Core Fuzzing - A Versatile Platform for Security Verification

Alenkruth Krishnan Murali Ashish Venkat
University of Virginia
{alenkruth, venkat}@virginia.edu

I. Introduction

The growing complexity in modern systems has placed substantial limits on our ability to comprehensively assess threats and deploy security mitigations. As the industry is responding to an endless stream of hardware and software attacks, it is more clear than ever that, in addition to verifying software, verifying if the underlying microarchitecture is secure and free from exploitable vulnerabilities is critical to building reliable systems. Modern microprocessors contain several microarchitectural features that improve performance and save power. For example, techniques like speculative execution enable significant performance improvements, but also manifest as powerful tools in an attacker's toolbox allowing software checks to be bypasses, rendering code that was considered previously impermeable to software attacks vulnerable in the speculative domain. The rising complexity in modern microarchitectures not only necessitates significant time and effort dedicated to verification, but also invariably allows complex corner case vulnerabilities to escape. Therefore, reliable hardware security verification techniques with better coverage are increasingly considered to be a critical part of hardware design flows.

This work proposes Core Fuzzing, a microarchitectural approach to security verification where the key attributes of the processor microarchitecture are fuzzed at runtime to expose previously unseen execution paths and discover potential vulnerabilities in the microarchitecture and the software code that runs atop it. In software fuzzing, changing input datasets exposes bugs in control-flow paths previously not traversed. Similarly, by dynamically changing the underlying microarchitectural configuration, our *core fuzzing* framework seeks to expose previously unseen execution behaviors with vulnerable side effects, in any given program. In particular, given a design specification, our framework explores the entire microarchitectural configuration space to identify a set of valid configurations and fuzz the processor microarchitecture during runtime to observe execution flows violating the set of well-defined configurable security contracts. In contrast to existing hardware security approaches that rely on static checkers or multiple directed tests to expose microarchitectural bugs, *core fuzzing* is able to examine several distinct microarchitectural configurations for exploitable vulnerabilities through reconfiguration, while executing a single test program.

The key to core fuzzing is an out-of-order, superscalar, and dynamically reconfigurable/morphable RISC-V core that can sift itself through various configurations that feature different branch predictor designs, instruction windows, cache organizations, and functional unit configurations, among other microarchitectural knobs that are traditionally fixed at design time and hence remain inaccessible to security verification techniques that are typically deployed post the design phase. Although morphable and reconfigurable microarchitectures [4], [12], [14], [15], [27] were proposed to improve performance over traditional designs, by catering to diverse workloads, *core fuzzing*, to the best of our knowledge, is the first approach to leverage reconfigurable architectures for security verification.

In addition to the novel reconfigurable microarchitectural approach, we create a configurable security contract/policy enforced during verification. To this end, the reconfigurable RISC-V core is equipped with an oracle that observes program execution through performance counters and exposes instructions to privileged software to perform targeted mutation of microarchitectural parameters by setting control/status registers. The core also implements processor-wide Dynamic Information Flow Tracking (DIFT) [20], to track the flow of sensitive information during execution and flag violations or deviations from the security contract as the core morphs itself to different microarchitectural configurations and exposes previously unseen execution paths.

Our approach is expected to substantially accelerate the process of security verification and security-aware design space exploration in comparison to state-of-the-art strategies that are primarily simulation-based, while complementing existing verification methods by exposing new bugs and vulnerabilities in hardware and software.

II. Technical Approach

A. Design of the Reconfigurable Core

We use SonicBOOM [26], a superscalar, out-of-order, open source, RISC-V core written in Chisel to prototype our idea. We modify the design to make the core reconfigurable during runtime and add additional logic to tag and track information flow between individual hardware modules.

Figure 1 presents a representative block diagram of the SonicBOOM core with design modifications. By design, the SonicBOOM core is configurable during compilation, i.e., the microarchitectural parameters are fixed at compile-time. As shown in the figure, most microarchitectural modules within the core are reconfigurable.

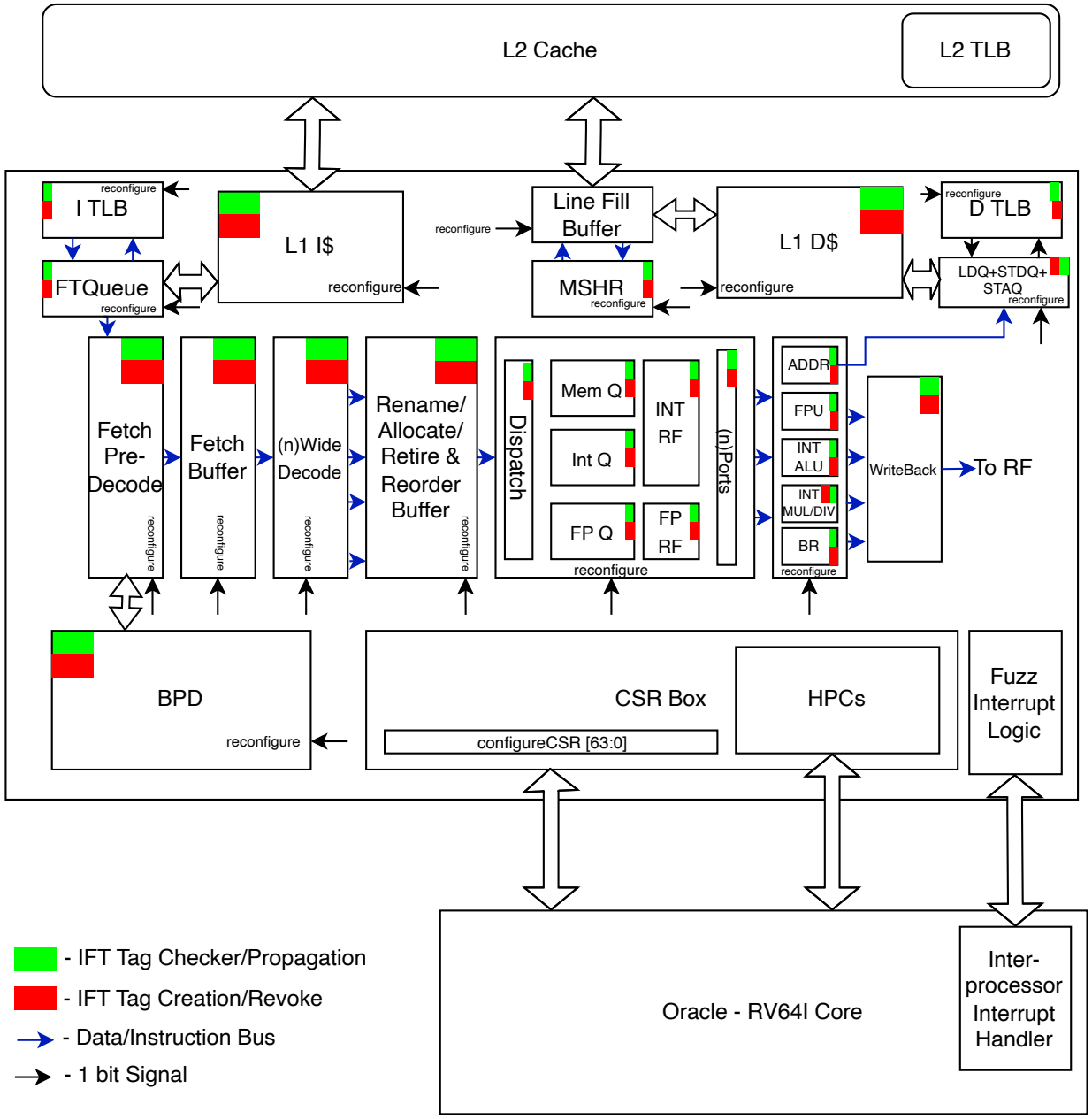

Fig. 1. Representative Block Diagram of the Core Fuzzing tool microarchitecture

This work redesigns the core to be reconfigurable during runtime with minimum overhead (performance, power, and area), with the goal of exposing new execution flows that may lead to vulnerable microarchitectural side effects. In particular, we enable the reconfiguration of the modules based on the values held in a new Control and Status Registers (CSR), *configureCSR*, which specifies the current microarchitectural configuration. Upon every reconfiguration, we will leverage existing and newly added Hardware Performance Counters (HPC) to track performance and security events of interest to drive targeted mutation and generate reports.

The brain of the *core fuzzing* tool is the Oracle. The Oracle is a 64-bit RISC-V core supporting privileged ISA specification and inter-processor interrupts. The Oracle and the reconfigurable BOOM core operate in unison in a master/subordinate arrangement, where the Oracle continuously monitors the HPCs to determine the execution flow pattern and then trigger the reconfiguration of the BOOM core to the next *valid* microarchitectural configuration as informed by a specific mutation strategy. The *validity* of a particular microarchitectural configuration is determined by a predefined specification of the design space.

1) Dynamic Information Flow Tracking: Dynamic Information Flow Tracking is implemented in the core fuzzing tool

by creating tags/taints for the information flowing between modules based on information privilege.

The green boxes on modules within the core in Figure 1 represent taint checkers. The taint checkers monitor the information flowing between module interfaces. At every module interface, they ensure that the output originating from a tainted input is always tainted. Additionally, they check if there are implicit flows between objects with different taints and flag them. Implicit flows are flagged based on the strictness of the security policy. Stateless channels of information leakage are converted to stateful channels through tagging, allowing implicit information leakage through stateless channels to be identified. When a security policy violation is detected at one or many checkers, it is propagated to one of the newly added HPCs, which are continuously monitored by the Oracle that raises a suitable exception.

The red boxes on the modules represent the initial tainting and untainting units. Based on the tainting policy (memory partitioning, privilege levels), the tainting unit will taint the data brought into the Cache. Assuming a case where a specific memory region is protected, the tainting unit will taint all instructions and/or data brought into the instruction and/or data cache from that memory region. Once the instruction and its associated data are tainted, the taint propagation units ensures that the taint is propagated until the instruction retires. Upon the retirement of a tainted instruction, the associated registers and data in the caches are untainted. This will often lead to propagating the untainting process throughout the pipeline requiring untainting units throughout the core. While the MSHRs, Buffers, and Queues are stateful units susceptible to side channels they do not need a special untainting unit since it is unlikely for a retiring instruction to be stored in a buffer or a queue.

B. Core Fuzzing Execution Flow

Figure 2 represents the execution flow of the Core Fuzzing tool. The green processes in the flowchart are carried out by the Oracle while the blue processes are carried out by the reconfigurable core.

The Oracle and the reconfigurable SonicBOOM core share a common memory region containing the test program. We start by booting the Oracle and loading the test program in the shared memory region. We then set up BOOM to execute the test program by resetting it to the initial state and clearing out the HPCs to avoid performance events from bootup polluting the results. Upon receiving a trigger from the Oracle, the BOOM core starts executing the test program in the chosen privilege mode. The MRET instruction helps switch privilege mode from the Machine mode, and SRET from Supervisor mode. After setting up additional Machine/Supervisor CSRs, the MRET instruction can be used to start execution in the Supervisor mode, and SRET can be used to start execution in the User mode. Depending on the testing scenario the program can also be executed in the machine mode. Once the test program execution begins, the BOOM core continuously updates the performance counters, while executing the test program. Whenever there is a trigger to reconfigure from the Oracle, the core reconfigures while imposing a small downtime before resuming execution. Meanwhile, the Oracle monitors the HPCs at fixed intervals and decide if a reconfiguration is necessary, and in particular, when there is a high chance of exposing a microarchitectural vulnerability.

A test program runs *N* number of times, as required, to successfully finish a fuzzing campaign. When no security policy violations are detected, the BOOM core triggers an interprocessor interrupt at the end of the campaign. The MEPC/SEPC CSRs store the Exception Program Counter address. When this matches with the address of the final instruction in the test program, the Oracle generates a final report listing the microarchitectural configurations, the observed information flows, and performance counter values.

But when there is a violation detected at any point of testing, the BOOM core will interrupt the Oracle and pause execution till the Oracle finishes collecting microarchitectural state (IFT violations at module interfaces, and Performance counter readings) and information about the violating instruction (PC). Once the Oracle stores the microarchitectural state and the PC in the shared memory region, it signals the BOOM core to continue execution. The final report, in this case, is expected to contain information about all policy violations and recommend the secure microarchitectural configuration(s) for the given security policy. Moreover, the report could also shed light on bugs present in the software that manifest as microarchitectural security policy violations.

III. Preliminary Results

In this section, we present results from a short core-fuzzing run. The reconfigurable core in this experiment has the ability to reconfigure its branch predictor from a TAGE predictor to a GShare predictor during runtime. We demonstrate the possibility of exposing previously unknown execution flows with vulnerable microarchitectural side effects through reconfiguration during runtime.

A. Threat Model

Since Core Fuzzing is a security verification tool targeted to identify vulnerable microarchitectural configurations during the design specification phase, our threat model includes all discovered and currently undiscovered attacks which exploit microarchitectural features. Core fuzzing uses Dynamic Information Flow Tracking to detect unintentional or malicious information flowing between microarchitectural modules during verification. By enforcing a strong security policy through information flow tracking, core fuzzing should be able to identify most, if not all attacks exploiting microarchitectural vulnerabilities.

The threat model also assumes that the DIFT mechanism and its implementation are fully trusted and do not contain any vulnerabilities. Moreover, the custom hardware performance counters and the Oracle which monitor and infer execution flows from the IFT mechanism are trusted. Having a trusted

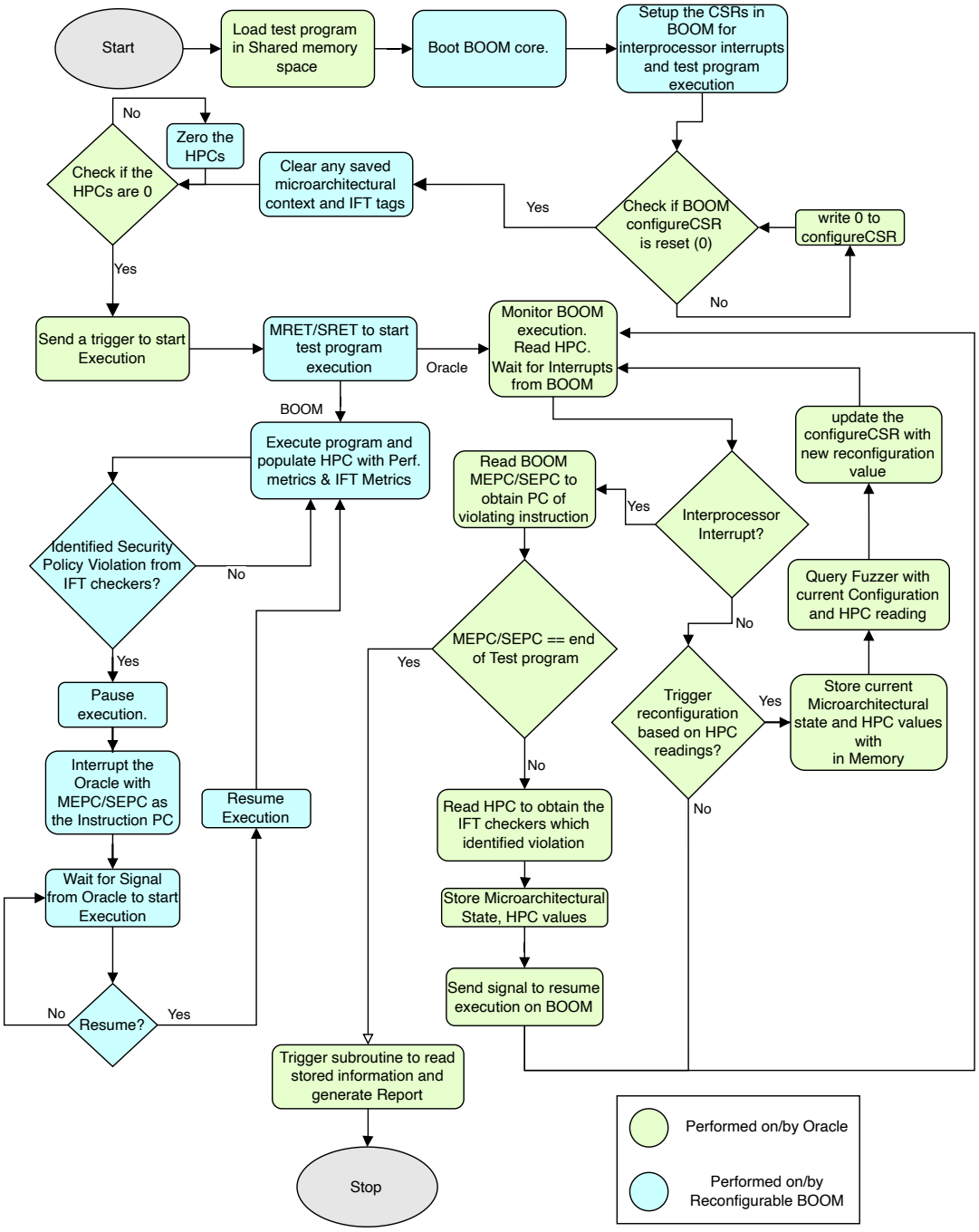

Fig. 2. Flowchart describing the Execution flow of the Core Fuzzing tool

Oracle, IFT mechanism, and monitoring mechanism validates the authenticity of the results obtained from the tool.

B. Spectre-v1 and Branch Predictor Reconfiguration

The reconfigurable core has two branch predictor configurations, a TAGE predictor and GShare Predictor. By default, a 6-bank TAGE Branch Predictor is used by the core. We add a 7^{th} bank with 256 entries, a 16-bit global history, and a 7-bit tag which function as the Branch History Table when the core reconfigures to use a GShare predictor. Currently, the reconfiguration is triggered by the test program after executing 4 Million cycles by updating a bit in a custom reconfiguration CSR. The secret is extracted by performing a FLUSH+RELOAD attack on the data cache which acts as a side-channel.

We fine-tune the Spectre-v1 [16] attack targeted for RISC-V and the SonicBOOM [19] core to be able to exploit a GShare predictor. This Spectre-v1 code is tuned to be not able to mistrain the default TAGE predictor in the core but

can mistrain the GShare predictor. Once the GShare predictor is mistrained, the core misspeculates on a conditional branch leading to leakage of secret in the new configuration as shown in Figure III-B.

As of this writing, we are working on implementing Information Flow Tracking within the core. Once the core is equipped with IFT, information leakage via stateful and stateless side channels during unknown execution paths, the misspeculation window in the presented example, will be flagged.

```
This emulator compiled with JTAG Remote Bitbang
    client. To enable, use +jtag_rbb_enable=1.
Listening on port 44209
m[0x0x80002790] = want(") =?= guess(hits,dec,char)
    1.(9, 34, ") 2.(1, 1, )
m[0x0x80002791] = want(S) =?= guess(hits,dec,char)
    1.(1, 1, ) 2.(1, 2, )
m[0x0x80002792] = want(e) =?= guess(hits,dec,char)
    1.(1, 1, ) 2.(1, 2, )

Triggering a reconfiguration
m[0x0x80002793] = want(c) =?= guess(hits,dec,char)
    1.(6, 99, c) 2.(1, 1, )
m[0x0x80002794] = want(r) =?= guess(hits,dec,char)
    1.(7, 114, r) 2.(1, 1, )
m[0x0x80002795] = want(e) =?= guess(hits,dec,char)
    1.(8, 101, e) 2.(1, 1, )
m[0x0x80002796] = want(t) =?= guess(hits,dec,char)
    1.(7, 116, t) 2.(1, 1, )
*** PASSED *** Completed after 14463565 cycles
[UART] UART0 is here (stdin/stdout).
```

Fig. 3. Results from reconfiguring the core during Spectre-v1 attack.

IV. Discussion

Reconfiguring the microarchitecture while running the test program will expose bugs or vulnerabilities in hardware and software that will not be apparent during regular execution on a fixed microarchitecture. The shown example executes for 14 Million cycles which is a short time to sift through multiple microarchitectural configurations. As the size of the test program increases, we have a large runtime window in which we can fuzz through several microarchitectural configurations exposing multiple unknown execution paths and information leakage channels due to changes throughout the pipeline. This can reveal unexpected interactions between software and specific hardware features, highlighting potential hardware vulnerabilities or inconsistencies. By observing the impact of microarchitectural changes on software execution, designers can identify and address hardware-related bugs, such as incorrect memory ordering or faulty instruction pipelines.

Altering the microarchitecture can change the timing, dependencies, and resource availability, potentially exposing race conditions, synchronization problems, or subtle software bugs previously masked by the original microarchitecture's behavior while providing an opportunity to test the software under diverse hardware scenarios. Additionally, it helps test the software's resilience against unexpected events and errors. By intentionally creating microarchitectural scenarios leading to cache evictions, pipeline stalls, or incorrect branch predictions, developers can evaluate how the software handles such events while exposing unseen bugs, vulnerabilities, exception handling issues, or potential security risks that may arise due to the altered microarchitecture.

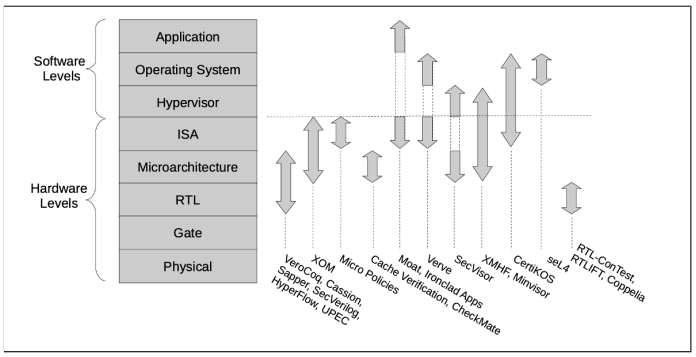

Fig. 4. Security Verification Approaches and the level of abstraction as described by [7]

V. Related Work

Generally, verification is performed at different abstraction levels: from the software (application) level to the hardware (RTL/Gate) level [1], [5], [21], [24] as shown in Figure 4. Formal verification techniques [2], [6], [8], [11], [13], [22], [23] have been widely used owing to their robustness, coverage, and ease of use. A mathematical model of the design is verified against a set of defined properties using a formal verification tool. Moreover, the formal verification tools sometimes require modeling the hardware in specific languages [3], [13], [22], [23], [25] that differ from the commonly used HDLs (Hardware Description Language). The survey by Erata, et al. [7] discusses in detail the multiple hardware security verification approaches available.

Some works [9], [18] do not verify the RTL design but use fuzzing-based approaches to generate targeted tests to verify the Black-box CPU designs for security policy [10], [17] violations. Unlike the previously mentioned static verification techniques that used formal methods and solvers, methods like *Revizor* [18] do not need access to the design specified in a Hardware Description Language. The security of the design can be verified either by simulations or by running tests on actual hardware. The tool runs a fuzzing campaign guiding successive rounds based on the results from the previous round to maximize coverage and reduce effort.

VI. Acknowledgement

This work was supported in part by Semiconductor Research Corporation (SRC) contract 2022-CT-3105 and NSF CCRI Grant CNS-2213700.

References

[1] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 2017, pp. 1691–1696.

[2] Axiomise, "Formalisa app - axiomise," Aug 2020. [Online]. Available: https://www.axiomise.com/riscv-formal-app/

[3] M.-M. Bidmeshki and Y. Makris, "VeriCoq: A Verilog-to-Coq converter for proof-carrying hardware automation," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2015, pp. 29–32.

[4] J.-C. Chiu, Y.-L. Chou, and P.-K. Chen, "Hyperscalar: A novel dynamically reconfigurable multi-core architecture," in *2010 39th International Conference on Parallel Processing*, 2010, pp. 277–286.

[5] S. Deng, D. Gümüşoğlu, W. Xiong, Y. S. Gener, O. Demir, and J. Szefer, "SecChisel: Language and Tool for Practical and Scalable Security Verification of Security-Aware Hardware Architectures," 2017, published: Cryptology ePrint Archive, Paper 2017/193. [Online]. Available: https://eprint.iacr.org/2017/193

[6] S. EDA, "Questa secure check - exhaustive verification of secure paths." [Online]. Available: https://eda.sw.siemens.com/en-US/ic/questa/formal-verification/secure-check/

[7] F. Erata, S. Deng, F. Zaghloul, W. Xiong, O. Demir, and J. Szefer, "Survey of Approaches and Techniques for Security Verification of Computer Systems," 2016, published: Cryptology ePrint Archive, Paper 2016/846. [Online]. Available: https://eprint.iacr.org/2016/846

[8] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz, "Processor hardware security vulnerabilities and their detection by unique program execution checking," 2018. [Online]. Available: https://arxiv.org/abs/1812.04975

[9] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures," in *Proceedings 2020 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2020/02/23018.pdf

[10] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1868–1883.

[11] C. Inc., "Jasper security path verification app." [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/security-path-verification-app.html

[12] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core fusion: Accommodating software diversity in chip multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 186–197. [Online]. Available: https://doi.org/10.1145/1250662.1250686

[13] Y. Jin and Y. Makris, "A proof-carrying based framework for trusted microprocessor IP," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 824–829.

[14] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt, "Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 305–316.

[15] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, "Composable lightweight processors," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. USA: IEEE Computer Society, 2007, p. 381–394.

[16] P. Kocher, J. Horn, A. Fogh, a. D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[17] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel, "Axiomatic hardware-software contracts for security," *International Symposium on Computer Architecture*, Jun. 2022, mAG ID: 4281986595 S2ID: 399b86ac034a889fa70d43b62fa0f773068ea211.

[18] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein, "Revizor: Testing Black-Box CPUs against Speculation Contracts," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 226–239, event-place: Lausanne, Switzerland. [Online]. Available: https://doi.org/10.1145/3503222.3507729

[19] M. Sabbagh, Y. Fei, and D. Kaeli, "Secure speculative execution via risc-v open hardware design," June 2021.

[20] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XI. New York, NY, USA: Association for Computing Machinery, 2004, p. 85–96. [Online]. Available: https://doi.org/10.1145/1024393.1024404

[21] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete Information Flow Tracking from the Gates Up," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: Association for Computing Machinery, 2009, pp. 109–120, event-place: Washington, DC, USA. [Online]. Available: https://doi.org/10.1145/1508244.1508258

[22] C. Trippel, D. Lustig, and M. Martonosi, "CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 947–960.

[23] C. Trippel, D. Lustig, and M. Martonosi, "Security Verification via Automatic Hardware-Aware Exploit Synthesis: The CheckMate Approach," *IEEE Micro*, vol. 39, no. 3, pp. 84–93, 2019.

[24] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A Hardware Design Language for Timing-Sensitive Information-Flow Security," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 503–516, event-place: Istanbul, Turkey. [Online]. Available: https://doi.org/10.1145/2694344.2694372

[25] T. Zhang and R. B. Lee, "New Models of Cache Architectures Characterizing Information Leakage from Cache Side Channels," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 96–105, event-place: New Orleans, Louisiana, USA. [Online]. Available: https://doi.org/10.1145/2664243.2664273

[26] 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.

[27] H. Zhong, S. A. Lieberman, and S. A. Mahlke, "Extending multicore architectures to exploit hybrid parallelism in single-thread applications," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 25–36.