Focus Session: Advanced Hybrid Hardware Fuzzing

Chen Chen
Texas A&M University
chenc@tamu.edu

Stephen Muttathil
Texas A&M University
stephen7929@tamu.edu

Mohamadreza Rostami
Technische Universität Darmstadt
mohamadreza.rostami@tu-darmstadt.de

Nikhilesh Singh
Technische Universität Darmstadt
nikhilesh.singh@tu-darmstadt.de

Lichao Wu
Technische Universität Darmstadt
lichao.wu@tu-darmstadt.de

Ahmad-Reza Sadeghi
Technische Universität Darmstadt
ahmad.sadeghi@tu-darmstadt.de

Jeyavijayan Rajendran
Texas A&M University
jv.rajendran@tamu.edu

***Abstract*—Modern processors are increasingly complex, with rich microarchitectural features and heterogeneous components. This complexity expands the attack surface and makes security vulnerabilities harder to detect using traditional security techniques. Hardware fuzzing has emerged as a scalable approach for uncovering insecure behaviors in modern processors. However, it often struggles to (i) explore hard-to-reach design spaces due to its randomness and (ii) locate the root causes of vulnerabilities due to design complexity.**

This work presents advanced hybrid hardware fuzzing techniques that combine the complementary strengths of fuzzing, formal verification, and static analysis to systematically detect and localize vulnerabilities in processors. Specifically, we investigate (i) the use of formal verification to guide fuzzing toward hard-to-reach design spaces, thereby enabling the discovery of subtle vulnerabilities, and (ii) the use of static analysis to extract and monitor timing behaviors at the register-transfer level (RTL), enabling localization of timing vulnerabilities that can arise even in functionally correct designs.

Finally, we outline future research directions, including using large language models to generate expert-informed tests, leveraging prior design knowledge to enhance fuzzing effectiveness on new processors, and transferring effective strategies from white-box fuzzing to black-box fuzzing environments.

***Index Terms*—Hardware Security, Hardware Fuzzing, Formal Verification, Information Flow Tracking**

I. Introduction

Modern processors continue to grow in complexity as computing needs and performance requirements increase, incorporating deeper pipelines, more sophisticated speculative execution, and more advanced microarchitectural features. However, they also expand the attack surface, leading to a massive growth of hardware vulnerabilities [1]. For example, the National Vulnerability Database has shown that the number of newly detected hardware vulnerabilities increased from 3 in 2012 to 814 in 2024 [2] as shown in Fig. 1. Many of these vulnerabilities arise not only from functional flaws, but also from complex interactions between microarchitectural features and timing behaviors, making them difficult to detect [3], [4].

Conventional hardware security strategies include directed testing, random regression, and formal verification [1], [5]. These techniques are effective for validating functional correctness, but struggle to scale to large designs such as modern processors and system-on-chips (SoCs) due to state-space

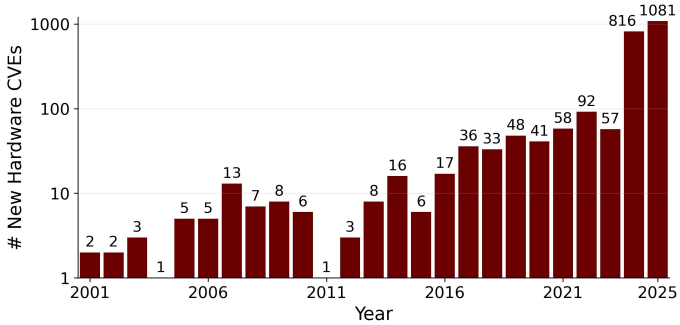

Fig. 1: Number of Hardware Vulnerabilities in the National Vulnerability Database [2].

explosion and the need for expertise to write assertions. In addition, the growing adoption of open-source ISAs such as `RISC-V`, which support custom extensions, provides processor designers unprecedented flexibility [6]. This flexibility also introduces a major security challenge: ensuring that all the designs are compliant and functionally correct. Moreover, unlike software vulnerabilities, most hardware vulnerabilities cannot be patched after fabrication. Therefore, to enhance the detection of hardware vulnerabilities before fabrication, there is an increasing need for automated, scalable, efficient, and compatible tools and methodologies with the existing IC design and verification flow.

Hardware fuzzing has shown promise for improving the security of processor designs [7]. Inspired by software fuzzing, hardware fuzzing has demonstrated strong capabilities in enhancing design-space exploration and uncovering vulnerabilities [8]–[10]. Compared to traditional verification approaches, including runtime checking and formal verification, hardware fuzzing offers greater automation, improved scalability, and broader coverage, making it particularly effective for pre-silicon vulnerability detection. Prior efforts have shown that hardware fuzzing can successfully expose functional flaws in processors [8], [11]–[14].

However, standalone hardware fuzzing has two limitations. First, its reliance on randomized or mutation-based input generation makes it inefficient for exploring hard-to-reach design spaces. As hardware designs become more complex, fuzzing often encounters a coverage plateau, where additional

978-3-9826741-1-7/DATE26/© 2026 EDAA

mutations fail to trigger new behaviors despite continued execution [15]. As a result, standalone fuzzing can miss critical vulnerabilities that reside in hard-to-reach design spaces. Second, traditional coverage metrics (e.g., line, branch, toggle) are not aligned with security properties such as confidentiality or isolation, thereby providing poor guidance for security goals. Moreover, even when fuzzing detects vulnerabilities, it cannot explain which internal root causes are responsible for them.

In this paper, we discuss how hybrid hardware fuzzing can address these limitations by integrating with traditional techniques like formal verification [16] and information flow tracking (IFT) [17]. First, we discuss how hybrid hardware fuzzers leverage the advances in formal verification techniques to generate seeds to guide fuzzers to explore hard-to-reach design spaces [11], [15], [18]–[23]. We then discuss how hybrid hardware fuzzers leverage IFT to enhance the confidentiality and isolation of computing systems by detecting timing and speculative vulnerabilities and guiding fuzzers to detect them faster [24]–[29]. In addition, we highlight open challenges that currently hinder the effectiveness and efficiency of hardware fuzzing and outline research directions suggested by the techniques reviewed in this paper.

II. Background

A. Hardware Fuzzing

Processor fuzzers operate by repeatedly generating instruction sequences and executing them on a processor-under-test (PUT). These instruction sequences are compiled into executable binaries and simulated or emulated to observe processor behaviors. A processor fuzzing framework consists of several core components, including a seed generator, a mutation engine, a coverage feedback collector, and a vulnerability detector. The seed generator generates a set of seeds, which are executed to gather behavioral information from a PUT. The coverage feedback collector collects coverage feedback during execution, indicating which parts of the design have been executed. Tests that lead to new or interesting behaviors are then modified to generate additional inputs, enabling the fuzzer to iteratively expand its exploration of design spaces.

Vulnerability detection within a fuzzer depends on the target class of vulnerabilities. For functional correctness, fuzzers often compare the behaviors of a PUT against a golden reference model (GRM) to identify mismatches as potential vulnerabilities in the PUT [8]. When targeting security vulnerabilities, however, detection becomes more challenging, as insecure behavior may not manifest as incorrect functional outputs. Instead, vulnerabilities may be revealed through microarchitectural side effects, such as timing differences or unintended information propagation.

B. Timing and Speculative Vulnerabilities

Timing vulnerabilities arise when variations in execution time depend on sensitive data values. These variations are typically caused by interactions between operands and microarchitectural features, such as caches, pipelines, or execution units. An instruction sequence may appear architecturally identical across executions, yet exhibit different execution times when operated on different data [30], [31]. Such behavior enables attackers to infer secret information by carefully observing timing differences. These vulnerabilities are difficult to detect because they do not violate functional correctness and often require analyzing execution time across multiple runs.

Speculative vulnerabilities represent another class of microarchitectural vulnerabilities that have become increasingly prominent as processor designs have grown more aggressive in performance optimization [32], [33]. Speculative execution allows processors to execute instructions before it is certain that they should be committed, reducing latency and improving throughput. When speculation is incorrect, the architectural effects of these instructions are flushed out across pipeline stages. However, microarchitectural side effects may persist, creating observable signals that can be exploited to leak sensitive information. These signals may propagate through a variety of channels [32], [34]–[36], including architectural state, microarchitectural resources, or timing behaviors, enabling attackers to infer data that should remain inaccessible.

C. Formal Verification

Formal verification relies on mathematical reasoning engines to exhaustively analyze system behaviors. At its core are SAT (Boolean Satisfiability) and SMT (Satisfiability Modulo Theories) solvers, which determine whether logical constraints derived from a design and its specification are satisfiable [5], [22]. These solvers enable *property checking*, where correctness or security properties (e.g., safety or liveness properties) are expressed as assertions and automatically verified against a PUT. If a property is violated, the solver produces a concrete counterexample trace, which precisely explains how the violation occurs. This capability makes formal verification effective for uncovering corner cases that are extremely unlikely to be triggered by simulation or random testing.

To manage complexity, several analysis techniques are commonly used. *Symbolic execution* represents inputs and internal states symbolically rather than concretely, allowing a single execution to reason about many possible behaviors at once by accumulating path constraints. *Bounded Model Checking (BMC)* further improves practicality by restricting analysis to executions up to a fixed number of steps or cycles, translating the bounded behavior into SAT/SMT instances. While BMC cannot prove properties beyond the chosen bound, it is highly effective at finding deep vulnerabilities and protocol violations within realistic time limits. Together, SAT/SMT solving, property checking, symbolic execution, and BMC form the foundational toolbox of modern formal verification, used by industrial standard formal tools like *JasperGold* [37]. The toolbox also provides strong correctness guarantees while highlighting the inherent scalability challenges that motivate hybrid approaches.

D. Information Flow Tracking

Information flow tracking (IFT) is a security analysis technique that monitors how sensitive information propagates

through a system [17]. The core idea is to label sensitive data (e.g., secrets, privileged values, or untrusted inputs) with tags (often called taints) and then track how these tags flow through computations, storage elements, and control logic. If tainted data reaches a security-critical sink, such as an output interface, timing-dependent state, or microarchitectural buffer, in a way that violates a security property, IFT reports a potential vulnerability. Because IFT reasons about data dependencies rather than functional correctness, it can detect confidentiality and isolation violations even when the system's functional behavior appears correct.

IFT is particularly valuable for hardware security because many vulnerabilities arise from implicit or transient data flows, such as speculative execution, buffering, or timing-dependent hardware behaviors. By tracking information at fine granularity (e.g., registers, wires, or microarchitectural structures), IFT can expose how secrets influence internal states that are later observable through side channels. However, IFT alone faces scalability and precision challenges such as over-tainting due to control-flow dependencies, so it is most effective when combined with complementary techniques (e.g., fuzzing) that drive execution into security-critical design spaces.

III. Hybrid Fuzzing Approaches

A. Overview

Hybrid fuzzing approaches combine hardware fuzzing with traditional techniques, such as formal verification and information flow tracking (IFT), to overcome limitations of standalone hardware fuzzing. Table I summarizes representative hybrid fuzzing techniques, illustrating how different techniques are integrated with fuzzing, along with their respective advantages, limitations, and classes of vulnerabilities detected. Fig. 2 provides an overview of the different ways these techniques can be integrated within a hybrid fuzzing framework. In addition, we describe the processor and System-on-Chip (SoC) benchmarks used to evaluate the effectiveness of these hybrid fuzzers, highlighting their different microarchitectural features.

B. Formal-Guided Fuzzers

Leveraging advances in formal verification, hardware fuzzing moves beyond purely random, coverage-driven exploration by using formal techniques to guide fuzzers into hard-to-reach design spaces. Collectively, they aim to improve the exploration of hard-to-reach design spaces and accelerate the discovery of security-relevant behaviors that are difficult to expose with standalone fuzzing.

A first group of works (*HyPFuzz* [11], *FormalFuzzer* [19], and *BMCFuzz* [21]) integrates fuzzing with formal verification techniques such as cover-property checking and bounded model checking (BMC). These approaches leverage formal engines to generate high-quality seeds for hard-to-reach regions and to guide fuzzing toward deep or constrained design states. *HyPFuzz* demonstrates that formal tools can significantly accelerate coverage but also highlights their high computational cost and sensitivity to cover-property formulation. *FormalFuzzer* emphasizes security-driven feedback by incorporating assertion templates and cost functions, shifting the focus from generic coverage to vulnerability-oriented exploration. *BMCFuzz* further analyzes prior hybrid fuzzers and argues that one-way integrations, where formal tools only feed fuzzing, fail to fully exploit the complementary strengths of both techniques.

A second group (*FuSS* [20], *SymbFuzz* [15], and *FMTC* [18]) applies symbolic execution in a selective and demand-driven manner. Rather than performing expensive symbolic execution exhaustively, these techniques invoke symbolic execution or symbolic simulation only when fuzzing stagnates or when specific uncovered regions are targeted. *FuSS* constrains symbolic exploration using information extracted from fuzzing to mitigate state explosion. *SymbFuzz* integrates symbolic execution into industrial UVM flows and relies on assertions to detect security violations that may not manifest as functional mismatches. *FMTC* combines symbolic execution with mutation-based fuzzing using a multiplexer-centric coverage metric to guide exploration.

The final group (*TargetFuzz* [23] and *PGTest* [22]) focuses on directed exploration and security-specific observability. *TargetFuzz* formalizes directed graybox fuzzing by selecting target design spaces and using SAT to generate seeds that explore those spaces. *PGTest* uses IFT to generate property inputs for formal tools and improves precision in detecting security violations, especially those involving unintended information propagation.

Overall, these fuzzers share several fundamental limitations despite their methodological differences. First, scalability remains a key challenge: integrations with formal methods or symbolic execution often incur high computational cost, suffer from state explosion, or are constrained by bounded analysis, making them difficult to apply to large, complex processors. Second, their effectiveness is tightly coupled to guidance quality, such as cover-property formulation, assertion templates, target design space selection, cost functions, or coverage metrics, so incomplete or coarse guidance can collapse multiple behaviors into a single signal and cause security-critical paths to be missed. Third, these approaches assume white-box visibility into the design, requiring full access to internal signals, states, and structural details. This assumption limits their portability across proprietary, third-party, or black-box components and reduces their generalizability to realistic industrial settings, where such internal visibility is often unavailable or restricted.

C. IFT-Guided Fuzzers

Leveraging IFT advances hardware fuzzing by moving beyond purely functional checking toward the detection of security-relevant vulnerabilities, especially in processors and SoCs where vulnerabilities arise from microarchitectural behaviors (e.g., speculation, transient data propagation, and timing variation). A common approach is to make fuzzing *security-aware* by introducing (i) detection strategies beyond output mismatches, (ii) guidance signals beyond generic code coverage, and (iii) observing microarchitectural behaviors rather than just monitoring high-level functional outputs.

TABLE I: Hybrid Fuzzing Techniques. N.A. refers "Not Applicable."

Fuzzer	Approach	Advantages	Limitations	Vulnerabilities found
FMTC [18]	Symbolic execution	Explore deeply nested multiplexers	Path explosion, narrow coverage scope, limited use of fuzzing context	N.A.
HyPFuzz [11]	Formal Property Checking	Explore corner cases	Path explosion, limited use of fuzzing context	Functional flaws
FormalFuzzer [19]	Formal Property Checking	Explore corner cases	Path explosion, limited use of fuzzing context	Functional flaws
SymbFuzz [15]	Symbolic execution	Explore corner cases, identify stalled coverage points	Path explosion, constraint complexity	Functional flaws
FuSS [20]	Symbolic execution	Explore corner cases, identify stalled coverage points	Path explosion, limited use of fuzzing context	N.A.
BMCFuzz [21]	Bounded Model Checking	Explore corner cases	Path explosion, bounded depth limitation, limited use of fuzzing context	Functional flaws
PGTest [22]	SAT/SMT	Explore corner cases, reduce redundant testing	Path explosion, high instrumentation overhead	Hardware trojans
TargetFuzz [23]	SAT	Explore corner cases, target security-critical regions	Path explosion, limited use of fuzzing context	N.A.
TaintFuzzer [24]	Map output changes to input port's mutation	Explore security-critical areas	High runtime overhead	Functional flaws
WhisperFuzz [25]	Track timing flows	Detect and locate vulnerabilities	High instrumentation overhead	Timing vulnerabilities
Specure [26]	Track microarchitectural flows	Detect and locate vulnerabilities	High instrumentation overhead	Speculative vulnerabilities
DejaVuzz [27]	Differential microarchitectural flows	Trigger transient windows efficiently	High instrumentation overhead	Speculative vulnerabilities
PhantomTrails [28]	Taint architecturally inaccessible memory	Detect transient data leaks	High fuzzing throughput degradation	Speculative vulnerabilities
MileSan [29]	Track microarchitectural flows	Discover both constant-time violations and speculative vulnerabilities	High instrumentation overhead	Microarchitectural vulnerabilities

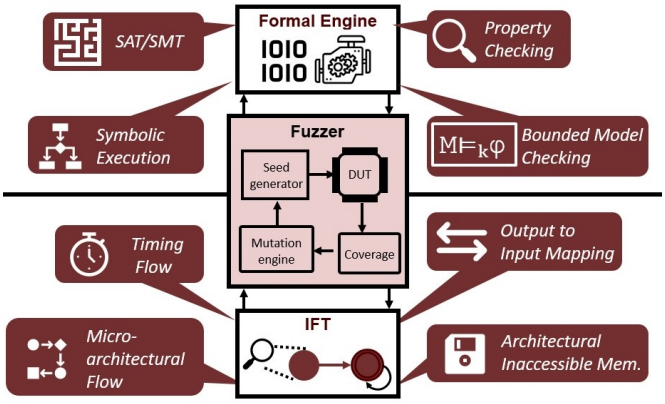

Fig. 2: Overview of hybrid hardware fuzzers and their integration with formal verification and information-flow tracking.

TaintFuzzer targets SoC-level vulnerabilities under gray-box constraints, leveraging mappings between inputs and outputs and security-oriented cost functions with FPGA-emulation feedback to generate hardware-centric mutations without requiring a GRM [24]. *WhisperFuzz* targets timing vulnerabilities, emphasizing root-cause localization and timing-behavior coverage by modeling sequential behaviors of RTL modules [25]. *Specure* statically extracts potential direct leakage channels from microarchitectural states to architectural states, then fuzzes processor-under-tests (PUTs) under speculative windows and prioritizes exploration via a leakage-path coverage metric aligned with speculative leakage [26]. *DejaVuzz* improves controllability and observability for speculative vulnerability detection by leveraging dynamic, swappable memory and differential information-flow tracking [27].

A complementary direction focuses on reducing overfitting and improving detection models. *Phantom Trails* proposes a fuzzer-friendly transient-leak detector that flags violations at their root causes, rather than requiring full end-to-end covert-channel exploits, integrates tainting into software simulations, and classifies leaks using flush signals, enabling template-free fuzzing [28]. *MileSan* explicitly diagnoses hardware/software overfitting in microarchitectural fuzzers and introduces a microarchitecture-agnostic leakage sanitizer based on differential hardware-software taint tracking, alongside *Taint-Aware In-Situ Simulation*-based program generation to avoid architectural taint explosion while supporting diverse test cases [29].

Overall, these works remain constrained by specialization and cost: most techniques target specific vulnerability classes (e.g., speculative or timing vulnerabilities) and therefore do not generalize to broader security properties without redesigning coverage metrics. Many approaches rely on white-box visibility, IFT modeling, or RTL-level instrumentation, making them sensitive to modeling accuracy and increasing integration and computational overhead, which limits scalability to large processors and SoCs. While newer fuzzers reduce overfitting, the efficiency of input generation and taint explosion remains a challenge, and root-cause localization can still be difficult

for complex, cross-microarchitectural feature interactions.

D. Evaluation Benchmarks

Hybrid hardware fuzzing techniques are evaluated across a diverse set of open-source processor and SoC benchmarks that expose different microarchitectural features and security-relevant behaviors. `CVA6` [38] and `Rocket Core` [39] are representative in-order `RISC-V` processors with deep pipelines and operating system support, making them suitable for stressing privilege transitions, exception handling, and memory system interactions, but without aggressive speculation or out-of-order execution. In contrast, `BOOM` [40] is an out-of-order `RISC-V` core with speculative execution, enabling the study of more advanced transient execution, timing leakage, and the interactions among microarchitectural features. OpenRISC designs such as `mor1kx` [41] and `OR1200` [42] provide 32-bit in-order pipelines with simpler control logic and limited speculation, offering long-validated designs for exploring corner cases in control flow and forwarding paths. SoC benchmarks including `Ariane SoC` [43], `OpenTitan` [44], `PiCoSoC` [45], and `VeeRwolf` [46] extend these cores with SoC peripherals, memory hierarchies, and interconnect components, enabling evaluation of fuzzing techniques at the SoC level. Finally, several studies incorporate standalone RTL modules and peripherals such as `UART`, `SPI`, `I2C`, `AES`, `FFT`, and control blocks to assess scalability and precision in non-CPU designs [22], [23]. Together, these benchmarks span varying levels of microarchitectural complexity, providing a comprehensive evaluation foundation for hybrid fuzzing techniques for diverse hardware vulnerabilities.

IV. Discussion and Future Directions

Despite the substantial progress made by hybrid fuzzing techniques, their limitations also reveal open challenges that remain largely unresolved. We summarize the challenges and outline research directions that target them in this section.

Challenge 1: Instruction semantics to trigger microarchitectural features. Instruction generation in hybrid hardware fuzzing remains largely disconnected from the microarchitectural behavior it is intended to exercise. Even when formal verification or IFT is used to guide fuzzing, the resulting tests remain instruction sequences whose effectiveness depends on triggering specific internal mechanisms, such as privilege checks, pipeline hazards, speculation paths, or timing-dependent control logic. In practice, commonly used coverage metrics do not indicate whether these mechanisms were activated. As a result, hybrid fuzzers may reach new coverage points or generate formally valid seeds that nonetheless fail to stress the microarchitectural behaviors relevant to security. This leads to wasted effort and slow progress, particularly when targeting deep control logic or rare state interactions. One future research direction is to develop fuzzing strategies that connect instruction semantics with internal microarchitectural behaviors.

Challenge 2: Limited observability from black-box settings. Currently, all hybrid fuzzing techniques rely heavily on white-box access to the designs, leveraging RTL-level visibility for formal verification, information flow tracking, and fine-grained coverage extraction. While this assumption holds during pre-silicon verification and open-source evaluation, a large fraction of real-world hardware deployed in the field operates as a black box, with internal structure and source code unavailable to the engineers. In such environments, direct access to internal signals, control logic, and microarchitectural state is not possible, significantly limiting the applicability of existing hybrid fuzzing approaches. As a result, an open challenge is how to achieve comparable effectiveness in black-box settings using only externally observable behavior such as architectural outputs, timing variation, or side-channel signals. Addressing this challenge requires new techniques that can infer or approximate internal hardware behavior without explicit RTL access, potentially by transferring strategies learned in white-box settings, leveraging architectural and microarchitectural models, or using learning-based approaches to correlate observable effects with hidden internal states.

Challenge 3: Optimal scheduling for hybrid fuzzers. The decision of when and how to invoke the formal tool versus the fuzzer is difficult in hybrid fuzzing. The runtime behavior of both components varies widely across designs and even across different regions of the same design. Some coverage points can be solved quickly by formal verification, while others cause the solver to stall or time out. At the same time, fuzzing progress is highly dependent on the quality of the seeds it receives from formal verification, since these seeds determine the subsequent mutation trajectory. Poor scheduling decisions can therefore waste significant time on hard or low-impact coverage points, while leaving the fuzzer under-guided or stalled. Similarly, naively selecting uncovered coverage points for formal targeting does not account for whether reaching a given point is likely to enable further coverage through mutation. As a result, existing hybrid fuzzers often rely on fixed heuristics that do not adapt well as coverage evolves. Improving effectiveness requires scheduling and point-selection mechanisms that adapt to observed tool behavior and coverage progress over time, rather than relying on static policies.

Challenge 4: Dependence on domain expertise for vulnerability models. Hybrid fuzzing frameworks are not self-contained solutions, but toolkits that rely heavily on expert knowledge to be effective. Unlike conventional hardware fuzzing, which can be applied with minimal understanding of the internal design, hybrid approaches require engineers to define threat models, select which signals or flows to track, configure formal properties, and determine how different analysis components should interact. These decisions strongly influence what classes of vulnerabilities can be detected and how efficiently exploration proceeds. In practice, this means that the effectiveness of a hybrid fuzzer often depends more on the domain expertise than on the underlying algorithms themselves. As a result, hybrid fuzzing is difficult to apply consistently across different designs and organizations, and successful configurations are rarely transferable. Reducing this reliance on manual expertise remains an open challenge

and may require automated techniques that can infer relevant security properties, analysis targets, or guidance strategies directly from the design, rather than relying on handcrafted vulnerability models.

V. Conclusion

Hybrid hardware fuzzing has emerged as a powerful response to the growing complexity of modern processors, addressing limitations of standalone fuzzing, formal verification, and information-flow tracking. By combining these complementary techniques, hybrid approaches enable more systematic exploration of hard-to-reach design spaces and more effective detection and localization of security vulnerabilities, including timing and speculative vulnerabilities that do not violate functional correctness. Results across diverse processor and SoC benchmarks demonstrate that hybrid fuzzing can significantly improve coverage and vulnerability detection compared to standalone fuzzing. However, important challenges remain, including weak connections between fuzzing approaches and microarchitectural behaviors, reliance on white-box observability and domain expertise, and inefficient scheduling between different techniques. Future research directions can focus on microarchitecture-aware input generation, improved support for black-box settings, and intelligent coordination strategies, ultimately making hardware vulnerability detection more scalable and practical for real-world designs.

Acknowledgment

Our research work was partially funded by Intel's Scalable Assurance Program, ONR Award #N00014-18-1-2058, the Lockheed Martin Corporation, NSF-Grant 2344914, SRC-3308.001, the SCALE Fellowship Program (M2402515), 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.

References

[1] 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," *USENIX Security Symposium*, pp. 213–230, 2019. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/dessouky

[2] "National Vulnerability Database," https://nvd.nist.gov/, 2023.

[3] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," *USENIX Security Symposium*, 2018. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[4] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," *IEEE Symposium on Security and Privacy*, 2019.

[5] Y. Kukimoto, "Introduction to Formal Verification," https://ptolemy.berkeley.edu/projects/embedded/research/vis/doc/VisUser/vis_user/node4.html, 2011.

[6] RISC-V, "RISC-V Webpage," https://riscv.org/, 2021.

[7] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing Hardware Like Software," *USENIX Security Symposium*, 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/trippel

[8] 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," *USENIX Security Symposium*, pp. 3219–3236, 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/kande

[9] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "DIFUZZRTL: Differential Fuzz Testing to Find CPU Bugs," *IEEE Symposium on Security and Privacy*, pp. 1286–1303, 2021.

[10] 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

[11] C. Chen, R. Kande, N. Nguyen, F. Andersen, A. Tyagi, A.-R. Sadeghi, and J. Rajendran, "HyPFuzz: Formal-Assisted processor fuzzing," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1361–1378. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/chen-chen

[12] C. Chen, V. Gohil, R. Kande, A.-R. Sadeghi, and J. Rajendran, "Psofuzz: Fuzzing processors with particle swarm optimization," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023, pp. 1–9.

[13] V. Gohil, R. Kande, C. Chen, A.-R. Sadeghi, and J. Rajendran, "Mabfuzz: Multi-Armed Bandit Algorithms for Fuzzing Processors," *Design, Automation & Test in Europe Conference & Exhibition*, 2024.

[14] M. Rostami, M. Chilese, S. Zeitouni, R. Kande, J. Rajendran, and A.-R. Sadeghi, "Beyond Random Inputs: A Novel ML-based Hardware Fuzzing," *Design, Automation & Test in Europe Conference & Exhibition*, pp. 1–6, 2024.

[15] S. S. Miftah, A. Srivastava, H. Kim, S. Wei, and K. Basu, "SymbFuzz: Symbolic Execution Guided Hardware Fuzzing," *IEEE/ACM International Symposium on Microarchitecture*, 2025.

[16] E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem *et al.*, "Handbook of Model Checking," vol. 10, 2018.

[17] W. Hu, A. Ardeshiricham, and R. Kastner, "Hardware Information Flow Tracking," *ACM Computing Surveys*, 2021.

[18] T. Li, H. Zou, D. Luo, and W. Qu, "Symbolic Simulation Enhanced Coverage-Directed Fuzz Testing of RTL Design," *IEEE International Symposium on Circuits and Systems*, pp. 1–5, 2021.

[19] N. F. Dipu, M. M. Hossain, K. Z. Azar, F. Farahmandi, and M. Tehranipoor, "FormalFuzzer: Formal Verification Assisted Fuzz Testing for SoC Vulnerability Detection," *IEEE Asia and South Pacific Design Automation Conference*, 2024.

[20] A. Jayasena, S. S. Nallapaneni, and P. Mishra, "FuSS: Coverage-Directed Hardware Fuzzing with Selective Symbolic Execution," *ACM Transactions on Embedded Computing Systems*, 2025.

[21] S. Shen, J. Liu, W. Feng, F. Song, and Z. Wu, "BMCFuzz: Hybrid Verification of Processors by Synergistic Integration of Bound Model Checking and Fuzzing," *IEEE/ACM International Conference On Computer Aided Design*, 2025.

[22] H. Sun, Z. Yang, S. Jin, and H. Xu, "Hardware Hybrid Testing Technology Based on Information Flow Path Guidance," *IEEE International Seminar on Artificial Intelligence, Networking and Information Technology*, 2025.

[23] R. Saravanan and S. M. P. Dinakarrao, "TargetFuzz: Enabling Directed Graybox Fuzzing via SAT-Guided Seed Generation," *Asia and South Pacific Design Automation Conference*, 2026.

[24] M. M. Hossain, N. F. Dipu, K. Z. Azar, F. Rahman, F. Farahmandi, and M. Tehranipoor, "TaintFuzzer: SoC Security Verification Using Taint Inference-Enabled Fuzzing," *IEEE/ACM International Conference on Computer Aided Design*, 2023.

[25] 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," *USENIX Security Symposium*, 2024.

[26] 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," *ACM/IEEE Design Automation Conference*, pp. 1–6, 2024.
[27] J. Xu, Y. Zhou, X. Zhang, Y. Li, Q. Tan, Y. Zhang, Y. Zhou, R. Chang, and W. Shen, "DejaVuzz: Disclosing Transient Execution Bugs with Dynamic Swappable Memory and Differential Information Flow Tracking Assisted Processor Fuzzing," *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2025.
[28] A. de Faveri Tron, R. Isemann, H. Ragab, C. Giuffrida, K. von Gleissenthall, and H. Bos, "Phantom Trails: Practical Pre-Silicon Discovery of Transient Data Leaks," *USENIX Security*, 2025.
[29] T. Kovats, F. Solt, K. Ceesay-Seitz, and K. Razavi, "MileSan: Detecting Exploitable Microarchitectural Leakage via Differential Hardware-Software Taint Tracking," *ACM SIGSAC Conference on Computer and Communications Security*, 2025.
[30] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," *The Cryptographers' Track at the RSA Conference.*, 2006.
[31] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ Flush: a fast and stealthy cache attack," *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 279–299, 2016.
[32] "CVE-2023-20593 Detail," https://nvd.nist.gov/vuln/detail/CVE-2023-20593, 2023.
[33] D. Moghimi, "Downfall: Exploiting speculative data gathering," *USENIX Security*, 2023.
[34] R. Z. *et al.*, "(m)wait for it: Bridging the gap between microarchitectural and architectural side channels," *USENIX Security*, 2023.
[35] P. B. *et al.*, "ÆPIC leak: Architecturally leaking uninitialized data from the microarchitecture," *USENIX Security*, 2022.
[36] B. G. *et al.*, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," *USENIX Security*, 2018.
[37] "Jasper RTL Apps," https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html, 2022.
[38] F. Zaruba and L. Benini, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 27, no. 11, pp. 2629–2640, Nov 2019.
[39] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," no. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html
[40] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine," *Fourth Workshop on Computer Architecture Research with RISC-V*, May 2020. [Online]. Available: https://www.semanticscholar.org/paper/SonicBOOM%3A-The-3rd-Generation-Berkeley-Out-of-Order-Zhao/41e9f0797fc92d36db32acee5fc4eb19fa3567ec
[41] OpenRISC, "mor1kx source code," https://github.com/openrisc/mor1kx, 2020.
[42] OpenRISC, "or1200 source code," https://github.com/openrisc/or1200, 2020.
[43] lowRISC Community, "Ariane: A 64-bit risc-v cpu," https://github.com/lowRISC/ariane, 2018.
[44] OpenTitan Project, "Opentitan: An open source root of trust," https://opentitan.org/, 2019.
[45] L. Griffiths, "Picosoc," https://lawrie.github.io/blackicemxbook/PicoSoC/PicoSoC.html, 2019.
[46] CHIPS Alliance, "Veerwolf: A risc-v soc platform," https://github.com/chipsalliance/VeeRwolf, 2020.