RISC-V Microarchitecture Simulation State Enumeration

Griffin Knipe
knipe.g@northeastern.edu
Northeastern University
Boston, Massachusetts, USA

Derek Rodriguez
rodriguez.der@northeastern.edu
Northeastern University
Boston, Massachusetts, USA

Yunsi Fei
yfei@ece.neu.edu
Northeastern University
Boston, Massachusetts, USA

David Kaeli
kaeli@ece.neu.edu
Northeastern University
Boston, Massachusetts, USA

ABSTRACT

Security-related side-channel attacks, such as Spectre and Meltdown, exploit edge cases in microarchitectural execution. This class of attacks have been enabled by modern performance features common on many microprocessors. Identifying potential vulnerabilities during the design process requires functional verification capable of exploring all possible hardware states for security holes. Given the requirements of many of today's applications, the commercial markets are demanding that these side-channel exposures be remedied. Leveraging formal verification tools during the design process comes with significant engineering resource investment and these costs will increase as microarchitectural design features are introduced and complexity continues to grow.

We present our initial efforts to enable security-oriented verification during the microarchitectural design process for complex RISC-V microarchitectures. We have pursued this goal by integrating bounded microarchitectural verification with Akita, a state-of-the-art simulation framework designed for cycle-based performance evaluation. This approach will allow computer architects to evaluate and trade-off new and existing performance features, while evaluating their security implications using a single framework. This paper presents how to build a logical bridge between formal verification and Akita event-based modeling that enables this integration. We demonstrate a strategy on how to decompose Akita models into basic components that can be reassembled as an axiomatic model for formal verification.

ACM Reference Format:
Griffin Knipe, Derek Rodriguez, Yunsi Fei, and David Kaeli. 2021. RISC-V Microarchitecture Simulation State Enumeration. In *Proceedings of Fifth Workshop on Computer Architecture Research with RISC-V (CARRV '21)*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The discovery of covert side-channels in modern microarchitectures has highlighted that billions of existing computing devices are susceptible to leaking data across program boundaries. Current microprocessors are supposed to be designed to ensure safe execution for concurrently executing programs. Researchers have shown that speculative execution [12], fault resolution [13], and hierarchical memory [14] can provide attack surfaces for side-channel and covert-channel data leakage on today's microarchitectures. To ensure tomorrow's computing devices are capable of safeguarding sensitive information, architects must make security a major priority when designing new high-performance microprocessors.

In the wake of these discoveries, computer architects have developed microarchitectural defenses to be employed in future pipeline designs. Such defense techniques include hardening caches [10, 17, 24] and TLBs [9]. Unfortunately, there are other data stores that still leak information [4]. Some catch-all defenses, such as those that leverage dynamic information flow tracking (DIFT) [20, 25], allow the processor to refrain from committing tainted data until preceding speculation is resolved.

Despite these efforts, researchers continue to develop attacks for new microarchitectures, designed to resist prior known attacks [19]. Such cases are indicative of inadequacies in current design verification techniques to handle the complexity of modern performance features, which serve as attack surfaces that can be exploited. Prior work has described security verification tools capable of detecting these vulnerabilities before fabrication [16, 23]. We take a similar path, though our focus is to evaluate both security and performance together. We want to help computer architects catch vulnerable performance features during microarchitecture design. Our approach is to tightly integrate these verfication tools with current performance evaluation tools.

This paper presents our initial efforts to integrate a RISC-V model developed with Akita [22], a state-of-the-art microarchitectural simulator targeting performance evaluation, and CheckMate [23], a design verification toolset capable of detecting vulnerable execution patterns in hardware. This integration will provide computer architects with a holistic perspective of microarchitectural performance and security when testing new design features. This evaluation technique has the potential to mitigate side-channel attacks on future microarchitecture designs. This paper provides the following:

(1) a side-by-side comparison of the execution modeling abstractions provided by both Akita and CheckMate, identifying similarities which enable us to integrate these tools;
(2) methods for the static analysis of a RISC-V-based Akita model, to extract the information necessary to translate the model to the CheckMate DSL; and

CARRV '21, June 17, 2021, Virtual Event
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(3) a discussion of the remaining challenges to further improve our static analysis techniques, which, when completed, will enable seamless integration of these powerful tools.

2 BACKGROUND

2.1 Akita

Akita is an event-driven simulation framework introduced for the evaluation of cycle-based performance. The first microarchitecture that was modeled with Akita was a GPU modeling framework named MGPUSim, a multi-GPU simulator [22]. When using Akita, a simulation is comprised of a set of components, where each component represents a different element in the simulated system.

Akita components execute events to advance the simulation state. During an event, a component may update its internal state, schedule future events, or send outgoing messages to other components. Upon receiving a message, an Akita component will typically schedule a future event. Components will choose to schedule future events based on the combination of a few types of conditions. These conditions include receiving incoming messages from other components, as well as the current state of the component.

2.2 CheckMate

CheckMate utilizes formal methods to identify microarchitectural event orderings which may comprise transient execution side-channel attacks [23]. The user specifies a microarchitecture as a set of first-order relations, which describes all possible event orderings during execution. In general, an event corresponds to an instruction traversing a microarchitectural location. CheckMate can then enumerate all possible event sequences, according to the user provided constraints. A generalized event sequence is shown in the happens-before graph in Figure 1.

CheckMate searches all possible event sequences for the target vulnerability event sequence. Finding a match indicates that the microarchitecture is vulnerable to an attack. Using this method, CheckMate demonstrates its ability to detect Spectre, Meltdown, SpectrePrime, and MeltdownPrime in a 5-stage out-of-order microarchitecture [23].

3 YORI

Yori [11] is a RISC-V microarchitecture simulation which utilizes Akita as a simulation engine and targets SonicBOOM [27] as the reference microarchitecture. SonicBOOM is the reference architecture for this work, as it supports fundamental performance features, including transient execution. Prior work has exploited these performance features to reproduce a Spectre [12] attack on an FPGA model of SonicBOOM [7].

The Yori simulator is comprised of three Akita simulation components: 1) the Instruction-Fetch Unit (IFU), 2) the Execution Backend, and 3) the Memory Unit. The IFU and Memory Units are currently implemented as abstract models in the interest of focusing development time on detailed models in the Execution Backend. The IFU and Memory Units are responsible for delivering instructions to the pipeline and handling memory requests, respectively.

The Execution Backend consists of multiple SonicBOOM pipeline stage models that are responsible for instruction processing and retirement. These pipeline stages include Decode, Register-Rename, Reorder-Buffer, Issue, Register-Read, and Execute.

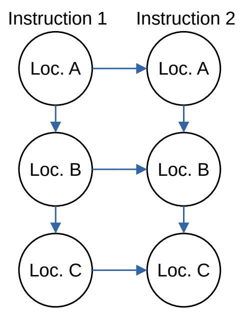

Figure 1: A CheckMate happens-before graph consists of a sequence of instructions, representing possible workload execution. Each instruction traverses microarchitectural locations as it executes. A happens-before node represents an (instruction,location) pair. The temporal relationships between each node in the graph is described by the directed edges, where the source node "happens-before" the destination node.

In its current configuration, Yori is capable of scalar out-of-order execution for 64-bit integer RISC-V binaries. Yori supports transient execution, where mispeculated instructions complete before the corresponding branch is resolved. However, the average number of transient instructions per mispeculated branch differs between Yori and SonicBOOM. Our ongoing work is focused on updates to Yori to ensure a faithful reproduction of execution on SonicBOOM.

In the following sections, we use Yori's IFU model to explain the bridge between Akita models and CheckMate. Figure 2 shows some of the possible events, the conditions that trigger these events, and the results of each event for the IFU model.

In this model, the Generate Packet Response (GenPacketRsp) event can be triggered by the reception of a Packet Request (PacketReq) message, or it can be scheduled during the Write Instruction Cache (WriteICache) and Cancel Request (CancelReq) events. The GenPacketRsp event may result in the IFU sending one of two message types. If the requested instruction exists in the Instruction Cache, then the component will send a Packet Response (PacketRsp) message during event execution. If the requested instruction does not exist in the Instruction Cache, the component will send a Memory Read Request (MemReadReq) message.

4 BRIDGING AKITA TO CHECKMATE

Given the flexible modelling capabilities of Akita, Akita simulation events can map directly to CheckMate happens-before nodes, making translation from Akita to CheckMate possible [11]. Doing so requires the user to implement simulation events at a granularity similar to the IFU shown in Figure 2. This translation effort requires a tool that can extract the fundamental modelling constructs of CheckMate from an Akita model, which include:

(1) Intra-instruction happens-before relationships
(2) Inter-instruction happens-before relationships

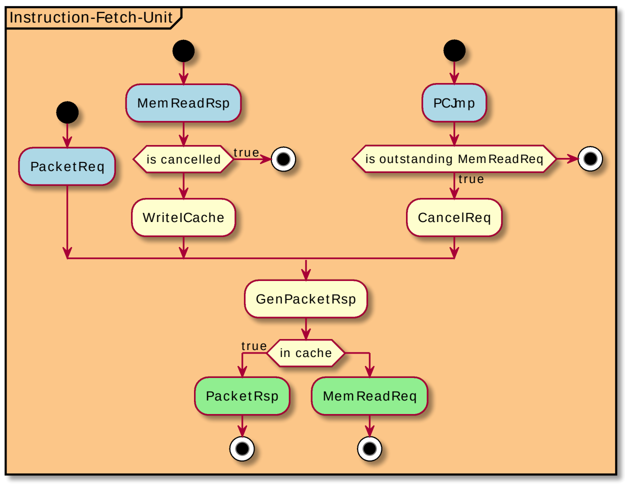

Figure 2: The IFU model utilizes the Instruction Cache and the Program Counter to deliver instructions that will be executed by the next stages in the pipeline. Facilitating this functionality requires the ability to read and write the instruction cache and to generate instruction packets for the the next stages. Each event is triggered by a combination of incoming messages and the component's state. An event may trigger future events or outgoing messages to other simulated components.

(3) The translation of imperative conditions to first-order relations of event orderings

Akita simulations are written using the Go programming language. We use Go's static analysis toolset [6] to extract the CheckMate modelling constructs from the Yori's model of SonicBOOM.

4.1 Akita Static Analysis Method

The first level of abstraction that we use to convert an Akita simulation to a CheckMate model is the logical model shown in Figure 2. Our static analyzer uses two functions as the entry point to each simulated component: 1) the incoming message notification function and 2) the event handling function. These functions are common to all simulated Akita components. Each component employs a unique implementation that is specific to the component's functionality. These two entry-points are the only functions that are externally triggered by the simulation engine in Yori's configuration of the Akita framework. Therefore, all lines of code that can update simulation state will be reachable by the static analyzer through these functions.

The incoming message notification (*NotifyRecv*) function is called when a component receives a message from another component during simulation. A generalized implementation is shown in Listing 1. The component can choose to schedule a future event, depending on the type of incoming message and a condition variable testing the current state of the component.

The abstract syntax tree (AST) for this function is shown in Figure 3. Each message type node, labeled "Type 1" or "Type 2" in the figure, corresponds to a case of the switch statement in Listing 1.

Listing 1: Simulated components are notified of incoming messages when this function is called by source of the message. The receiving component schedules a simulation event that corresponds to the incoming message and the current state of the component.

```
func NotifyRecv(port Port) {
    msg := port.Retrieve()
    switch msg.(type) {
        case msgTypeOne:
            if conditionIsTrue {
                engine.Schedule(eventTypeOne)
            } else {
                engine.Schedule(eventTypeThree)
            }
        case msgTypeTwo:
            if conditionIsTrue {
                engine.Schedule(eventTypeTwo)
            }
    }
}
```

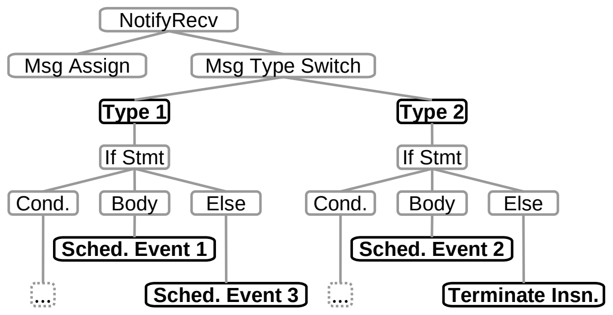

Figure 3: The AST for the *NotifyRecv* simulation component function, shown in Listing 1, is a graph representation of the program statements. The dashed-outline boxes represent sub-trees corresponding to the Go code which comprises the condition variable for the *if statement*. The AST nodes in bold correspond to event and message nodes in the IFU logical model, shown in Figure 2.

Each node contains a sub-tree of nodes corresponding to program statements that schedule future events. These sub-trees are visually similar to parts of the IFU logical model, as shown in Figure 2.

The message type nodes of the AST correspond to the input message nodes of the IFU, and are denoted as PacketReq, MemReadRsp, and PCJmp. The event scheduling nodes of the AST correspond to the event nodes of the IFU, shown as WriteICache, CancelReq, and GenPacketRsp. Therefore, the AST for the NotifyRecv function contains some parts of the information comprising the IFU logical model.

Note, that the AST sub-trees, identified by the dashed-outline nodes, represent program statements corresponding to the generation of the condition variables for *if statements.* These sub-trees are discussed further in Section 4.3.

Listing 2: The simulation engine calls this function to trigger the execution of the event that is passed as an argument. The component chooses the execution behavior based on the event type.

```
func Handle(ev Event) {
    switch ev.(type) {
        case eventTypeOne:
            handleEventOne(ev)
        case eventTypeTwo:
            handleEventTwo(ev)
    }
}
```

To complete the necessary information to formulate the IFU logical model, we can analyze the Akita event handling (Handle) function. The Handle function is triggered by the simulation engine when a scheduled event is ready to execute on the component. The generalized implementation, shown in Listing 2, calls a function corresponding to the type of event that is executing. Within these functions, the component may update its internal state, send messages to other components, or schedule future events. These actions are shown in the AST in Figure 4.

The event type nodes of the AST, shown as "Case 1" and "Case 2", correspond to cases in the switch statement in Listing 2. Each node contains sub-trees which correspond to the execution behavior for each event type. The event scheduling node and message sending node represent the results of the event execution that is observable by the simulation engine.

Each of these nodes can map directly to a node in the logical model for the IFU, as shown in Figure 2. A case node in the AST corresponds to a possible event in the IFU, such as WriteICache. According to Figure 2, the WriteICache event results in the scheduling of a GenPacketRsp event. This corresponds to an event scheduling node in the AST. By combining the information obtained from the static analysis of the NotifyRecv and Handle functions, we have the necessary information to generate a full logical model of the Yori IFU. This technique is extensible to all types of Akita components that model execution at a granularity similar to Yori's IFU implementation.

4.2 Intra-Instruction Happens-Before Relationships in Akita

Intra-instruction happens-before relationships describe the order of microarchitectural locations traversed by a single instruction in CheckMate [23]. In Akita, these locations correspond to simulation events that may occur through multiple Akita components. The information needed to construct this location traversal path is apparent in a system of logical models. Consider the Yori simulation, consisting of the IFU in Figure 2 and the Memory Unit in Figure 5.

We can connect the logical models through their corresponding input and output messages. For example, the MemReadReq output message of the IFU connects to the MemReadReq input message of the Memory Unit. This allows us to generate the intra-instruction event graph flowing through both components, as shown in Figure 6.

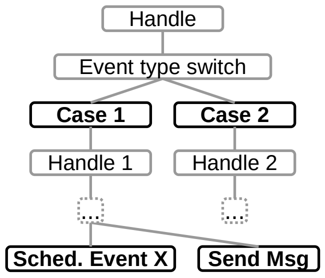

Figure 4: Simplified AST for the generalized *Handle* simulation component function shown in Listing 2. The dashed-outline nodes represent sub-trees corresponding to the Go code which determines the scheduling of further simulation events, or the sending of messages to other simulation components. The nodes in bold correspond to the event execution results that correspond to future events or outgoing messages in Figure 2.

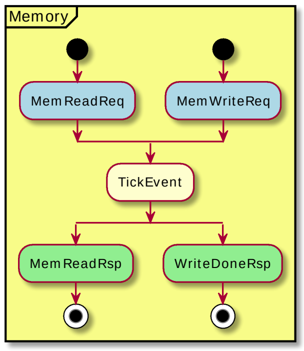

Figure 5: This basic memory unit implements only a generic Ticking Event (TickEvent) that handles both read and write requests. The logical model for this component shows both types of requests as possible triggers for the TickEvent and the respective response types as possible results.

This figure is conceptually equivalent to one instruction path in the generic CheckMate happens-before graph, shown in Figure 1.

4.3 Translation Challenges

Beyond the extrapolation of intra-instruction happens-before relationships from an Akita model, we must also be able to extract inter-instruction happens-before relationships and translate imperative conditions to first-order relational constraints on event orderings.

An inter-instruction happens-before relationship refers to the temporal relationship between happens-before nodes of different instructions [23]. These are shown as horizontal edges in the happens-before graph in Figure 1. This type of relationship is necessary to

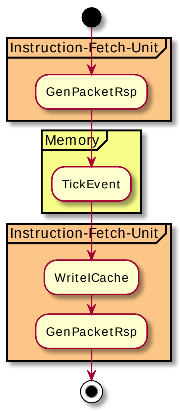

Figure 6: The logical models for the IFU and the Memory Unit can be connected by using simulation messages as an interface. This allows us to follow instruction traversal through multiple Akita components and construct a complete intra-instruction happens-before graph for a single instruction.

covey the out-of-order completion of instructions at a microarchitectural location, which occurs in microarchitectures vulnerable to side-channel attacks.

However, simulation conditions that dictate inter-instruction execution order in an Akita component are not encoded in the logical model in Figure 2. These conditions are represented in Listing 1 as the *conditionIsTrue* variable and by the dashed-outline AST sub-tree nodes in Figure 3 and Figure 4. These conditions are formulated by the current state of the simulated component, which is the combined result of the preceding simulation events. This problem is further complicated by the diversity of Akita components, each of which will formulate these conditions differently for each implementation.

As of now, we understand that this will require a robust static analysis technique that is capable of tracing the effects of preceding simulation events on the current state of the Akita component. We can simplify this problem by standardizing the formulation of these condition variables for all Akita components with a common software interface. However, this strategy must be implemented using a conservative approach to ensure that Akita's primary role as a flexible microarchitectural simulator is not impacted by the CheckMate translation.

5 RELATED WORK

Prior work has explored side-channel attack reproduction and defense characterization on microarchitectural simulators. Zsim is an event-driven microarchitectural simulator intended to scale in simulation performance for many-core systems [18]. Researches have used ZSim to test a defense against cache timing side-channel attacks [10]. FireSim [8] is an FPGA-accelerated simulator used in prior work to reproduce Spectre attacks [7] on the BOOM microarchitecture [5]. Gem5 [2] is a microarchitecture simulation framework capable of CPU-GPU simulation for multiple ISA's, including RISC-V. Prior work simulates Spectre attacks on an ARM microprocessor to evaluate the difference between the simulated attack and an actual attack on reference hardware [1].

EMSim [21] facilitates electromagnetic side-channel simulation given a detailed model of the target microarchitecture. Similar to our approach, this work provides the means to detect one class of side-channel attack using a cycle-based simulator. However, EMSim performs measurements at simulation runtime while we aim to provide attack detection through static analysis.

Alternative approaches to CheckMate for microarchitecuture vulnerability detection include Coppelia [26] and Speculator [16]. Coppelia translates RTL to C++ and uses Klee[3] to verify security constraints in the hardware-description language through symbolic execution. Unlike CheckMate, this tool does not target transient execution side-channel attacks. Speculator tracks the speculative execution of instructions on hardware using run-time performance counters [16]. This enabled researchers to discover two variants of Spectre-v2.

Prior work discusses the widening of the gap between microarchitecture design complexity and the capabilities of available design verification tools [15]. The authors present transaction graphs which convey the distinct states in execution and the conditions that cause an instruction to move from one state to the next. These transactions graphs are used for design verification and are structurally similar to our logical model shown in Figure 2.

6 CONCLUSION AND FUTURE WORK

In this paper, we have discussed our initial efforts to improve design-time microarchitectural security verification by integrating the CheckMate verification tool with the cycle-based performance simulator Akita. This effort is motivated by the constant threat of new transient execution side-channel attacks which exploit fundamental microarchitectural performance features, even on the newest microprocessors. We observed that Akita and CheckMate model execution using similar constructs which enables the automated translation from an Akita simulation to a CheckMate model.

To explore a solution for framework translation, we developed Yori, a cycle-based model of the SonicBOOM RISC-V microarchitecture. We chose to target a state-of-the-art open-source ISA and microarchitecture for this work so we can leverage the wealth of novel design features proposed by other researchers and evaluate their performance in the context of our integrated toolset.

The extraction of intra-instruction happens-before relationships from Yori is well understood. However, we continue to work on solutions to extract inter-instruction happens-before relationships. This involves the process of understanding the effects of preceding simulation events on the state of the modeled system using static analysis. This will allow us to translate the formulation of condition variables from an imperative Akita model to first-order relations between events in CheckMate.

Solving these remaining problems should provide the necessary information to facilitate complete translation between the frameworks. The resulting tool will equip the user with the ability to detect transient execution side-channel attacks within microarchitectures modelled with Akita.

REFERENCES

[1] Pierre Ayoub and Clémentine Maurice. 2021. Reproducing Spectre Attack with Gem5: How To Do It Right?. In *Proceedings of the 14th European Workshop on Systems Security* (Online, United Kingdom) *(EuroSec '21)*. Association for Computing Machinery, New York, NY, USA, 15–20. https://doi.org/10.1145/3447852.3458715

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[4] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 249–266. https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[5] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A. Patterson, and Krste Asanović. 2017. *BOOM v2: an open-source out-of-order RISC-V core.* Technical Report UCB/EECS-2017-157. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html

[6] Go. 2021. Go Analysis. https://pkg.go.dev/golang.org/x/tools/go/analysis. Accessed: 05-13-2021.

[7] Abraham Gonzalez, Ben Korpan, Jerry Zhao, Ed Younis, and Krste Asanović. 2019. Replicating and Mitigating Spectre Attacks on an Open Source RISC-V Microarchitecture. In *Proceedings of the 3rd Workshop with RISC-V (CARRV-3)*. Association for Computing Machinery, New York, NY, USA.

[8] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. 2018. Firesim: FPGA-Accelerated Cycle-Exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, Hoboken, NJ, 29–42. https://doi.org/10.1109/ISCA.2018.00014

[9] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *Proceedings of the 56th Annual Design Automation Conference 2019* (Las Vegas, NV, USA) *(DAC '19)*. Association for Computing Machinery, New York, NY, USA, Article 60, 6 pages. https://doi.org/10.1145/3316781.3317903

[10] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A Defense against Cache Timing Attacks in Speculative Execution Processors. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (Fukuoka, Japan) *(MICRO-51)*. IEEE Press, Hoboken, NJ, 974–987. https://doi.org/10.1109/MICRO.2018.00083

[11] Griffin Knipe. 2021. *Unifying Performance and Security Evaluation for Microarchitecture Design Exploration.* Master's thesis. Northeastern University. Unpublished.

[12] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, Hoboken, NJ, 1–19.

[13] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[14] Yangdi Lyu and Prabhat Mishra. 2018. A survey of side-channel attacks on caches and countermeasures. *Journal of Hardware and Systems Security* 2, 1 (2018), 33–50.

[15] Yogesh Mahajan, Carven Chan, Ali Bayazit, Sharad Malik, and Wei Qin. 2007. Verification Driven Formal Architecture and Microarchitecture Modeling. In *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE '07)*. IEEE Computer Society, USA, 123–132. https://doi.org/10.1109/MEMCOD.2007.371235

[16] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. 2019. Speculator: A Tool to Analyze Speculative Execution Attacks and Mitigations. In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico) *(ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 747–761. https://doi.org/10.1145/3359789.3359837

[17] Gururaj Saileshwar and Moinuddin K. Qureshi. 2019. CleanupSpec: An "Undo" Approach to Safe Speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 73–86. https://doi.org/10.1145/3352460.3358314

[18] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer architecture news* 41, 3 (2013), 475–486.

[19] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 753–768. https://doi.org/10.1145/3319535.3354252

[20] Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. 2019. ConTExT: Leakage-Free Transient Execution. *CoRR* abs/1905.09100 (2019). arXiv:1905.09100 http://arxiv.org/abs/1905.09100

[21] Nader Sehatbakhsh, Baki Berkay Yilmaz, Alenka Zajic, and Milos Prvulovic. 2020. EMSim: A Microarchitecture-Level Simulation Tool for Modeling Electromagnetic Side-Channel Signals. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Hoboken, NJ, 71–85.

[22] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. 2019. MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 197–209. https://doi.org/10.1145/3307650.3322230

[23] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2019. Security verification via automatic hardware-aware exploit synthesis: The CheckMate approach. *IEEE Micro* 39, 3 (2019), 84–93.

[24] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (Fukuoka, Japan) *(MICRO-51)*. IEEE Press, Hoboken, NJ, 428–441. https://doi.org/10.1109/MICRO.2018.00042

[25] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 954–968. https://doi.org/10.1145/3352460.3358274

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

[27] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanović. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. In *Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*. Association for Computing Machinery, New York, NY, USA.