Position Paper: A Case for Exposing Extra-Architectural State in the ISA

Jason Lowe-Power
University of California, Davis
Computer Science
jlowepower@ucdavis.edu

Venkatesh Akella
University of California, Davis
Electrical and Computer Engineering
akella@ucdavis.edu

Matthew K. Farrens
University of California, Davis
Computer Science
farrens@cs.ucdavis.edu

Samuel T. King
University of California, Davis
Computer Science
kingst@ucdavis.edu

Christopher J. Nitta
University of California, Davis
Computer Science
cjnitta@ucdavis.edu

ABSTRACT

The recent Meltdown and Spectre attacks took the community by surprise. Rather than exploiting an incorrect implementation of the ISA, these attacks leverage the undocumented implementation-specific speculation behavior of high-performance microarchitectures to affect the *extra-architectural* state of the machine (e.g., caches).

Inspired by these novel speculation-based attacks, we argue it is time to rethink the traditional ISA layers. Programmers and security professionals need a framework to reason about the effects of speculation and other microarchitectural performance optimizations. We propose judiciously extending the ISA to include the extra-architectural state so that an ISA implementation either completely squashes all system state changes caused by mis-speculated instructions or the potential changes are rigorously documented. We hope this new framework will give architects and security researchers tools to reduce the likelihood of future surprise vulnerabilities.

CCS CONCEPTS

• **Security and privacy** → **Hardware security implementation**; • **Computer systems organization** → **Superscalar architectures**;

KEYWORDS

speculation, security, ISA

ACM Reference Format:
Jason Lowe-Power, Venkatesh Akella, Matthew K. Farrens, Samuel T. King, and Christopher J. Nitta. 2018. Position Paper: A Case for Exposing Extra-Architectural State in the ISA. In *HASP '18: Hardware and Architectural Support for Security and Privacy, June 2, 2018, Los Angeles, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3214292.3214300

1 INTRODUCTION

Security has become a first-order design constraint. The software development community has already recognized this, and it is now clear computer architecture must also design systems with security in mind. However, this requires revisiting decades-old hardware development patterns.

Many of the architectural innovations that have driven the increase in compute capability for the past 50 years were designed ignoring their security implications. For instance, caches [3, 21], branch predictors [4, 11], and even pipelines [10] can be used as communication side-channels, and recently Meltdown [19] and Spectre [18] showed how attackers can leverage processors' speculative execution hardware to induce applications to leak secret information, without even needing to exploit any "bugs" in the hardware. Meltdown and Spectre cause instructions to execute that should not execute. However, the architectural state of these processors is never incorrect *by today's definition of the ISA*. Instead, these transiently executed instructions affect structures outside of the current definition of the architectural state causing information leakage.

There has been significant work providing hardware support for making software more secure (e.g., Intel TXT and SGX [7, 12], ARM TrustZone [9], and many others), providing software support for using insecure hardware (e.g., microcode changes [1] and runtime support [15, 16]), and closing hardware vulnerabilities (e.g., side-channels [10, 28], network processors [29], and information flow tracking [23, 24]). Still, computer architects design for performance

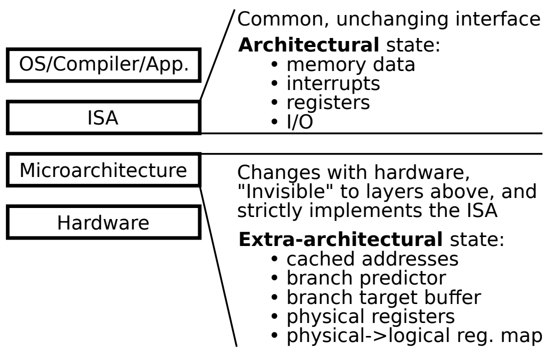

Figure 1: Architectural and extra-architectural state

first and rarely consider the security implications of performance optimizations, since it is unlikely that consumers will be happy if they are told they must give up performance enhancements in order to make their machines more secure. Recent events have demonstrated that this has to change, and that architects must address this issue immediately.

Figure 1 shows the incredibly successful layered architecture of modern systems. This approach allows software developers and hardware engineers to innovate separately at a much faster rate than they could if they synchronized at every step. However, these recent attacks have shown some of the "invisible" optimizations in the microarchitecture layer are in fact visible and can be exploited.

By strictly separating the programmers from the hardware implementation, they lose visibility into potential side-effects of instruction execution. The architectural "state" of the machine has long been defined as all the information necessary to correctly save and restart a process. Unfortunately, this definition is inadequate from a security standpoint. Ideally, for a given hardware implementation of an ISA, a security researcher needs to understand all the changes to *any* hardware structure shared by one or more processes. Currently, there is no way for security researchers to do this because many structures (caches, buffers, etc.) are not part of the architectural state. Thus, there needs to be a better *framework* to enable reasoning about potential security vulnerabilities.

We argue it is time to rethink the definition of the ISA to include both the architectural state and the *extra*-architectural state (see Figure 1). This will require the hardware manufacturers to *either* explicitly identify all of the various structures in the system that may be affected by a particular instruction or ensure the instruction does not modify *any* structures in an undefined way. Security experts and programmers can then use this information to design safer and more secure applications.

In this paper, we focus on speculation-based attacks like Meltdown and Spectre as a case study. Current architectures use three techniques to ensure speculation does not affect the architectural state: preventing, undoing, and specifying speculative architectural state changes. We discuss these techniques, examples of extending these ideas to the extra-architectural state, and potential changes to ISA definitions to reduce the likelihood of future speculation-based attacks.

2 SPECULATION-BASED ATTACKS

Meltdown and Spectre induce the processor to execute transient instructions – instructions that execute during mis-speculation and are then invalidated/squashed – which affect system state that is *not part of the architectural definition.* We define the system state that is not part of the architectural definition as the *extra-architectural state* (Figure 1). The extra-architectural state includes any structure that is potentially visible to other processes (addresses present in the cache, the branch predictor state, etc.).

In the case of Meltdown and Spectre, the transient instructions cause changes to the addresses in the L1 cache. The addresses held in the cache are not part of the architectural state, since caches are logically invisible to the programmer. However, changes to the *extra-architectural* state are detectable by an attacker because the cache state is shared between multiple actors.

For these extra-architectural state attacks to be successful two things must occur:

(1) The attacker causes a modification to the extra-architectural state and this modification is based on a secret.
(2) The attacker can perceive the extra-architectural state change.

The code below shows an example code snippet that is vulnerable to the Spectre attack [18], specifically the bounds check bypass attack (Spectre-V1).

```
if (x < array1_size)
    array2[array1[x] * 512];
```

In this example, the branch predictor is primed to assume the branch is not taken (`x < array1_size` is `true`). Then, the attacker uses a specially crafted `x`, the branch is predicted not taken, and the next line (`array2[array1[x] * 512];`) is transiently executed, reading a value outside of the bounds of `array1` and accessing memory (`array2[...]`) based on the secret data value. While the speculation hardware eventually correctly recovers and squashes the transient instructions, *they have already affected the extra-architectural state.* Specifically, the address in `array2` that is evicted from the cache depends on the *value* of the out-of-bounds accesses, leaking secret information.

This attack meets the two requirements defined above. First, the attacker causes a modification to the extra-architectural state (i.e., the addresses in the L1 cache). Then, this change is perceived by the attacker by using a flush-and-probe technique to detect the currently cached address.

Meltdown similarly leverages the transient instructions that execute under a mis-speculation, although it exploits the fact that some speculative implementations delay memory address permission checks and allow transient instructions to access privileged addresses.

Proposed mitigations

Mitigations for Meltdown and Spectre were announced at the same time as the vulnerabilities were made public. Kernel page table isolation (KPTI) mitigates meltdown by separating the kernel page table from the user-mode page table [14]. The x86 ISA defines that it is illegal to speculate past a write to the page table pointer (CR3). Thus, a correct implementation of the ISA will prevent any speculative state change from loads to a higher privilege level blocking the Meltdown behavior.

There are two documented versions of Spectre with different mitigation strategies. First, to mitigate the bound check bypass attack described above (Spectre-V1), Intel, ARM, and AMD have proposed adding new instructions or changing current instructions to ensure the code after the branch does not execute speculatively [1, 2, 13]. To mitigate Spectre-V1, programmers (possibly with the help of static analysis techniques) must insert an `LFENCE` instruction after each bounds check to prevent any following instructions from being speculatively executed. This is an example of exposing the extra-architectural state to the ISA and allowing systems programmers and security experts to reason about how to prevent attacks.

Second, to mitigate the branch target injection attack (Spectre-V2), Intel and ARM have proposed adding instructions to flush the branch target buffer (BTB) [1, 13]. Then, the system can proactively flush the BTB if it believes there is a chance to attack the program.[1]. Again, this mitigation technique exposes some of the extra-architectural state to the systems-level programmer by adding new instructions and new machine-specific configuration registers.

There are two new versions of these attacks, MeltdownPrime and SpectrePrime [25] that use cache coherence messages as the side channel instead of the shared L1 data cache. The current proposed mitigation techniques also apply to these versions of the speculation attacks since the mitigation techniques focuses on limiting when the processor can speculate, not on the specific side channel used.

These mitigations allow low-level software developers (e.g., systems and compiler writers) to modify their software to protect it from Meltdown and Spectre. However, the behavior that was triggered by these attacks is still not formally documented. There is no reason to think there are not other, similar, speculation-based attacks that may be found in the future. For example, might it be possible to trick the prefetcher into reading data for an attacker? As long as instructions change the extra-architectural state in ways that are undocumented but observable, vulnerabilities will exist.

3 RETHINKING THE ISA

Out of order superscalar architectures employ a variety of techniques to prevent possible incorrect program execution that can arise from speculation. These techniques fall three main categories.

(1) *Prevent speculative state change.* Example: Stores to memory. Once the processor issues a write to memory, it may be prohibitively difficult for the write to be undone. Thus, all speculative stores are inserted into a store queue and are not issued to memory until the store instruction is committed (i.e., is not speculative).

(2) *Undo speculative state change.* Example: Register writes. Most speculative processors use register renaming to reduce data hazards. Used in conjunction with a reorder-buffer (ROB), this allows the processor to undo any register writes that occur during a mis-speculation.

(3) *Specify speculative state change.* Example: Relaxed consistency. There are cases where performance can be significantly improved if the non-speculative semantics of the application are relaxed. One example is the memory consistency model that specifies the possible interleavings of memory accesses from multiple processors. By allowing a non-sequential interleaving, processors can use load and store buffers, significantly increasing performance.

We argue these techniques must be extended to cover both the architectural state and the extra-architectural state. Below, we discuss an example of how to accomplish this goal.

[1] There are software techniques to isolate indirect branches from speculation proposed as well (e.g., "retpoline" [27]).

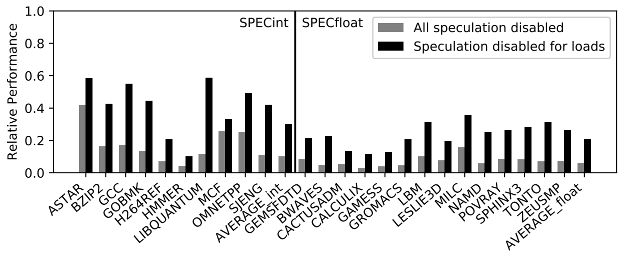

Figure 2: Performance for disabling all speculation and disabling speculative loads compared to full speculation (higher is better).

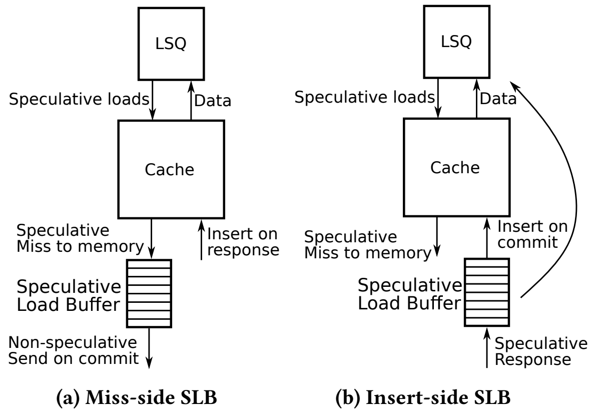

Figure 3: Two possible speculative load buffer designs.

3.1 Prevent speculative state change

3.1.1 Straw man: Disabling speculation. The simplest way to bring the ISA definition in line with the microarchitecture implementation is to remove speculation. We performed a preliminary simulation to estimate the performance effects of removing speculation from modern out-of-order processors. We used the gem5 architecture simulator [8] and ran each SPEC workload for 500 million instructions after warming up for 200 million instructions.

Figure 2 shows the performance hit for disabling speculation is dramatic. We disabled speculation by marking every instruction as serializing which forces all previous instructions to commit before issuing the current instruction. Disabling speculation in general causes an average 17× performance hit for the SPECfloat workloads and a 10× performance reduction for SPECint.

We also investigated disabling speculation only for load instructions. This would neutralize the Meltdown and Spectre attacks, but may not be strong enough to eliminate all speculation-based attacks. Disabling speculation only for loads has less of an impact on performance, although we still see on average a 4.9× slowdown for SPECfloat and a 3.3× slowdown for SPECint.

3.1.2 Hardware-only change: Speculative load buffer. Figure 3 shows two systems with the proposed speculative load buffer. The SLB on the miss side of the cache (Figure 3a) prevents speculative loads from affecting the extra-architectural state of the cache. Any loads that miss in the cache are not allowed to access the rest of the memory system until they are no longer speculative, similar to a store buffer. The SLB can be placed behind the cache and is only used on cache misses since cache hits do not change the extra-architectural state of the cache (i.e., the addresses currently present in the cache)[2].

We present the SLB as an example of a hardware change that eliminates speculative extra-architectural state changes. Unfortunately, while the SLB may increase security transparently to the programmer, like eliminating speculation, it likely will cause performance degradation. We are currently evaluating the performance impact of the SLB.

3.2 Undo speculative state change

A higher performing solution to Meltdown and Spectre than disabling speculation or the miss-side SLB is to ensure that the extra-architecture state change caused by loads is undone if the load is mis-speculated. Traditionally, architects have not worried about the impact of speculative loads since the state is only affected when the value is written into an architectural register, and speculative loads can be undone by simply squashing the register write. However, loads affect the cache state, which is part of the extra-architecture state. Therefore, we also need to have the ability to "undo" the cache insertions.

Figure 3b shows an insert-side SLB. The insert-side SLB logically extends the idea of the reorder buffer (ROB) into the cache. This speculative load buffer is modeled after the prefetch buffer that is already part of many cache designs. For speculative loads, instead of inserting new lines into the cache as soon memory responds, the data is inserted into the speculative load buffer. Only when the instruction commits is it allowed to move into the cache. We can rollback mis-speculated loads by invalidating entries in the speculative load buffer whenever the corresponding entry is invalidated in the ROB. In order to ensure that the speculative load buffer is not also part of the extra-architectural state, it must not be shared between processes. To cover lower level caches and cache coherence side channels, we can either further extend this idea, disallow sharing of lower level caches (e.g., via cache partitioning), or create a special prefetch-like memory request that does not invalidate other cache's cache lines.

The speculative load buffer does not eliminate all cache side channels, but it is an example of how extending the ISA definition to include extra-architectural state allows designers to reason about ways to reduce the vulnerability space.

3.3 Specify speculative state change

We believe a more general approach to closing these speculative execution vulnerabilities is to have the ISA explicitly specify the effects of an instruction on both the architectural state and the extra-architectural state. This includes the possible instruction interleaving allowed by a specific microarchitectural implementation of speculation and any effects on shared structures (caches, branch predictors, etc.).

[2]In this design, speculative loads could affect the LRU information in the cache state. To prevent this extra-architectural state change, the LRU information should only be updated when the load is no longer speculative.

The state of ISAs today is similar in spirit to the state of ISAs before formal memory consistency models. For many years, ISAs used prose and details of the microarchitectural implementation to incompletely describe their memory model [22]. However, ISA creators have realized that formality is important and have since provided formal memory models [26].

Similarly, we believe that speculation should be formally specified. Like the inflection point of going from uniprocessors to multiprocessors, we are at an inflection point today where security is now a first-order design constraint. Not all applications require rigorous security guarantees, but for the applications that do require it, the ISA should provide all of the information necessary.

Manufacturers are moving in this direction in an *ad hoc* fashion. For instance, Intel's [1], AMD's [2], and ARM's [13] proposed response to Spectre is to implement new instructions which limit speculation on the surrounding instructions. These instructions either cause explicit serialization of the out-of-order pipeline or explicitly flush certain structures (e.g., the branch target buffer) to force non-speculative program execution. This is currently specified using prose and requires understanding the underlying microarchitectural implementation.

The main challenge in exposing the extra-architectural state in the ISA is providing the right interface between the hardware and the programmer. We want an interface that is general. A general interface allows implementers of the ISA to innovate, enables compatibility between different hardware generations, and may simplify programming. However, we also want an interface which contains enough detail so security-minded applications are not required to put a fence between every single instruction.

It is not obvious how to apply these contradictory requirements to extra-architectural state. For instance, it is likely too specific for the ISA to specify the number of sets and ways of the cache. Instead, ISAs could use a model like the value in cache lifetime (ViCL) which abstracts away the specific cache design and only represents cache insertions and evictions [20]. ViCL is currently used for validating memory consistency models and was used to discover new types of speculation attacks [25]. Similar models may be appropriate for other caching structures that are part of the extra-architectural state (e.g., branch predictor).

4 RETHINKING THE SYSTEM

When we rethink the ISA design, we must also rethink the rest of the system's software including the operation system and the compiler. There are many possible ways to combine the three techniques described above when extending the ISA to cover the extra-architectural state and these combinations result in different system design tradeoffs. We can continue the traditional architecture technique of transparency by restricting or undoing speculative state change which requires minimal system changes to provide security. Or, we can potentially use a simpler hardware implementation and push security up the stack to the system software and compiler developers (e.g., Singularity [17] and the Factored Operating System (fos) [30]). Pushing security up the stack requires specifying the potential extra-architectural state changes so that these developers can reason about their software. We are currently investigating combinations of these approaches and how they interact with hardware, software, operating system, and compiler design.

Not all applications have strict security requirements and applications with more lax requirements should be able to use all of the potential performance of speculative architectures. One potential approach is for the ISA to expose multiple security levels to the operating system to use for different applications. This could be implemented by adding more security rings to the processor. Currently, most systems use three levels of security rings: user-mode, kernel-mode, and hypervisor-mode. Another ring could be added for applications that process secret data, and the hardware would guarantee that there is no speculative state change while in this ring (by disabling speculation or disabling caches while the processor is executing in the "secret" ring, for example). This would potentially limit the most negative performance impacts to applications that do not have rigorous security requirements.

Specifying the changes in the extra-architectural state could constrain future hardware designs. We believe there is a middle ground where hardware designers can hide implementation some details by undoing all speculative changes to the extra-architectural state (e.g., by implementing a speculative load buffer) and specify other changes (e.g., speculation fences). Finding the right interface that allows space for hardware designers to innovate and system developers to reason about security is an exciting research challenge.

5 FUTURE RESEARCH DIRECTIONS AND CONCLUSIONS

In this paper we have proposed a framework which will allow architects, system software writers, and security experts to better design and mitigate speculation-based attacks like Meltdown and Spectre. We discussed that enforcing only "functional correctness" is not enough in today's security-first era. We believe ISAs need to expand the definition of "correctness" to include the entire state of the system by specifying the speculative state changes to both the *architectural and extra-architectural state*. This will allow programmers and security professionals to *reason* about when applications should and should not allow speculation.

The main research questions we are pursing are *how to specify* speculative state change and *what changes to specify*. By extending the ISA to include more state, there is a significant burden placed on the programmer. On the other hand, using the approach of preventing/undoing all changes in the hardware will increase design complexity and could cause performance regressions.

We are currently investigating the tradeoffs between software complexity, hardware complexity, performance, and vulnerability to choose which speculative state changes to allow, to undo, and to specify. We can leverage prior work on memory consistency models [5] as a starting point for specifying speculative state change. Models like sequential consistency for data-race free programs [6] that have simple semantics in the common case but allow for many performance optimizations are a good example of finding the right tradeoff between complexity and expressiveness.

The separation of the architecture definition (ISA) from the microarchitectural implementation allowed families of machines to be developed, and was a tremendous commercial success. However, given the increasing importance of security, it may be time to rethink the traditional hardware-software interface. If the total behavior of the processor is more rigorously described, security researchers have a far better chance to catch future vulnerabilities *before they make it into products.*

6 ACKNOWLEDGMENTS

We thank Lena Olson, Swapnil Haria, and the anonymous reviewers for providing feedback on this paper.

REFERENCES

[1] 2018. *Intel Analysis of Speculative Execution Side Channels.* White Paper. Intel. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf.

[2] 2018. *SOFTWARE TECHNIQUES FOR MANAGING SPECULATION ON AMD PROCESSORS.* White Paper. AMD. https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf.

[3] Onur Aciiçmez. 2007. Yet Another MicroArchitectural Attack:: Exploiting I-Cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture (CSAW '07).* ACM, New York, NY, USA, 11–18. https://doi.org/10.1145/1314466.1314469

[4] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. On the Power of Simple Branch Prediction Analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS '07).* ACM, New York, NY, USA, 312–320. https://doi.org/10.1145/1229285.1266999

[5] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: a tutorial. *Computer* 29, 12 (Dec 1996), 66–76. https://doi.org/10.1109/2.546611

[6] Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering—a New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90).* ACM, New York, NY, USA, 2–14. https://doi.org/10.1145/325164.325100

[7] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. 2013. *Innovative Technology for CPU Based Attestation and Sealing.* White Paper. Intel Corporation.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[9] Rob Coombs. 2015. *Securing the Future of Authentication with ARM TrustZone-based Trusted Execution Environment and Fast Identity Online (FIDO).* Technical Report. https://www.arm.com/files/pdf/TrustZone-and-FIDO-white-paper.pdf.

[10] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. 2012. Side-channel Vulnerability Factor: A Metric for Measuring Information Leakage. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12).* IEEE Computer Society, Washington, DC, USA, 106–117. http://dl.acm.org/citation.cfm?id=2337159.2337172

[11] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking Branch Predictors to Bypass ASLR. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49).* IEEE Press, Piscataway, NJ, USA, Article 40, 13 pages. http://dl.acm.org/citation.cfm?id=3195638.3195686

[12] David Grawrock. 2009. *Dynamics of a Trusted Platform: A Building Block Approach* (1st ed.). Intel Press.

[13] Richard Grisenthwaite. 2018. *Cache Speculation Side-channels.* White Paper Version 1.1. arm. https://developer.arm.com/support/security-update/download-the-whitepaper.

[14] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems,* Eric Bodden, Mathias Payer, and Elias Athanasopoulos (Eds.). Springer International Publishing, Cham, 161–176.

[15] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith. 2010. Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically. In *2010 IEEE Symposium on Security and Privacy.* 159–172. https://doi.org/10.1109/SP.2010.18

[16] Matthew Hicks, Cynthia Sturton, Samuel T. King, and Jonathan M. Smith. 2015. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15).* ACM, New York, NY, USA, 517–529. https://doi.org/10.1145/2694344.2694366

[17] Galen C. Hunt and James R. Larus. 2007. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.* 41, 2 (April 2007), 37–49. https://doi.org/10.1145/1243418.1243424

[18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom.

2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01203

[19] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[20] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2015. CCICheck: Using &Micro;Hb Graphs to Verify the Coherence-consistency Interface. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 26–37. https://doi.org/10.1145/2830772.2830782

[21] D Page. 2003. Defending against cache-based side-channel attacks. *Information Security Technical Report* 8, 1 (2003), 30 – 44. https://doi.org/10.1016/S1363-4127(03)00104-3

[22] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. https://doi.org/10.1145/1785414.1785443

[23] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/1024393.1024404

[24] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. 2009. Complete Information Flow Tracking from the Gates Up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 109–120. https://doi.org/10.1145/1508244.1508258

[25] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. *ArXiv e-prints* (Feb. 2018). arXiv:cs.CR/1802.03802

[26] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2017. TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 119–133. https://doi.org/10.1145/3037697.3037719

[27] Paul Turner. 2018. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886

[28] Zhenghong Wang and Ruby B. Lee. 2007. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 494–505. https://doi.org/10.1145/1250662.1250723

[29] Hassan M. G. Wassel, Ying Gao, Jason K. Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. 2013. SurfNoC: A Low Latency and Provably Non-interfering Approach to Secure Networks-on-chip. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 583–594. https://doi.org/10.1145/2485922.2485972

[30] David Wentzlaff and Anant Agarwal. 2009. Factored Operating Systems (Fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Oper. Syst. Rev.* 43, 2 (April 2009), 76–85. https://doi.org/10.1145/1531793.1531805