuzzer Controller* serves as the stateful core of the Fuzzilicon framework, generating test cases, performing mutations, communicating with the *Fuzzer Agent* to deploy tests and collect μcode coverage, and coordinating with the *Watchdog* to manage the target CPU. The *Watchdog* functions as a remote keyboard and storage device for the *Fuzzer Agent*, responsible for hard-resetting the target, booting it to Unified Extensible Firmware Interface (UEFI), and starting the *Fuzzer Agent* executable. It exposes a Representational State Transfer (REST) Application Programming Interface (API) that allows the *Fuzzer Controller* to issue control commands, for instance, upon detecting communication timeouts or unresponsive behavior, the *Fuzzer Controller* triggers a reset command that causes the *Watchdog* to physically cycle power via probes soldered to the target motherboard. The *Fuzzer Agent* deploys as a standalone UEFI application on the target CPU to eliminate OS interference. It injects μcode instrumentation and custom patches, executing fuzzing test inputs, collecting coverage, and communicating with the *Fuzzer Controller* over User Datagram Protocol (UDP) for task coordination, test input delivery, and μcode coverage collection.

This modular hardware layout ensures that the *Fuzzer Controller* and *Watchdog* remain unaffected by crashes or lockups in the target CPU, enabling long-term autonomous operation over days or weeks.

B. μcode Instrumentation and Patching

Fuzzilicon enables runtime instrumentation of μoperation by deploying patched μcode sequences and configuring μcode *hook table*. As outlined in Section IV, the goal is to guide the fuzzer toward exploring all possible paths within μcode implementations-effectively covering all reachable μcode addresses. To achieve this, we instrument the target μcode addresses by adding entries to the μcode *hook table* that redirect execution to coverage collection logic, then resume from the original address upon completion.

To preserve target CPU functionality, coverage collection minimizes state changes. Our instrumentation first saves a required subset of the general-purpose registers in the CPU's staging buffer [45], records the coverage event and updates the in-memory coverage map, restores the saved state, executes the overwritten μoperation, and finally jumps to the originally intended next μcode address to continue normal execution. To the best of our knowledge, the `write-to-staging-buffer` is the only known μoperation capable of storing register values into memory without needing an additional register operand. Since the staging buffer is shared among all μcode routines, Fuzzilicon selects an address location normally used by the `udbgwr` instruction, ensuring a minimal state impact [43], [45]. This approach ensures that the target's architectural and microarchitectural behavior remains unaffected.

Since the instrumentation only uses a subset of the available general-purpose registers, we only save and restore the state of these registers, keeping the instrumentation overhead profile low while maintaining correctness.

Our reverse engineering analysis uncovered that the μcode engine implements address hook redirection through a paired addressing mechanism. Specifically, when a redirection hook is configured for an even address S to destination D, the μcode engine automatically redirects the corresponding odd address $S+1$ to destination $D+1$ without requiring an explicit hook entry. Consequently, determining whether the hook was triggered by an even or odd μcode address becomes critical for proper execution flow control and coverage calculation. To handle this differentiation, Fuzzilicon places an immediate jump at the hook redirect destination D, to ensure that μcode execution originating from address S will execute this instruction, while execution triggered from $S+1$ will skip it and continue to subsequent instructions. This mechanism enables Fuzzilicon to accurately differentiate and log the specific entry address (S or $S+1$) that triggered the hook.

Since applying a μcode hook redirection overwrites the processor's native μcode at addresses S, $S+1$, Fuzzilicon manually inlines the corresponding original μcode instructions at the conclusion of the injected sequence. Following this restoration, an immediate jump instruction is inserted to resume execution at the appropriate continuation address ($S+1$,

$S+2$, or jump target if the original μoperation was a jumping instruction).

For instrumenting a new μcode address to enable coverage collection, both the inlined original μcode instructions and the jump instruction must be updated accordingly. Given that each of the 16 μcode hook registers maps to a pair of addresses (even and odd), complete reconfiguration necessitates writing 144 distinct values[1] through the `udbgwr` instruction. To minimize the computational overhead associated with this setup process, Fuzzilicon implements a custom μcode utility function that accepts a memory address parameter and performs the setup of all active *hook table* entries within a single instruction decode cycle. Consequently, reconfiguring all hook targets requires only a single `udbgwr` invocation per test case iteration, substantially reducing the instrumentation overhead and improving overall fuzzing efficiency.

C. Coverage Collection Strategy

Our target platform is constrained to only 16 μcode hooks, which limits the instrumentation coverage to 32 addresses per instrumentation round. Given the approximately $32k$ (exact number `0x7C00`) μcode address space, collecting comprehensive coverage for all μcode addresses within a single fuzzing test case requires Fuzzilicon to execute the test case $\frac{0x7C00}{32} = 992$ times. This baseline approach introduces significant performance overhead.

To address this fundamental limitation, Fuzzilicon implements the following optimized coverage collection scheduling mechanism that significantly reduces execution overhead while maintaining comprehensive coverage collection capabilities. During the initial fuzzing phase, Fuzzilicon instruments all x86 instruction entry points residing below the `0x1000` address range [43]. However, these instruction entry addresses are aligned to 8 addresses, resulting in a total of $\frac{\text{0x1000}}{8} = 512$ entry locations. Using 16 hook registers, each fuzzing input is executed $\frac{512}{16} = 32$ times to achieve initial coverage.

Subsequently, static analysis and μcode disassembly techniques can be used to extract basic blocks from μcode. A basic block is defined as a set of μoperations that once the first μoperation of the block executes, unconditionally execute till the last μoperation. Further, the only entry point is the first μoperation, while the last μoperation is always the last executed instruction of the basic block [58].

When instrumenting one μcode address of any basic block, the coverage can be propagated to all addresses that are part of the block. Furthermore, the set $\Phi(y)$ captures all conditionally reachable successors of a basic block y. Only these successors within $\Phi(y)$ are selected for subsequent instrumentation and test re-execution when the coverage of y is non-empty. This selective approach eliminates unnecessary iterations while ensuring complete and fine-grained coverage collection.

Overall, this optimization strategy significantly reduces the computational overhead from 992 times re-executions required in the baseline approach to at best 32 iterations, plus the number of conditional branches encountered during μcode execution, which benefits us with at best $31\times$ less overhead.

D. Hypervisor-Based Program Execution

We implemented a type-1 hypervisor, running on top of UEFI, providing fuzzing input isolation using the *Intel VMX* virtualization technology. We provide a reproducible fuzzing execution harness by initializing a new virtual CPU instance, memory isolation, and executing controls. First, we instantiate a fresh virtual CPU by initializing the Virtual Machine Control Structure (VMCS), which is the main configuration structure for *Intel VMX*. We set the initial values of stack and program counter registers, set up VM-execution control setting [59], and set the memory translation-related registers. Using *Extended Page Tables* [60], we provide the same virtualized view of main memory to each fuzzing input, split into three parts: execute-only, read-write, read-only, initialized once. Before invoking the fuzzing input (in an execute-only region), the read-write region is zeroed to purge transient state from prior executions. The read-only region is only initialized once at fuzzing start, containing required memory structures for x86 instruction execution, like a Global Descriptor Table (GDT), and a Task State Segment (TSS) data structure. Using the VM-execution control settings, we configure the fuzzing guest VM to exit on problematic actions like executing an instruction that introduces nondeterminism, interaction with hardware components, arrival of an external interrupt, and the VM's timeout budget. The hypervisor may handle each event and decide to stop or continue fuzzing input execution. Using the timeout, we prevent loops in the fuzzing input from stalling the fuzzing process. When finishing executing a fuzzing sample, the architectural state of the virtual CPU instance is captured and sent to the fuzzing controller into the bug-detection pipeline.

E. Fuzzing Input Generation and Mutation

Each fuzzing campaign starts with an initial corpus, comprising randomly generated byte sequences or valid instruction gadgets extracted from software libraries, such as *libcxx*. Valid instructions are extracted from software libraries and randomly concatenated until a configurable size threshold is achieved.

Input mutation is performed using either a custom genetic algorithm guided by coverage and execution-depth heuristics, or established mutators (e.g., Havoc) from the *libafl* library [48]. Our custom mutation engine implements random 1-to-8 byte mutations and cross-over operations on the fittest fuzzing samples. The fitness evaluation of a given sample is computed using the μcode coverage metric, combined with the ratio of executed bytes to total generated bytes. We preserve the top-k samples per generation and utilize them to seed subsequent generations. Consequently, the generated fuzzing inputs may comprise valid ISA instruction sequences or arbitrary invalid byte sequences, all evolved through μcode coverage feedback mechanisms

[1] $16 \times (2 \times 4 + 1)$ Two triads and the *hook table* entry for each *hook table* entry.

F. Differential Testing with Serialization Oracle

To implement the serialization oracle discussed in Section IV-E, we encounter a non-trivial challenge: program layout changes cause relative addressing instructions to reference incorrect locations. We address this by analyzing each instruction's behavior according to the modified program layout. Our oracle maintains a list of all relative addressing instructions, extracts these from the original test case, and recomputes their offsets for the transformed program, following established binary rewriting methodologies [61]. However, fuzzing inputs are comprised of randomly generated byte sequences rather than compiler-generated code, enabling jump or control flow instructions to target mid-instruction locations (Appendix B). Introducing serialization in such cases creates non-equivalent execution between serialized and non-serialized variants, generating numerous false positives during vulnerability detection. We solve this problem by adapting Superset Disassembly [62] techniques, unrolling the original program to discover additional instruction gadgets. However, we utilize a different relocation approach: rather than relocating entire blocks, we relocate individual instructions to new locations, inserting fence instructions between each relocated instruction to ensure deterministic execution ordering.

A corner case involves self-modifying code and fuzzing test cases that attempt to alter their own instructions during execution. Since self-modifying code dynamically changes program execution flow at runtime, it can introduce new relative addressing instructions that static binary rewriting cannot anticipate or handle, as rewriting occurs before program execution. We address this limitation by restricting self-modifying behavior in fuzzing test cases through execute-only page permissions. The hypervisor detects and interrupts any fuzzing input that attempts to write instructions to execute-only memory pages, preventing dynamic code modification that would compromise our serialization oracle's correctness.

VI. Evaluation

In this section, we first discuss Fuzzilicon's automatic detection of μSpectre [42] class vulnerabilities. We then present a specialized use case where Fuzzilicon targets speculative execution leakage at the μcode level, leading to two newly discovered vulnerabilities and three new and interesting findings related to speculative behavior in μcode. Next, we evaluate μcode coverage effectiveness by comparing total achieved coverage, coverage acquisition speed, and the impact of random versus carefully crafted seeds and mutation engines. Since existing works [24] lack clear coverage definitions, direct comparison is not feasible; instead, we compare against a baseline x86 fuzzer without coverage guidance to demonstrate Fuzzilicon's advantages over the class of existing approaches. Each coverage experiment runs for 48 hours with at least three repetitions to minimize noise. Finally, we analyze Fuzzilicon's performance characteristics and μcode coverage overhead.

A. F1. μSpectre Vulnerabilities Detection

The μSpectre vulnerability class, discovered by Mosier et al. [42], exploits μcode-level speculative execution where branches are statically predicted (taken, not-taken, or stalled). This design choice enables data leakage through mispredicted branches that continue speculative execution across *x86* instruction boundaries, allowing subsequent instructions to leak information. During our fuzzing campaign, Fuzzilicon automatically detected this vulnerability class during μcode instrumentation (**F1**). The μcode instrumentation itself does not modify the test case's instruction sequence. However, the act of instrumenting μcode can trigger speculative execution behaviors at μcode-level that wouldn't occur naturally, for instance, by having μcode-level branches that are statically predicted. When the same test case is re-executed with serialization fences (such as `LFENCE` instructions), these fences prevent speculation from reaching the following instruction. The divergence in architectural state between the speculative and non-speculative executions reveals the speculative behavior, allowing Fuzzilicon to automatically identify instances of the μSpectre vulnerability class without prior knowledge.

B. Use Case: Fuzzing x86 Microcode for Speculative Leakage

Speculative execution enables processors to transiently execute instructions (μoperation) before preceding control flow is resolved. While effective for performance, this behavior creates the potential for sensitive information to leak through microarchitectural side effects [1], [3], [4], [63]. Prior work has largely focused on instruction-level speculation behaviors [1], [64], [65]. Fuzzilicon opens a new frontier: fuzzing the μcode sequences that execute transiently within speculative windows to discover information leakage or side channels originating at the μcode level. Rather than modifying the branch predictor behavior directly, we built a reusable μcode template (Listing 1) that creates a speculative execution context. We leverage an existing μoperation, `UJMPCC_DIRECT_NOTTAKEN_CONDNZ` (Opcode: `0x151`) which is designed to always predict "not taken" and initiate speculative execution at the μcode level [42]. We use this created speculative path as a stable speculative entry point for μcode fuzzing.

Speculative Template Construction: We reserve a region of the μcode patch RAM as the speculative body (see Listing 1). During each fuzzing iteration, Fuzzilicon injects a randomized or guided sequence of μoperations into this region. These μoperations execute speculatively because, as discussed, `UJMPCC_DIRECT_NOTTAKEN_CONDNZ` always will be mispredicted and create a speculative window. These μoperations cause microarchitectural effects that should be rolled back by design upon the end of the speculative window.

Leakage Observation: Although the speculative path does not commit architecturally, its execution may leave detectable architectural and microarchitectural traces. Fuzzilicon detects such leakage by comparing the architectural state before and after execution of the template (see Listing 1). This enables the detection of persistent side effects introduced solely by speculative μoperations execution.

This use case demonstrates how the Fuzzilicon framework enables targeted fuzzing of speculative behavior at the μcode level, uncovering security-relevant leakages that are invisible to architectural-level fuzzers. Below, we summarize key findings made possible through this capability.

```
<entry>
tmp2 := ZEROEXT_DSZ64(0xabab)
tmp0 := ZEROEXT_DSZ64(0x1000)
tmp1 := LDPPHYS_DSZ32_ASZ16_SC1(tmp0)
tmp0 := SUB_DSZ64(tmp0, tmp1)

; Speculative branch | misprediction forced
UJMPCC_DIRECT_NOTTAKEN_CONDNZ(tmp0, <taken>)
; Speculative Window Start
; --- INSERT MICRO-OPERATION HERE ---
rax := ZEROEXT_DSZ64(0xdead)
NOPB
NOP SEQW SYNCFULL
NOPB
; Speculative Window Ends

<taken>
unk_256() !m1 SEQW LFNCEWAIT, UEND0
```

Listing 1. Speculative μcode window using forced misprediction template

F2. Speculative writes to the CRBUS persist after the speculative window is discarded.

We discovered that certain μoperations writing to the CRBUS within speculative execution windows persist their side effects even when the CPU subsequently discards the speculated execution. This leads to incorrect or unintended state changes that violate speculative execution's rollback guarantees. For example, executing micro-op `0x4292180220 MOVETOCREG_DSZ64(rax, 0x692)` writes to CRBUS address `0x692`, which controls the μcode *hook table* activation. When `rax` contains `0x1` (deactivate), all μcode patched, including security patches, are disabled, effectively bypassing current μcode security updates, even when executed speculatively and subsequently rolled back. We provide a comprehensive list (See Appendix C) of μcode operations whose effects persist during speculative windows despite execution rollback by triggering unrecoverable CPU lockups that constitute denial-of-service conditions. We categorize their behaviors as either reliable (StableTimeout), always causing a lock-up or unreliable (Unstable), sometimes causing a denial-of-service.

F3. Speculative updates to the segment selector caches persist after the speculative window is discarded. Our analysis reveals a critical vulnerability in segment selector cache management during μcode-level speculative execution. When a μoperation targeting segment selector caches (opcode `0xc6b - WRSEGFLD`) executes within a speculative window, their state modifications persist even when the triggering branch is subsequently mispredicted and the speculative execution should be discarded. This vulnerability can be exploited when default μcode implementations or security updates contain segment selector cache writes that execute speculatively but fail to undergo proper rollback following branch mispredictions. Critically, these persistent writes can bypass permission checks that would normally guard μoperation execution. An adversary capable of executing code on the target processor can exploit this behavior across all privilege levels, potentially achieving denial-of-service conditions, privilege escalation to ring-0, or unauthorized memory access. We demonstrate this vulnerability using μoperation `0xc6b26000037 WRSEGFLD(tmp7, GDT, BASE)` within our speculative execution template (Listing 1). This operation writes the `tmp7` register value into the segment selector cache as the GDT base address. The unauthorized modification can be detected through two mechanisms: indirectly via CPU crashes when the target memory contains invalid GDT structures, or directly through the Intel *VMX* hypervisor API that permits segment cache inspection and manipulation in virtualized environments.

F4. Microcode-implemented instructions terminate speculation. Our analysis shows that speculative execution at the instruction level terminates when encountering instructions requiring μcode implementation and the instruction cache is empty. Specifically, when a speculative window initiates, for example, following a mispredicted call-return sequence, the first instruction dispatched to a μcode sequencer halts further speculative execution. This behavior indicates that μcode dispatch boundaries function as implicit speculation barriers within *Intel*'s processor implementation. This architectural characteristic creates a timing-based observable that adversaries can exploit for information disclosure. An attacker can leverage these timing variations to infer sensitive information about program execution paths.

F5. Speculative μops leave traces on performance counters. Our analysis shows that certain μoperations executed speculatively produce measurable side effects on architecturally visible performance counters. For instance, speculative execution of the undocumented `UNK_256` μoperation increments the μcode sequencer counter (`MS_DECODE.MS_ENTRY`) even when executed within speculative windows that are subsequently discarded. This behavior indicates that non-retired μcode execution paths can leak internal processor state through architecturally observable side-channel artifacts. *Intel*'s ISA documentation reveals that this behavior aligns with documented processor specifications, although it can leak information.

Overall, these findings demonstrate the effectiveness of Fuzzilicon for exploring speculative behavior at the μcode-level, and its capability to uncover information leakage channels and safety violations not observable through conventional fuzzing techniques.

C. Coverage Analysis

In this section, we present the results on the effectiveness of μcode coverage feedback for exploring the CPU architecture. During this analysis, we fixed the mutation engine on the AFL Havoc engine due to its superior performance in all experiments compared to our custom mutation. Also, we

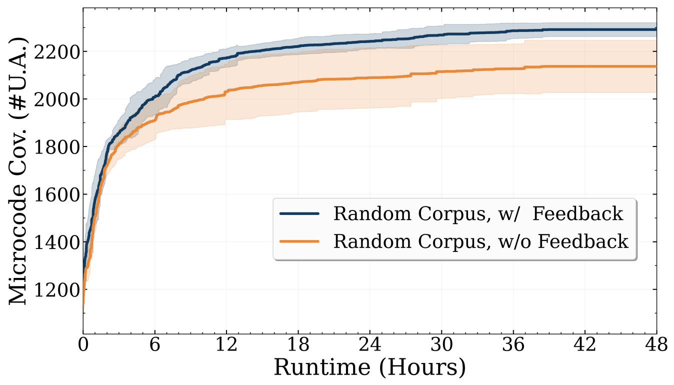

Fig. 3. Effectiveness of μcode-coverage feedback on exploration with Havoc mutators and random corpus over time. #U.A. stands for number of unique μcode addresses.

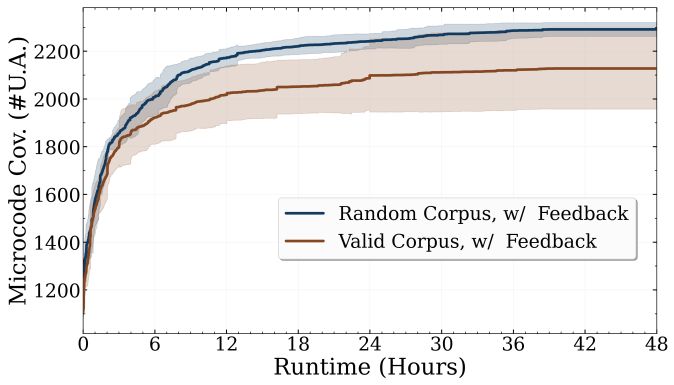

Fig. 4. Effectiveness of corpus on exploration with Havoc mutator and μcode-coverage feedback over time. (#U.A. stands for number of unique addresses.)

evaluated the impact of fuzzing corpus selection by conducting experiments using two distinct corpus types: (1) **Random Corpus**, consisting of concatenated random bytes up to a fixed size limit, and (2) **Valid Corpus**, constructed by extracting unique *x86* instructions from three widely-used software libraries: `libc++`, `libz`, and `libzip`.

Figure 3 shows the μcode coverage metric over time when using random corpus in both fuzzing setups, with μcode coverage feedback enabled in one configuration and disabled in the other. The fuzzing setup without feedback (`Random Corpus, w/o feedback`) represents the baseline *x86* fuzzers that lack coverage metrics , while the setup with feedback (`Random Corpus, w/ Feedback`) represents Fuzzilicon. As illustrated in the graph, Fuzzilicon outperforms baseline *x86* fuzzing methodologies not only in final coverage achievement (across all runs) but also achieves the same coverage metric approximately $8\times$ faster on average. This demonstrates the effectiveness of μcode coverage feedback in guiding the fuzzer toward exploring the CPU design. To further evaluate the impact of corpus quality, we conducted experiments using both random corpus and valid corpus while maintaining feedback activation in both scenarios. As shown in Figure 4, average coverage achieved by *Random Corpus* consistently outperforms *Valid Corpus*, which our analysis attributes to the inherent diversity of random corpora generated from random bytes, whereas the valid corpora contain structured instructions that may exhibit redundancy across different corpus samples.

However, Figures 3 and 4 only present the average number of unique μcode addresses achieved in each fuzzing setup, lacking information about the total overlap between these addresses and the comparative effectiveness when combining all runs. To evaluate this metric comprehensively, we calculated the μcode-coverage overlap matrix (Figure 5), μcode-coverage uniqueness matrix (Figure 6), and exclusive μcode-coverage analysis (Figure 7) across all runs for different setups.

Figure 5 presents the μcode-coverage overlap matrix, where diagonal cells indicate the total number of unique μcode addresses achieved across all runs for each fuzzing setup, while off-diagonal cells represent the overlap between row and column configurations. As evident in Figure 5, *Valid Corpus with Feedback* achieved $2,528$ unique addresses in total, surpassing all other configurations. Furthermore, configurations with μcode-coverage feedback (`Valid Corpus, w/ Feedback` and `Random Corpus, w/ Feedback`) consistently outperform their respective counterparts without feedback (`Valid Corpus, w/o Feedback` and `Random Corpus, w/o Feedback`), demonstrating the effectiveness of our introduced μcode-coverage feedback in exploring more unique μcode addresses. Additionally, this analysis reveals that *Valid Corpus* configurations achieve higher total unique address coverage across multiple fuzzing rounds (accounting for fuzzing reset effects [66]) compared to *Random Corpus* configurations. This finding reconciles the apparent contradiction in Figure 4, where *Random Corpus* shows better average performance per run, but *Valid Corpus* configurations achieve higher total coverage for all fuzzing runs.

Figure 6 shows the μcode-coverage uniqueness matrix, where each matrix cell represents the number of unique μcode addresses covered by the row fuzzing setup but not by the column setup. This visualization supports the same conclusions as the overlap matrix: fuzzing setups with μcode-coverage feedback consistently achieve more unique μcode addresses compared to their counterparts without feedback. Moreover, fuzzing setups with *Valid Corpus* consistently achieve more unique μcode addresses than *Random Corpus* configurations, supporting our previous conclusions.

Finally, figure 7 illustrates the number of exclusive coverage points achieved by each fuzzing configuration compared to all other setups, specifically, how many unique μcode addresses a fuzzing setup discovered that other setups failed to achieve. As demonstrated, configurations with μcode-coverage feedback enabled consistently achieve at least $2\times$ higher exclusive μcode-coverage points compared to any setup without μcode-coverage feedback. This provides strong evidence for the effectiveness of μcode-coverage feedback and Fuzzilicon in exploring the CPU microarchitectural compared to existing

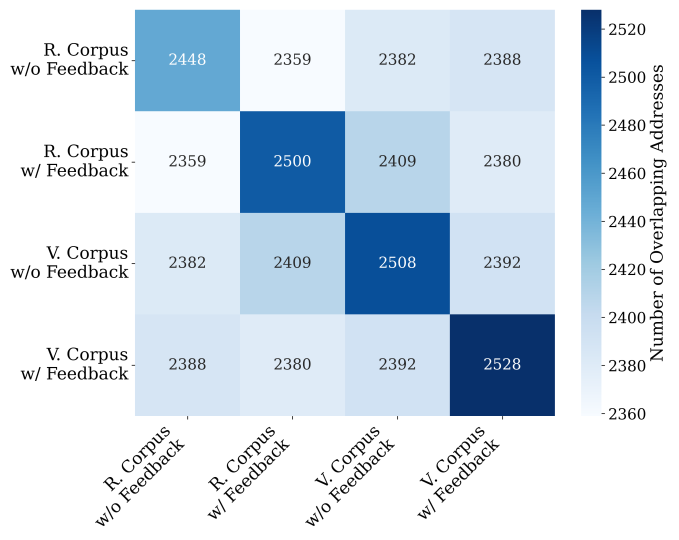

Fig. 5. μcode coverage overlap matrix. Addresses in row config overlapping with column config. `#U.A.` stands for number of unique addresses. `R.` stands for random, and `V.` stands for valid.

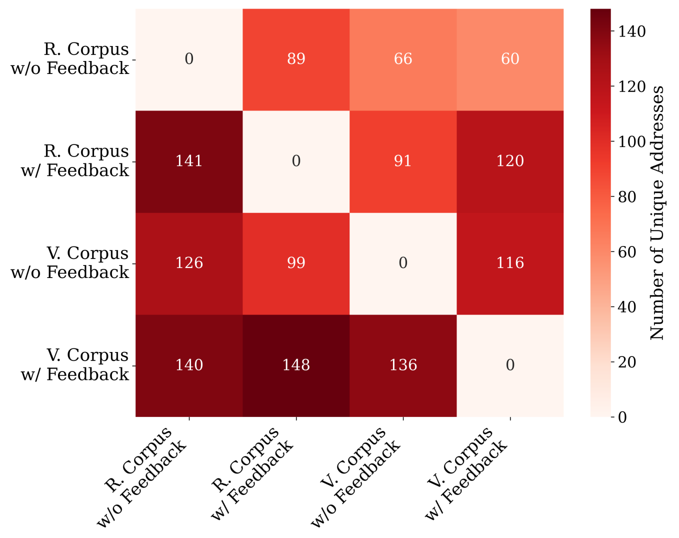

Fig. 6. μcode coverage uniqueness matrix. Addresses in row config NOT in column config. `#U.A.` stands for number of unique addresses. `R.` stands for random, and `V.` stands for valid.

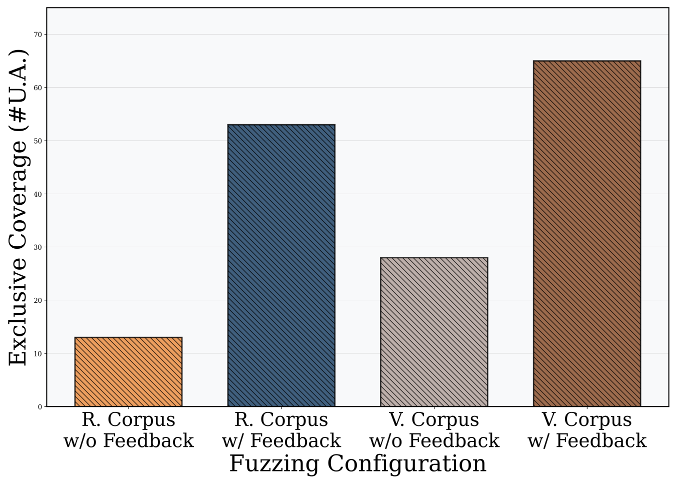

Fig. 7. Exclusive μcode Coverage for Different Fuzzing Configurations. (`#U.A.` Stands for Number of Unique Addresses. `R.` Stands for Random, and `V.` stands for Valid)

Fuzzing Step	**Average Time**	**std dev**
Hypervisor Setup arch-state	164.173us	3.317us
Hypervisor Setup memory	149.833us	3.943us
State Capturing	18.404us	1.600us
Coverage Setup	183.054us	86.030us
Coverage Collection	55.594us	2.808us
Total Overhead (Single Execution Round)	*571.058us*	-
Baseline total overhead	566.490ms	-
Fuzzilicon total overhead	18.274ms	-

Fig. 8. Timing Breakdown of Overheads in Fuzzilicon. *std dev* Stands for Standard Deviation.

baseline methodologies.

In total, across all our fuzzing campaigns, Fuzzilicon achieved coverage of $2,867$ unique μcode addresses, representing 16.27% of all hookable μcode addresses ($17,624$) in the target CPU architecture.

D. Performance Analysis

Fuzzilicon introduces two primary sources of overhead: 1) Hypervisor-based execution environment (reset architectural state, memory, start end stop VM execution), and 2) μcode-coverage collection and fuzzing framework overhead like capturing the architectural state. To quantify these, we instrumented Fuzzilicon to collect timing measurements from each source of overhead (see Table 8). In comparison to the baseline approach [45], Fuzzilicon optimization significantly reduces instrumentation overhead. The baseline approach, when fully parallelized across 16 hooks, incurs an overhead of 566.490 milliseconds to complete one fuzzing round, whereas Fuzzilicon completes the same task in only 18.274 milliseconds, achieving an approximately $31\times$ overhead reduction in the best-case scenario.

VII. Discussion

Portability to other CPU families. While the Fuzzilicon framework was evaluated on an *Intel N3350 processor* (Goldmont microarchitectural, `CPUID: 0x506ca`), its applicability extends beyond this specific CPU. The framework can be adapted to other Red-unlocked *Intel* processors with minimal modifications, primarily adjusting microarchitectural-specific parameters. No modification is required for processors sharing the same microarchitecture or CPU family. Extending Fuzzilicon to other CPU vendors or non-Red-unlocked CPUs is feasible if μcode update interfaces remain accessible. When μcode update capabilities are available, only the instrumentation logic requires adjustment to accommodate the target CPU's specific update procedures. Recently, *Google* researchers disclosed

AMD's EntryBleed vulnerability [67], which enables loading malicious μcode patches on *AMD* processors. The instrumentation logic of Fuzzilicon could potentially be extended to exploit this vulnerability for patching μcode updates in *AMD* systems. However, unlike *Intel*'s extensively reverse-engineered μcode architecture, *AMD*'s μcode is still poorly understood and undocumented. Consequently, we defer *AMD* CPU fuzzing to future research efforts.

VIII. Related work

We have structured the result of our literature research in Table VII. We have identified 19 hardware fuzzers [20]–[38] published in the last years that we categorized using the categories: **Type** (pre-silicon/post-silicon; does it require RTL source code), **Target** (on which target architecture was the fuzzer tested), **Input generation** (how is the input generated), **Platform** (how is the target CPU run; emulated CPU, CPU on Field Programmable Gate Array (FPGA), bare-metal, or requiring OS), **ISA-Simulation** (does the fuzzer require an additional ISA simulator), **Vulnerability detection** (which method is used to detect bugs), **Microarchitectural feedback** (does it use microarchitectural feedback), and **Fuzzing input restriction** (is the fuzzing input generation restricted before running it on the target).

In the following, we highlight the differences between Fuzzilicon and the three previous post-silicon fuzzers. Osiris [22] is a post-silicon *x86* fuzzer that detects timing-based side-channel vulnerabilities. It does so by measuring the duration instruction triplets take to execute. Fuzzilicon, in contrast, has to goal to find architectural and microarchitectural CPU bugs by analyzing the CPU state. SiliFuzz [24] initially fuzzes CPU ISA simulators using software coverage feedback to generate a test corpus and collect expected behavioral traces for each test case. Subsequently, the framework executes the generated corpus on the target CPU and validates architectural register states against the simulator-derived expected values. Fuzzilicon does not require an ISA simulator and runs on the bare-metal CPU, hence it does not restrict the input generation to ”non-destructive” *x86* instruction sequences. Like Fuzzilicon, RISCVuzz [37] does not use an ISA Simulator, but it restricts the instruction generation to not include, e.g., `WRMSR` equivalent instructions. Further, it targets the *RISC-V* architecture and runs on top of an OS.

Fuzzilicon is the first post-silicon CPU fuzzer that leverages hypervisor environments to not restrict fuzzing input generation to ”non-destructive” instructions. It runs on the bare-metal CPU without an OS. Further, it is the first CPU fuzzer using microarchitectural state to enhance the fuzzing feedback, leaning towards the concept of pre-silicon fuzzers that modify the behavior of the target CPU.

IX. Conclusion

In this paper, we presented Fuzzilicon, the first post-silicon fuzzer for x86 CPUs that leverages μcode-level feedback to systematically explore internal microarchitectural behavior. By introducing μcode coverage as a novel guidance signal, Fuzzilicon opens a new direction for hardware fuzzing beyond architectural observability. Our system integrates a lightweight, hypervisor-based execution environment to ensure isolated, deterministic test runs on real silicon and introduces a serialization oracle to detect vulnerabilities without relying on formal specifications.

We address and formalize the core challenges of post-silicon *x86* fuzzing *microarchitectural invisibility*, *absence of bug detection oracle*, *non-deterministic execution*, and *fault containment*. We demonstrate how Fuzzilicon overcomes these challenges through a principled design. Our evaluation shows that Fuzzilicon can uncover 5 significant findings, including two previously unknown μcode-level speculative-execution vulnerabilities (*F2* and *F3*) and automatically rediscover μSpectre vulnerabilities (*F1*). Fuzzilicon achieved 16.27% coverage of all hookable μcode paths, setting a new baseline for introspective fuzzing on proprietary CPUs. Along the way, we introduce optimized instrumentation strategies that reduce instrumentation overhead by $31\times$ compared to the baseline. Together, these contributions establish Fuzzilicon as a practical, efficient, and powerful framework for uncovering security-critical vulnerabilities in modern, closed-source processors.

Acknowledgment

Our research work was partially funded by Intel's Scalable Assurance Program, DFG-SFB 1119-236615297, the European Union under Horizon Europe Programme-Grant Agreement 101070537-CrossCon, NSF-DFG-Grant 538883423, and the European Research Council under the ERC Programme-Grant 101055025-HYDRANOS. This work does not in any way constitute an Intel endorsement of a product or supplier. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect those of Intel, the European Union, or the European Research Council. We would like to thank Jason Fung, Matthias Schunter, and Nassim Corteggiani from Intel for their valuable feedback.

Ethics Considerations

This research explores the security properties of commercial x86 CPUs through post-silicon fuzzing and μcode-level introspection. While our work targets undocumented processor internals, all experiments were conducted on isolated, non-networked hardware under tightly controlled conditions. We did not engage with production systems, user data, or shared infrastructure at any point during our evaluation.

In accordance with responsible disclosure protocols, we provided comprehensive documentation of our findings to Intel Corporation via `secure@intel.com`, including detailed descriptions of microcode-level anomalies, behavioral inconsistencies, potential speculative execution leakage paths, complete attack demonstrations, and a draft of our paper. Intel's security team acknowledged our research and validated the reported attack scenarios. Their assessment concluded that current mitigation strategies sufficiently address the identified

Year	Method	Type	Target	Input generation	ISA-Simulator	Vulnerability detection	Platform	μ-arch feedback	Fuzz-input restriction
2018	RFUZZ [20]	pre-silicon	RISC-V+	Stochastic	not-applicable	Assertion checking (i.d.)	FPGA	—	—
2021	DIFUZZRTL [21]	pre-silicon	RISC-V	Stochastic	yes	Golden reference model (o.d.)	FPGA	—	—
2021	EPEX [23]	pre-silicon	RISC-V	Stochastic	yes	Equivalent program (i.d.)	FPGA	—	—
2022	TheHuzz [25]	pre-silicon	RISC-V+	Stochastic	yes	Golden reference model (o.d.)	Emulation	—	—
2022	Cross-Level [...] [26]	pre-silicon	RISC-V	Stochastic	yes	Golden reference model (o.d.)	Emulation	—	—
2023	HyPFuzz [27]	pre-silicon	RISC-V	Formal-assisted	yes	Golden reference model (o.d.)	Emulation	—	—
2023	PSOFuzz [28]	pre-silicon	RISC-V	Stochastic	yes	Golden reference model (o.d.)	Emulation	—	—
2023	MABFuzz [29]	pre-silicon	RISC-V	Stochastic	yes	Golden reference model (o.d.)	Emulation	—	—
2023	MorFuzz [30]	pre-silicon	RISC-V	Template	yes	Golden reference model (o.d.)	Emulation	—	—
2023	SoCFuzzer [31]	pre-silicon	RISC-V	Stochastic	no	Assertion checking (i.d.)	FPGA+OS	—	—
2023	ProcessorFuzz [32]	pre-silicon	RISC-V	Stochastic	yes	Golden reference model (o.d.)	Emulation	—	—
2023	SurgeFuzz [33]	pre-silicon	RISC-V	Stochastic	no	Assertion checking (i.d.)	Emulation	—	—
2023	StressTest [34]	pre-silicon	unknown	Template	yes	Golden reference model (o.d.)	Emulation	—	—
2024	ChatFuzz [35]	pre-silicon	RISC-V	LLM-assisted	yes	Golden reference model (o.d.)	Emulation	—	—
2024	Cascade [36]	pre-silicon	RISC-V	BasicBlock	yes	Halting problem (i.d.)	Emulation	—	—
2024	FuzzWiz [38]	pre-silicon	not-applicable	Stochastic	not-applicable	Assertion checking (i.d.)	Emulation	—	—
2021	Osiris [22]	**post-silicon**	**x86**	Stochastic	**no**	Time measurement (o.d.)	OS	no	yes
2021	SiliFuzz [24]	**post-silicon**	**x86**	Stochastic	yes	Inter-device (o.d.)	OS	no	yes
2024	RISCVuzz [37]	**post-silicon**	RISC-V	Stochastic	**no**	Inter-device (o.d.)	OS	no	yes
2025	Fuzzilicon	**post-silicon**	**x86**	Stochastic	**no**	**Serialized oracle (i.d.)**	**Bare-metal**	**yes**	**no**

Fig. 9. Comparison of CPU fuzzers. Fuzzilicon is the first post-silicon *x86* fuzzer that does not require an operating system (hence has no limitations regarding input generation) and uses a *Serialized-oracle* model for bug detection. Further, it is the first general *x86* fuzzer that does not require an ISA-Simulation. The o.d. stands for output-driven. The i.d. stands for input-driven.

vulnerabilities, resulting in their decision not to assign CVE designations to these discoveries.

No human subjects or personal data were involved; therefore, no institutional ethics review was required. However, we followed the principles outlined in the Menlo Report [68], emphasizing respect for persons, beneficence, and responsible stewardship of research outcomes. We aim to advance understanding of CPU security while minimizing harm and supporting industry efforts to improve processor trustworthiness.

References

[1] T. Ormandy, "Zenbleed," 2023. [Online]. Available: https://lock.cmpxchg8b.com/zenbleed.html

[2] ——, "Reptar," 2023. [Online]. Available: https://lock.cmpxchg8b.com/reptar.html

[3] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018. [Online]. Available: https://doi.org/10.48550/arXiv.1801.01207

[4] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020. [Online]. Available: https://doi.org/10.1145/3399742

[5] S. Wiebing, A. de Faveri Tron, H. Bos, and C. Giuffrida, "InSpectre Gadget: Inspecting the Residual Attack Surface of Cross-privilege Spectre v2," in *USENIX Security*, Aug. 2024, distinguished Paper Award. [Online]. Available: Paper=https://download.vusec.net/papers/inspectre_sec24.pdfWeb=https://vusec.net/projects/native-bhiCode=https://github.com/vusec/inspectre-gadgetVideo=https://www.youtube.com/watch?v=bd7l-xhEtCE

[6] P. Koppe, B. Kollenda, M. Fyrbiak, C. Kison, R. Gawlik, C. Paar, and T. Holz, "Reverse engineering x86 processor microcode," in *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017, pp. 1163–1180. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/koppe

[7] National Institute of Standards and Technology, "Hardware security failure scenarios," NIST, Interagency Report IR 8517, Nov. 2024, available at https://doi.org/10.6028/NIST.IR.8517.

[8] S. R. Sarangi, A. Tiwari, and J. Torrellas, "Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006, pp. 26–37.

[9] C. Deutschbein and C. Sturton, "Mining security critical linear temporal logic specifications for processors," in *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. IEEE, 2018, pp. 18–23.

[10] B. Wile, J. Goss, and W. Roesner, *Comprehensive functional verification: The complete industry cycle*. Morgan Kaufmann, 2005.

[11] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "{HardFails}: insights into {software-exploitable} hardware bugs," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 213–230.

[12] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *LASER Summer School on Software Engineering*. Springer, 2011, pp. 1–30.

[13] I. Wagner and V. Bertacco, "Engineering trust with semantic guardians," in *2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2007, pp. 1–6.

[14] M. Hicks, C. Sturton, S. T. King, and J. M. Smith, "Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 517–529.

[15] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: a hardware description language for secure information flow," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 109–120, 2011.

[16] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A language for hardware-level security policy enforcement," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014, pp. 97–112.

[17] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *Acm Sigplan Notices*, vol. 50, no. 4, pp. 503–516, 2015.

[18] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, and J. Rajendran, "{TheHuzz}: Instruction fuzzing of processors using {Golden-Reference} models for finding {Software-Exploitable} vulnerabilities," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3219–3236.

[19] M. Rostami, C. Chen, R. Kande, H. Li, J. Rajendran, and A.-R. Sadeghi,

"Fuzzerfly effect: Hardware fuzzing for memory safety," *IEEE Security and Privacy*, vol. 22, no. 4, pp. 76–86, 2024.

[20] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: coverage-directed fuzz testing of rtl on fpgas," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '18. ACM, Nov. 2018. [Online]. Available: https://doi.org/10.1145/3240765.3240842

[21] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2021, pp. 1286–1303. [Online]. Available: https://doi.org/10.1109/sp40001.2021.00103

[22] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow, "Osiris: Automated discovery of microarchitectural side channels," *ArXiv*, vol. abs/2106.03470, 2021. [Online]. Available: https://www.semanticscholar.org/paper/c6ab953b41d2ea810657c3ee4989409177f51af2

[23] L. Klemmer and D. Große, "Epex: Processor verification by equivalent program execution," *Proceedings of the 2021 Great Lakes Symposium on VLSI*, 2021. [Online]. Available: https://doi.org/10.1145/3453688.3461497

[24] K. Serebryany, M. Lifantsev, K. Shtoyk, D. Kwan, and P. Hochschild, "Silifuzz: Fuzzing cpus by proxy," vol. abs/2110.11519, 2021. [Online]. Available: https://doi.org/10.48550/arXiv.2110.11519

[25] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, and J. Rajendran, "TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3219–3236. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/kande

[26] N. Bruns, V. Herdt, D. Große, and R. Drechsler, "Efficient cross-level processor verification using coverage-guided fuzzing," *Proceedings of the Great Lakes Symposium on VLSI 2022*, 2022. [Online]. Available: https://doi.org/10.1145/3526241.3530340

[27] C. Chen, R. Kande, N. Nguyen, F. Andersen, A. Tyagi, A. Sadeghi, and J. Rajendran, "Hypfuzz: Formal-assisted processor fuzzing," *ArXiv*, vol. abs/2304.02485, 2023. [Online]. Available: https://www.semanticscholar.org/paper/c16aa03dab11e9d05cda8b9c2e8d884be3ba6fdf

[28] C. Chen, V. Gohil, R. Kande, A. Sadeghi, and J. Rajendran, "Psofuzz: Fuzzing processors with particle swarm optimization," *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–9, 2023. [Online]. Available: https://doi.org/10.1109/ICCAD57390.2023.10323913

[29] V. Gohil, R. Kande, C. Chen, A. Sadeghi, and J. Rajendran, "Mabfuzz: Multi-armed bandit algorithms for fuzzing processors," *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2311.14594

[30] J. Xu, Y. Liu, S. He, H. Lin, Y. Zhou, and C. Wang, "Morfuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation," pp. 1307–1324, 2023. [Online]. Available: https://www.semanticscholar.org/paper/fbc2eea9ea1a103902477456a16566252835d29b

[31] M. M. Hossain, A. Vafaei, K. Z. Azar, F. Rahman, F. Farahmandi, and M. Tehranipoor, "Socfuzzer: Soc vulnerability detection using cost function enabled fuzz testing," in *2023 Design, Automation &amp; Test in Europe Conference &amp; Exhibition (DATE)*. IEEE, Apr. 2023, pp. 1–6. [Online]. Available: https://doi.org/10.23919/date56975.2023.10137024

[32] S. Canakci, C. Rajapaksha, L. Delshadtehrani, A. Nataraja, M. B. Taylor, M. Egele, and A. Joshi, "Processorfuzz: Processor fuzzing with control and status registers guidance," *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 1–12, 2023. [Online]. Available: https://doi.org/10.1109/HOST55118.2023.10133714

[33] Y. Sugiyama, R. Matsuo, and R. Shioya, "Surgefuzz: Surge-aware directed fuzzing for cpu designs," *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–9, 2023. [Online]. Available: https://doi.org/10.1109/ICCAD57390.2023.10323819

[34] I. Wagner, V. Bertacco, and T. Austin, "Stresstest: an automatic approach to test generation via activity monitors," *Proceedings. 42nd Design Automation Conference, 2005.*, pp. 783–788, 2005. [Online]. Available: https://doi.org/10.1145/1065579.1065788

[35] M. Rostami, M. Chilese, S. Zeitouni, R. Kande, J. Rajendran, and A. Sadeghi, "Beyond random inputs: A novel ml-based hardware fuzzing," *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2404.06856

[36] F. Solt, K. Ceesay-Seitz, and K. Razavi, "Cascade: CPU fuzzing via intricate program generation," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 5341–5358. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/solt

[37] F. Thomas, L. Hetterich, R. Zhang, D. Weber, L. Gerlach, and M. Schwarz, "Riscvuzz: Discovering architectural cpu vulnerabilities via differential hardware fuzzing." [Online]. Available: https://www.semanticscholar.org/paper/31654caccc4fe7f9af528704ad2971eb6c3931b2

[38] D. N. Gadde, A. Kumar, D. Lettnin, and S. Simon, "Fuzzwiz - fuzzing framework for efficient hardware coverage," *2024 International Symposium on Electronics and Telecommunications (ISETC)*, pp. 1–5, 2024. [Online]. Available: https://doi.org/10.1109/ISETC63109.2024.10797245

[39] L. Wu, M. Rostami, H. Li, J. Rajendran, and A.-R. Sadeghi, "{GenHuzz}: An efficient generative hardware fuzzer," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 1787–1805.

[40] L. Wu, M. Rostami, H. Li, and A.-R. Sadeghi, "Hfl: Hardware fuzzing loop with reinforcement learning," in *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2025, pp. 1–7.

[41] P. Borkar, C. Chen, M. Rostami, N. Singh, R. Kande, A.-R. Sadeghi, C. Rebeiro, and J. Rajendran, "{WhisperFuzz}:{White-Box} fuzzing for detecting and locating timing vulnerabilities in processors," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5377–5394.

[42] N. Mosier, H. Nemati, J. Mitchell, and C. Trippel, "Analyzing and exploiting branch mispredictions in microcode," 01 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2501.12890

[43] M. Ermolov, D. Sklyarov, and M. Goryachy, "Undocumented x86 instructions to control the cpu at the microarchitecture level in modern intel processors," *Journal of Computer Virology and Hacking Techniques*, vol. 19, pp. 351–365, 2022. [Online]. Available: https://doi.org/10.1007/s11416-022-00438-x

[44] D. S. Maxim Goryachy and M. Ermolov, "Chip red pill: How we achieved the arbitrary [micro]code execution inside intel atom cpus," *Offensive Con*, 2022, https://github.com/chip-red-pill/uCodeDisasm/. [Online]. Available: https://www.offensivecon.org/speakers/2022/maxim-goryachy.html

[45] P. Borrello, C. Easdon, M. Schwarzl, R. Czerny, and M. Schwarz, "Customprocessingunit: Reverse engineering and customization of intel microcode," in *2023 IEEE Security and Privacy Workshops (SPW)*, vol. 22. IEEE, May 2023, pp. 285–297. [Online]. Available: https://doi.org/10.1109/spw59333.2023.00031

[46] P. K. Benjamin Kollenda, "Everything you want to know about x86 microcode, but might have been afraid to ask. an introduction into reverse-engineering x86 microcode and writing it yourself," *CCC*, 2017. [Online]. Available: https://media.ccc.de/v/34c3-9058-everything_you_want_to_know_about_x86_microcode_but_might_have_been_afraid_to_ask

[47] Google, "Americal fuzzy loop," https://github.com/google/AFL, 2019.

[48] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*, ser. CCS '22. ACM, Nov. 2022.

[49] I. Wagner and V. Bertacco, "Reversi: Post-silicon validation system for modern microprocessors," in *2008 IEEE International Conference on Computer Design*. IEEE, 2008, pp. 307–314.

[50] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2123–2138.

[51] S. Dechand, "The magic behind feedback-based fuzzing," https://www.code-intelligence.com/blog/the-magic-behind-feedback-based-fuzzing.

[52] A. AWS, "What's the difference between type 1 and type 2 hypervisors?" https://aws.amazon.com/compare/the-difference-between-type-1-and-type-2-hypervisors/.

[53] GIGABYTE, "Gb-bpce-3350c (rev. 1.0)," https://www.gigabyte.com/de/Mini-PcBarebone/GB-BPCE-3350C-rev-10.

[54] P. Research, "Intel management engine jtag proof of concept," https://github.com/ptresearch/IntelTXE-PoC, 2022.

[55] Y. Alaoui, "Exploiting intel's management engine – part 1: Understanding pt's txe poc (intel-sa-00086)," https://kakaroto.ca/2019/11/exploiting-intels-management-engine-part-1-understanding-pts-txe-poc/, 2019.

[56] M. Goryachy and M. Ermolov, "How to hack a turned-off computer or running unsigned code in intel management engine," https://www.blackhat.com/eu-17/briefings.html, 2017.

[57] "Nixos," https://nixos.org/.

[58] GNU, "Basic blocks," https://gcc.gnu.org/onlinedocs/gccint/Basic-Blocks.html.

[59] Intel, "Intel® 64 and ia-32 architectures software developer's manual volume 3 (3a, 3b, 3c, & 3d): System programming guide," December 2024. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[60] S. Karvandi, "Hypervisor from scratch – part 4: Address translation using extended page table (ept)," https://rayanfam.com/topics/hypervisor-from-scratch-part-4/.

[61] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*, 2020, pp. 151–163.

[62] A. Caliskan, F. Yamaguchi, E. Dauber, R. Harang, K. Rieck, R. Greenstadt, and A. Narayanan, "When coding style survives compilation: De-anonymizing programmers from executable binaries," in *Proceedings 2018 Network and Distributed System Security Symposium*, ser. NDSS 2018. Internet Society, 2018.

[63] D. Moghimi, "Downfall: Exploiting speculative data gathering," 2023. [Online]. Available: https://www.semanticscholar.org/paper/51abdda691022e97b079f676634ea8ec222056f0

[64] M. Rostami, S. Zeitouni, R. Kande, C. Chen, P. Mahmoody, J. Rajendran, and A.-R. Sadeghi, "Lost and found in speculation: Hybrid speculative vulnerability detection," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[65] J. Hur, S. Song, S. Kim, and B. Lee, "Specdoctor: Differential fuzz testing to find transient execution vulnerabilities," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1473–1487.

[66] N. Schiller, X. Xu, L. Bernhard, N. Bars, M. Schloegel, and T. Holz, "Novelty not found: Adaptive fuzzer restarts to improve input space coverage (registered report)," in *Proceedings of the 2nd International fuzzing workshop*, 2023, pp. 12–20.

[67] AMD, "Amd cpu microcode signature verification vulnerability," https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7033.html, 2025.

[68] H. Security, "The menlo report: Ethical principles guiding information and communication technology research," https://www.dhs.gov/sites/default/files/publications/CSD-MenloPrinciplesCORE-20120803_1.pdf.

Appendix A
Microcode Patching and Feedback

Algorithm 2 Instrumentation Logic for Microcode Hooking and Coverage Collection pseudo-code

```
entry_$i$:
    JmpImm(hook_handler_$i$_even)
    JmpImm(hook_handler_$i$_odd)
hook_handler_$i$_even:
    StagingBuf[$0xba00$, $0xbb00$] ← R10, R14
    R10 ← $2 \cdot i + 0$
    R14 ← exit_$i$_even
    JmpImm(handler)
hook_handler_$i$_odd:
    StagingBuf[$0xba00$, $0xbb00$] ← R10, R14
    R10 ← $2 \cdot i + 1$
    R14 ← exit_$i$_odd
    JmpImm(handler)
handler:
    Save additional registers to StagingBuf
    cov_idx ← $R10 \cdot 2 + 0x1000$
    RAM[cov_idx] ← RAM[cov_idx] +1
    RAM[$R10 \cdot 8 + 0x1400$] ← RIP
    Collect any other state as needed
    Restore registers from StagingBuf
    JmpReg(R14)
exit_$i$_even:
    R10, R14 ← StagingBuf[$0xba00$, $0xbb00$]
    Instruction $i_0^{\text{even}}$
    JmpImm($a_0^{\text{even}}$)
exit_$i$_odd:
    R10, R14 ← StagingBuf[$0xba00$, $0xbb00$]
    Instruction $i_0^{\text{odd}}$
    JmpImm($a_0^{\text{odd}}$)
```

Appendix B
Serialized Oracle and Unrolling Misaligned Jumps

During fuzzing, we encounter `jump/call/ret` target instructions that do not point to instruction-aligned boundaries. An example is visualized in Figure 10: the `JMP` instruction on the left side jumps to the payload of the `MOV` instruction, continuing execution there.

When executing instructions misaligned, inserting fence instructions between regular instructions will change the execution of the program. In misaligned execution, instructions might be decoded that reach over several regular *x86* instructions. By inserting fence instructions, the decoded instruction will, therefore, change, resulting in a different program output.

When encountering such a jump during *serialization*, a new *serialization* originating at the jump's target address (see Figure 10) mus be started. We essentially suggest unwrapping the misaligned execution until the first illegal opcode and insert

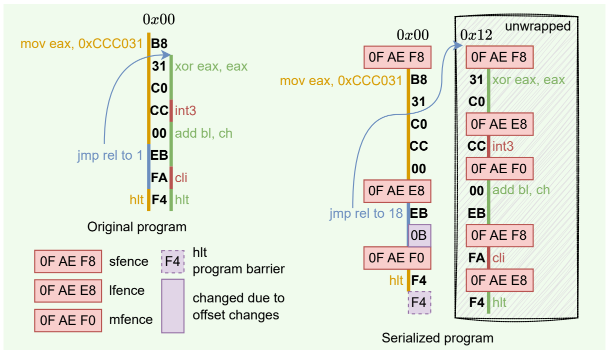

Fig. 10. An exemplary *serialization* of a program with an instruction-misaligned jump. The original program on the left contains a jump that jumps inside the payload of the `MOV` instruction. To *serialize* the program, first, the instruction-misaligned instructions are unwrapped, then, both separate programs are serialized and joined using a `HLT` instruction.

fence instructions between those instructions, like we would do for a regular instruction. This technique is detailed in Superset Assembly [62]. We propose joining the unwrapped and regular programs using, e.g., a `HLT` instruction, which will prevent the execution from continuing to the other program parts. Since all execution-flow changes are controlled, it can be ensured that the control flow will not cross the sub-program boundaries in an unintended fashion. This approach is only applicable for jumps with a target address known at *serialization* time.

During runtime address mapping, generally, the execution flow may be redirected to potentially any address of the original program, hence, the program must be unwrapped for each potential instruction offset. As a runtime optimization, the execution of a fuzzing input may be traced before *serialization* using our hypervisor. Then, *serialization* can be applied conditionally to relevant program parts only.

Appendix C
Speculative fuzzing CPU lock-ups

The complete dataset is available as a permanently archived artifact at https://doi.org/10.5281/zenodo.17012971, accessible through the file `speculative_fuzzing_CPU_lockups.csv`.

TABLE I
COMPREHENSIVE CATALOG OF MICROCODE OPERATIONS WITH PERSISTENT SPECULATIVE EFFECTS

Instruction	Type	Disassembly
0x0e7500037033	StableTimeout	tmp7:= LDSTGBUF_DSZ64_ASZ16_SC1(tmp3)
0x2e750003103a	Unstable	tmp1:= LDSTGBUF_DSZ64_ASZ16_SC1(tmp10) !m2
0x0822c6df2232	StableTimeout	tmp2:= MOVETOCREG_AND_DSZ64(tmp2, 0x00000003, 0x7c6) !m0
0x0a62019c02f0	Unstable	MOVETOCREG_BTR_DSZ64(tmp0, 0x0000000e, 0x701) !m0
0x0a62019c02fb	StableTimeout	MOVETOCREG_BTR_DSZ64(tmp11, 0x0000000e, 0x701) !m0
0x0a62019c02fd	StableTimeout	MOVETOCREG_BTR_DSZ64(tmp13, 0x0000000e, 0x701) !m0
0x0a621c8002f0	StableTimeout	MOVETOCREG_BTR_DSZ64(tmp0, 0x0000000e, 0x01c) !m0
0x0a628c5002b0	Unstable	MOVETOCREG_BTR_DSZ64(tmp0, 0x00000009, 0x48c)
0x0a62c3180271	Unstable	MOVETOCREG_BTR_DSZ64(tmp1, 0x00000004, 0x6c3)
0x0a62c31802d4	StableTimeout	MOVETOCREG_BTR_DSZ64(tmpv0, 0x0000000c, 0x6c3)
0x0a62fe1c033a	StableTimeout	MOVETOCREG_BTR_DSZ64(tmp10, 0x00000010, 0x7fe)
0x0a62fe5c033a	StableTimeout	MOVETOCREG_BTR_DSZ64(tmp10, 0x00000011, 0x7fe)
0x0a62fe9c02b5	StableTimeout	MOVETOCREG_BTR_DSZ64(tmp5, 0x0000000a, 0x7fe) !m0
0x1a62cd880330	StableTimeout	MOVETOCREG_BTR_DSZ64(tmp0, 0x00000012, 0x2cd) !m0,m1
0x1a62cd880332	StableTimeout	MOVETOCREG_BTR_DSZ64(tmp2, 0x00000012, 0x2cd) !m0,m1
0x0962015c03b2	StableTimeout	MOVETOCREG_BTS_DSZ64(tmp2, 0x00000019, 0x701)
0x0962019c02ff	StableTimeout	MOVETOCREG_BTS_DSZ64(tmp15, 0x0000000e, 0x701) !m0
0x096204440370	Unstable	MOVETOCREG_BTS_DSZ64(tmp0, 0x00000015, 0x104)
0x096205040230	Unstable	MOVETOCREG_BTS_DSZ64(tmp0, 0x105)
0x09621cd747f4	StableTimeout	tmp4:= MOVETOCREG_BTS_DSZ64(tmp4, 0x0000003f, 0x51c) !m0
0x09623b1b13f1	Unstable	tmp1:= MOVETOCREG_BTS_DSZ64(tmp1, 0x0000001c, 0x63b)
0x096269000233	StableTimeout	MOVETOCREG_BTS_DSZ64(tmp3, 0x069)
0x096275d402b0	Unstable	MOVETOCREG_BTS_DSZ64(tmp0, 0x0000000b, 0x575) !m0
0x0962c3180273	StableTimeout	MOVETOCREG_BTS_DSZ64(tmp3, 0x00000004, 0x6c3)
0x0962e11c0200	StableTimeout	MOVETOCREG_BTS_DSZ64(, 0x7e1)
0x0962fe1c033d	StableTimeout	MOVETOCREG_BTS_DSZ64(tmp13, 0x00000010, 0x7fe)
0x19628f0c02b7	StableTimeout	MOVETOCREG_BTS_DSZ64(tmp7, 0x00000008, 0x38f) !m1
0x1962c10c0300	StableTimeout	MOVETOCREG_BTS_DSZ64(, 0x00000010, 0x3c1) !m1
0x1962c2480271	Unstable	MOVETOCREG_BTS_DSZ64(tmp1, 0x00000005, 0x2c2) !m1
0x1962cdc80330	Unstable	MOVETOCREG_BTS_DSZ64(tmp0, 0x00000013, 0x2cd) !m0,m1
0x0042011c0232	StableTimeout	MOVETOCREG_DSZ64(tmp2, 0x701)
0x0042011f0230	Unstable	tmp0:= MOVETOCREG_DSZ64(tmp0, 0x701)
0x00420400023f	StableTimeout	MOVETOCREG_DSZ64(tmp15, 0x004)
0x00421a000200	StableTimeout	MOVETOCREG_DSZ64(, 0x00000000, 0x01a)
0x00421c000214	Unstable	MOVETOCREG_DSZ64(tmpv0, 0x01c)
0x00421d000238	StableTimeout	MOVETOCREG_DSZ64(tmp8, 0x01d)
0x004229140200	StableTimeout	MOVETOCREG_DSZ64(, 0x00000000, 0x529)
0x004229140235	Unstable	MOVETOCREG_DSZ64(tmp5, 0x529)
0x004229140237	StableTimeout	MOVETOCREG_DSZ64(tmp7, 0x529)
0x004229140238	Unstable	MOVETOCREG_DSZ64(tmp8, 0x529)
0x00422914023b	StableTimeout	MOVETOCREG_DSZ64(tmp11, 0x529)
0x004267000230	StableTimeout	MOVETOCREG_DSZ64(tmp0, 0x067)
0x004267000231	StableTimeout	MOVETOCREG_DSZ64(tmp1, 0x067)
0x004267000234	StableTimeout	MOVETOCREG_DSZ64(tmp4, 0x067)
0x004267000235	StableTimeout	MOVETOCREG_DSZ64(tmp5, 0x067)
0x004267000236	StableTimeout	MOVETOCREG_DSZ64(tmp6, 0x067)
0x004267000238	StableTimeout	MOVETOCREG_DSZ64(tmp8, 0x067)
0x004267000239	StableTimeout	MOVETOCREG_DSZ64(tmp9, 0x067)
0x00426700023a	StableTimeout	MOVETOCREG_DSZ64(tmp10, 0x067)
0x00426700023b	StableTimeout	MOVETOCREG_DSZ64(tmp11, 0x067)
0x00426700023e	StableTimeout	MOVETOCREG_DSZ64(tmp14, 0x067)
0x004270000230	StableTimeout	MOVETOCREG_DSZ64(tmp0, 0x070)
0x004270000232	StableTimeout	MOVETOCREG_DSZ64(tmp2, 0x070)
0x004277140230	StableTimeout	MOVETOCREG_DSZ64(tmp0, 0x577)
0x00428e1c0230	StableTimeout	MOVETOCREG_DSZ64(tmp0, 0x78e)
0x00428e1c0231	StableTimeout	MOVETOCREG_DSZ64(tmp1, 0x78e)
0x00428e1c0232	StableTimeout	MOVETOCREG_DSZ64(tmp2, 0x78e)
0x00428e1c0234	StableTimeout	MOVETOCREG_DSZ64(tmp4, 0x78e)
0x00428e1c0239	StableTimeout	MOVETOCREG_DSZ64(tmp9, 0x78e)
0x00428e1c023a	StableTimeout	MOVETOCREG_DSZ64(tmp10, 0x78e)
0x00428e1c023b	StableTimeout	MOVETOCREG_DSZ64(tmp11, 0x78e)
0x00429e1c0233	StableTimeout	MOVETOCREG_DSZ64(tmp3, 0x79e)

APPENDIX D
ARTIFACT APPENDIX

This appendix assists users and future researchers in utilizing the Fuzzilicon artifact to reproduce the results presented in our paper, including fuzzing campaigns, coverage analysis, and proof-of-concept demonstrations for reported findings. Due to the specialized hardware requirements for setting up the Fuzzilicon framework and the complexity of configuration (particularly red-unlock *Intel* CPU microcode), we are applying only for the availability badge, following NDSS guidelines.

Fuzzilicon is an x86 CPU fuzzer that leverages microcode coverage as feedback to guide fuzzing campaigns. The Fuzzilicon framework provides: (1) lightweight microcode instrumentation, (2) a minimal hypervisor for executing fuzzing inputs in isolated environments, (3) a serialization oracle for vulnerability detection, (4) core x86 CPU fuzzing components including input generation, mutation engine, feedback collection, and vulnerability detection, (5) specialized capabilities for microcode-level fuzzing to detect speculative execution vulnerabilities, (6) coverage evaluation comparing microcode-guided fuzzing against baseline approaches, and (7) detection of seven findings.

The artifact includes source code for all Fuzzilicon components and experiments, along with comprehensive documentation for future research applications.

This appendix is organized as follows: Section D-A describes artifact access, hardware/software requirements, configurations, and benchmarks. Section D-B provides high-level installation and configuration steps for environment preparation. Finally, Section D-C details the experimental workflow and instructions for reproducing each experiment described in the paper.

A. Description & Requirements

The Fuzzilicon artifact has specific hardware, software, and configuration requirements. Each category is detailed below:

1) How to access: The complete source code for the Fuzzilicon framework is open-sourced at https://github.com/0xCCF4/ufuzz and hosted at the Zenodo permanent archival repository at https://doi.org/10.5281/zenodo.17012971. These repositories contains source code and detailed documentation to reproduce all experimental results.

2) Hardware dependencies: The minimal hardware setup for running Fuzzilicon requires two Raspberry Pi 4 devices: (1) one serves as the fuzzer controller for generating test cases, collecting reports, and monitoring the target system, and (2) the second functions as a mass storage and virtual keyboard of the fuzzing agent.

Since Fuzzilicon targets x86 CPUs with microcode patch interface access capabilities, we selected the *Intel Apollo Lake (Celeron, Goldmont)* N3350 processor (`CPUID[1].EAX=0x506ca`). This CPU is integrated into the Gigabyte GB-BPCE-3350C board, which serves as our target platform. Given that the target CPU may become unresponsive during fuzzing and require automatic restarts, we employ a breadboard with power switching capabilities. This setup enables remote power cycling of the Gigabyte GB-BPCE-3350C board by connecting to the board's power control pins through a relay circuit, controlled by the Raspberry Pi.

To facilitate data communication between the Raspberry Pi devices and the target CPU board, we utilize a 5-port network switch for reliable network connectivity.

3) Software dependencies: The Fuzzilicon codebase is developed in the Rust programming language with the Cargo package manager for dependency management. We use NixOS as the OS of both Pis with the full system configuration available in our artifact.

Additionally, the artifact requires the `uasm.py` file from the CustomProcessingUnit project [45]. Fuzzilicon utilizes this script for compiling microcode updates during the fuzzing process.

4) Configuration: The *Intel Apollo Lake (Celeron, Goldmont)* N3350 target CPU does not natively provide access for applying customized microcode patches. To enable custom microcode patching capabilities, the CPU must first be "red-unlocked". The red-unlocking process follows established procedures documented in prior research [54]–[56]. Upon successful red-unlocking, users gain access to undocumented instructions that enable CRBUS access (`udbgrd` and `udbgwr` instructions), which are essential for microcode manipulation during fuzzing operations.

5) Benchmarks: None

B. Artifact Installation & Configuration

This section provides the high-level installation and configuration steps required to prepare the environment for artifact evaluation. For detailed instructions, troubleshooting guidance, and comprehensive documentation, please refer to the `README.md` file included in the artifact.

Dependency Installation: The Fuzzilicon project requires specific dependencies on both the Raspberry Pi controllers and the development environment:

```
# Dependencies
sudo apt install python3 python3-click \
     gcc-aarch64-linux-gnu build-essential git

# Rust Toolchain
curl --proto '=https' --tlsv1.2 -sSf
    https://sh.rustup.rs | sh -s --
    --default-toolchain none -y

rustup install nightly-2025-05-30
rustup target add x86_64-unknown-uefi
rustup target add aarch64-unknown-linux-gnu
rustup target add x86_64-unknown-linux-gnu
rustup default nightly-2025-05-30
```

CustomProcessingUnit [45] Setup: Download the CustomProcessingUnit project and place it in the parent directory of Fuzzilicon, or set the `UASM` environment variable to point to the `uasm.py` file location. Then apply the provided patch:

```
git apply uasm.py.patch
```

Raspberry Pi Image Creation: Install Nix package manager following the official documentation, then build the SD card images for the Raspberry Pi devices:

```
cd nix
# Configure SSH keys and IP addresses in
    configuration files
nix build .#images.master
nix build .#images.node
```

For subsequent deployments, use: `nix run`

Target Device Setup: Deploy the UEFI fuzzing application to the target CPU board:

```
HOST_NODE="<instrumentor_ip>" cargo xtask
    put-remote \
  --remote-ip <controller_address> \
  --source-ip <agent_address> \
  --netmask <network_mask> \
  --port <udp_port> \
  --startup <app_name>
```

Depending on the target fuzzing scenario, use `spec_fuzz` (speculative microcode fuzzing) or `fuzzer_device` (x86 instruction fuzzing) instead of `<app_name>`.

C. Experiment Workflow

Start the fuzzer master and execute experiments using the provided command-line interface:

```
fuzzer_master --help  # Available options/commands
```

Each of the following experiments includes built-in help documentation that allows you to specify parameters. For example:

```
fuzz_master afl --help

# Executes the main fuzzing loop with AFL
# mutations == Requires the `fuzzer_device` app
# running on the agent ==

# Usage: fuzz_master afl [OPTIONS]

# Options:
#   -s, --solutions <SOLUTIONS>
#   -c, --corpus <CORPUS>
#   -a, --afl-corpus <AFL_CORPUS>
#   -t, --timeout-hours <TIMEOUT_HOURS>
#   -d, --disable-feedback
#   -p, --printable-input-generation
#   -h, --help
```

Genetic Algorithm Fuzzing: This mode performs coverage-guided fuzzing using a genetic mutation algorithm to evolve test cases based on microcode coverage feedback.

```
fuzzer_master --database genetic_results.json \
  --instrumentor http://10.83.3.198:8000 \
  --agent 10.83.3.6:4444 \
  genetic
```

AFL-based Fuzzing Campaign: This mode executes the main fuzzing loop using AFL (American Fuzzy Lop) mutation strategies with microcode coverage guidance for comprehensive x86 instruction fuzzing.

```
fuzzer_master --database afl_campaign.json \
  --instrumentor http://10.83.3.198:8000 \
  --agent 10.83.3.6:4444 \
  afl
```

Speculative Microcode Fuzzing: This mode performs specialized fuzzing targeting speculative execution vulnerabilities by manipulating microcode patches and monitoring speculative execution behavior.

```
fuzzer_master --database spec_fuzzing.json \
  --instrumentor http://10.83.3.198:8000 \
  --agent 10.83.3.6:4444 \
  spec
```

Note: If you setup your agent and instrumentor on different IP addresses other than the default configuration, please modify the `--instrumentor` and `--agent` parameters accordingly in the above commands.

The fuzzing results will be stored to a database file (specifiable via the **–database** argument) (on fuzzer master) and can be viewed by running (on fuzzer master):

```
fuzz_viewer database.json
```

Reproducing Reported Findings The proof-of-concept implementations for the reported findings are located in the following directories within the artifact:

- *Microcode Speculation (`CRBUS`-based):*
  - Location: `speculation_ucode/src/main.rs`
  - Description: Microcode speculation harness for CRBUS interface
- *Microcode Speculation (`WRSGFLD`-based):*
  - Location: `fuzzer_device/examples/test_ucode_speculation.rs`
  - Description: Hypervisor speculation harness for `WRSGFLD` instruction
- *Speculative Window Termination:*
  - Location: `speculation_x86/src/main.rs`
  - Description: Terminating speculative execution window