Technical Report

UCAM-CL-TR-916
ISSN 1476-2986

Number 916

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks

Robert N. M. Watson, Jonathan Woodruff,
Michael Roe, Simon W. Moore,
Peter G. Neumann

February 2018

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Approved for public release; distribution is unlimited. Sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 ("CTSRD") and FA8750-11-C-0249 ("MRC2"), as part of the DARPA CRASH and MRC research programs. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government. Additional support was received from St John's College Cambridge, the Google SOAAP Focused Research Award, the RCUK's Horizon Digital Economy Research Hub Grant (EP/G065802/1), the EPSRC REMS Programme Grant (EP/K008528/1), the EPSRC Impact Acceleration Account (EP/K503757/1), the Isaac Newton Trust, the UK Higher Education Innovation Fund (HEIF), Thales E-Security, ARM Ltd, and HP Enterprise.

ISSN 1476-2986

Abstract

In this report, we consider the potential impact of recently announced Meltdown and Spectre microarchitectural side-channel attacks arising out of superscalar (out-of-order) execution on Capability Hardware Enhanced RISC Instructions (CHERI) computer architecture. We observe that CHERI's in-hardware permissions and bounds checking may be an effective form of mitigation for one variant of these attacks, in which speculated instructions can bypass software bounds checking. As with MMU-based techniques, CHERI remains vulnerable to side-channel leakage arising from speculative execution across compartment boundaries, leading us to propose a software-managed compartment ID to mitigate these vulnerabilities for other variants as well.

Acknowledgments

The authors of this report thank other members of the CTSRD, MRC2, and ECATS teams, our past and current research collaborators at SRI and Cambridge, as well as colleagues at other institutions who have provided invaluable feedback and continuing support throughout this work:

Ross J. Anderson	Jonathan Anderson	Graeme Barnes	John Baldwin
Hadrien Barral	Stuart Biles	Matthias Boettcher	David Brazdil
Ruslan Bukin	Brian Campbell	Gregory Chadwick	David Chisnall
James Clarke	Serban Constantinescu	Chris Dalton	Nirav Dave
Brooks Davis	Lawrence Esswood	Anthony Fox	Paul J. Fox
Paul Gotch	Richard Grisenthwaite	Jong Hun Han	Andy Hopper
Alex Horsman	Alexandre Joannou	Asif Khan	Myron King
Chris Kitching	Ben Laurie	Patrick Lincoln	Anil Madhavapeddy
Ilias Marinos	A. Theodore Markettos	Tim Marsland	Ed Maste
Alfredo Mazzinghi	Dejan Milojicic	Andrew W. Moore	Will Morland
Alan Mujumdar	Prashanth Mundkur	Steven J. Murdoch	Kyndylan Nienhuis
Robert Norton	Philip Paeps	Lucian Paul-Trifu	Alex Richardson
Michael Roe	Colin Rothwell	John Rushby	Hassen Saidi
Hans Petter Selasky	Peter Sewell	Muhammad Shahbaz	Lee Smith
Stacey Son	Ian Stark	Andrew Turner	Richard Uhler
Munraj Vadera	Jacques Vidrine	Philip Withnall	Hongyan Xia
Bjoern A. Zeeb			

The CTSRD team also thanks past and current members of its external oversight group for significant support and contributions:

Lee Badger	Simon Cooper	Rance DeLong	Jeremy Epstein
Virgil Gligor	Li Gong	Mike Gordon	Steven Hand
Andrew Herbert	Warren A. Hunt Jr.	Doug Maughan	Greg Morrisett
Brian Randell	Kenneth F. Shotting	Joe Stoy	Tom Van Vleck
Samuel M. Weber			

We would also like to acknowledge the late David Wheeler and Paul Karger, whose conversations with the authors about the CAP computer and capability systems contributed to our thinking on CHERI.

Finally, we are grateful to Howie Shrobe, MIT professor and past DARPA CRASH and MRC program manager, who has offered both technical insight and support throughout this work. We are also grateful to Robert Laddaga, Stu Wagner, Jonathan Smith, Dale Waters, John Launchbury, and Linton Salmon at the DARPA I2O, AEO, and MTO offices, who have continued support for our work at DARPA, as well as to Daniel Adams, Laurisa Georgen, and Marnie Dunsmore, SETAs supporting the DARPA CRASH, MRC, and SSITH programs.

Chapter 1

CHERI and Microarchitectural Side-Channel Attacks

The purpose of this report is to briefly describe our early thoughts on the impact of the Meltdown [9] and Spectre [8] vulnerabilities on the Capability Hardware Enhanced RISC Instructions (CHERI) architecture [14] developed by SRI International and the University of Cambridge. These recently announced vulnerabilities allow attackers to exploit side channels stemming from speculative execution to extract data across protection boundaries – for example, allowing an unprivileged userspace attacker to retrieve private data from the kernel or another process. CHERI is an architectural security feature providing fine-grained memory protection and scalable compartmentalization based on a hybrid capability-system model [13].

As an architectural security model, CHERI depends on secure implementation in the microarchitecture; as with other architectural models (such as virtual memory and enclave technologies), implementations of CHERI may be susceptible to side-channel leakage through microarchitectural timing. However, an unexpected benefit to CHERI is that it also offers some mitigating impact on one of the attack variants due to the potential for speculative checking of bounds in the implementation of CHERI. In the remainder of this report, we briefly introduce the concepts behind CHERI, describe the three variants of the Meltdown and Spectre attacks that have been published, describe current mitigation techniques, and then discuss potential implications for CHERI along with a proposed architectural change to help resist microarchitectural side channels. We also consider the longer-term prospects for remaining microarchitectural side channels.

1.1 Background

Before exploring the implications of CHERI for Meltdown and Spectre, we review the CHERI architecture, Meltdown and Spectre attacks, and known mitigations for these attacks on current architectures.

1.1.1 CHERI

CHERI is a hybrid capability-system architecture that blends historic ideas about capability systems with a conventional RISC architecture utilizing a Memory Management Unit (MMU) [14]. CHERI capabilities are implemented as a new hardware data type whose corruption and misuse are prevented by tagged memory and guarded manipulation. CHERI's hybrid nature enables capability-system features to converge with more conventional architectural design choices: capabilities are evaluated "within" virtual address spaces, and are implemented in a manner intended to align strongly with current architectures, microarchitectures, and memory subsystems.

CHERI is designed to continue to support (and extend) MMU-based software Trusted Computing Bases (TCBs) implemented in the C and C++ programming languages by making it efficient to implement the principles of *least privilege* and *intentionality* due to their strong mitigating impact on software vulnerabilities. Target software includes operating-system kernels and userspace libraries/tools, language runtimes, and large-scale C/C++ applications such as web browsers. Although the capability primitives can be used for many purposes, we have focused on investigating their use for highly compatible fine-grained language-oriented memory protection, as well as scalable software compartmentalization.

In the majority of our current use cases, CHERI capabilities protect references to concrete code or data in memory, rather than abstract software-defined objects. Most typically, this is done by using the capability type to implement C/C++-language pointers, which are then subject to architectural integrity protection, efficient bounds checking, permission checking, and other (so-called) safety properties. CHERI also supports *sealed capabilities* that are immutable and non-dereferenceable, allowing CHERI's protections to be applied to references to software-defined objects – for example, allowing unforgeable object references to be passed to untrustworthy software sandboxes, and to be used as the foundation for fast domain transitions.

With two notable exceptions (exception delivery and explicit protection-domain crossing), the CHERI instruction-set architecture (ISA) implements *monotonicity*: the rights and bounds of a new capability may be no greater than those of the capability from which it is derived. Tagging and guarded manipulation ensure strong *provenance validity* for capabilities: a capability cannot be dereferenced unless it is a valid capability derived from another valid capability via valid transformations. These transformations are ensured by the controlled introduction of initial capabilities in the architecture (at CPU reset), tagged memory that allows in-memory corruption by a data overwrite to be detected, and guarded manipulation that prevents non-monotonic transformation and tag corruption. A complete description of the evolving CHERI architecture can be found in our papers and technical reports [14].

1.1.2 Meltdown and Spectre Attacks

Speculative execution (or *out-of-order execution*) allows microarchitectures to speculatively pursue computations based on predictions about the unknown future outcomes of branches and data loads in execution. Instructions remain uncommitted until the missing data dependency is present, at which time the speculatively executed instructions can either be committed (due to the prediction being correct) or canceled (if the prediction was incorrect). With sufficiently good predictions, speculative execution allows parallel computations in hardware to substantially improve perceived serial execution. Today, this technique is widely deployed in high-performance CPU designs, as many software execution patterns are highly predictable based on past behavior: for example, CPUs routinely predict the outcome of branches and jumps, based on prior execution of the same code having a strong predictive ability for future executions.

Speculative execution is intended to be largely invisible to the architecture against which programmers write software: mispredicted instructions are canceled, preventing their architectural outcomes (register writebacks, memory stores, etc.) from being exposed to software. The primary exception to this is through timing; where speculation is substantially accurate, performance should improve, reducing the time it takes to perform operations that would otherwise be more expensive. This *timing side channel* is measurable through a variety of techniques (including highly accurate cycle counters in contemporary processors present for timekeeping purposes), and allows software to detect when speculation is being effective. Side effects of speculation are especially visible in its impact on caches, which will be filled with memory on the basis of not just architectural execution, but also speculative execution. Timing side channels therefore allow information to flow in two directions: (1) instruction execution can guide future predictions, impacting the behavior of future speculation; and (2) a committed instruction can observe cache timing information in order to infer prior speculated behavior.

The presence of timing side channels in the microarchitecture has long been understood: In the early 2000s, multiple researchers identified timing-based channels allowing the extraction of cryptographic keying material from hyperthreads sharing a first-level cache [2, 12, 10, 3, 4]. However, the degree to which the above two speculative effects, combined with cache timing information, can be exploited by an unprivileged software adversary to extract private information despite architectural security features (such as rings and virtual address spaces) has only recently become apparent with the announcement of the Meltdown [9] and Spectre[8] attack techniques. With these attacks, timing side channels allow malicious unprivileged code to extract the memory contents of the kernel or another target process by manipulating instruction speculation and triggering a cache-timing side channel back to the attacker. Three variants have been described [6]:

Variant 1: bounds-check bypass (CVE-2017-5753) Published by Koch et al., this vulnerability allows the attacker to manipulate speculation such that software invariants (e.g., array bounds checks) are violated in speculation, with their side effects (e.g., out-of-bounds loads and resulting computations) being detectable using cache side channels. The authors demonstrate an attack in which a speculated out-of-bounds array access can be triggered, allowing the contents of arbitrary kernel memory to be leaked via timing.

We observe that while bypassing bounds checks is a particularly catastrophic outcome of the underlying vulnerability, there might be other potential non-bounds-check implications of security significance – e.g., bypassing kernel access-control checks dependent on uncached data.

Variant 2: branch target injection (CVE-2017-5715) Published by Koch et al., this vulnerability relies on leakage of branch-predictor state between contexts, allowing an attacker to train the branch predictor such that they can maliciously influence speculated control flow in another security domain in order to leak data accessible in that domain back to the attacker via a cache timing side channel. The authors demonstrate an virtualization-based attack in which a guest operating system can guide speculated instructions in the host to access known addresses based on private information, allowing the contents of arbitrary host memory to be leaked via timing.

Variant 3: rogue data cache load (CVE-2017-5754) Published by Lipp et al., this vulnerability relies on speculative loads being performed before hardware permission checks (e.g., based on page permissions or ring), allowing architecturally inaccessible data to be revealed through cache timing. The authors demonstrate arbitrary kernel memory extraction from user space at hundreds of kilobytes per second.

1.1.3 Known Mitigations on Current Architectures

Several major CPU vendors have reported vulnerabilities in one or more of the variants, including Intel [7], AMD [1], and ARM [5]. Prior to the public announcement of the Meltdown and Spectre vulnerabilities, processor vendors were given substantial time to explore the impact of the vulnerabilities, and develop potential firmware- and software-based mitigations – in addition to develop longer-term hardware roadmaps that better address side-channel-based attacks. Architectural and software mitigations documented by these vendors generally fall into the following categories:

Branch avoidance in favor of conditional moves In some architectures (e.g., ARMv8, x86-64), conditional-move instructions permit conditional behavior without the need for a branch. Where these can be used in code generation for security-sensitive checks, there is no branch prediction to be manipulated. In some implementations, speculation is still possible, if the value to be tested by a conditional move is not yet available (e.g., due to depending on an as-yet incomplete speculative load); in this case, explicit speculation barriers will also be required (see below). However, for many common microarchitectures (e.g., most ARM microarchitectures at the time of writing), this is a reasonable form of mitigation as data-value speculation is not used, although software implementors will wish to start introducing explicit barriers to guard against future microarchitectural choices (see below).

Speculation barriers and code behaviors to limit speculation Where branches must be used, or on microarchitectures where conditional moves can depend on predicted values, speculation barrier instructions can be used to prevent further speculative loads based on unresolved branches or predicted values. These barriers can be inserted before security-critical branches (e.g., for bounds and permission checks), at some cost to performance due to a delay in issuing speculative memory accesses dependent on the unresolved branch.

Intel and AMD already provide a Load Fence (`LFENCE`) instruction as a barrier to await termination of all outstanding speculation before proceeding. Placing an `LFENCE` instruction following a software bounds check, but before memory access, will prevent speculative execution of the memory access until the result of the check is known. AMD further observes that ensuring that the outcome of a bounds check is a dependency for the load will have a similar effect in stalling the load – e.g., by using the result of the bounds check to mask the load address. On AMD processors, placing an `LFENCE` instruction prior to an indirect jump will have a similar effect in controlling speculation across the jump.

Intel and AMD are introducing three new classes of barriers to restrict speculation across branches: to limit indirect branch prediction after an explicit barrier (`IBPB`), to restrict branch speculation when set (`IBRS`), and to protect hyperthreads from branch-predictor manipulation (`STOBP`). These features will be made available via new processors and/or microcode patches.

ARM is introducing one additional new barrier instruction, Conditional Speculation Dependency Barrier (`CSDB`), which will prevent speculative loads, stores, instruction prefetches, and indirect branches after the barrier from influencing cache allocation until the dependency has been resolved.

The *retpoline* ("return trampoline") technique causes indirect jumps to occur via an architecture's function-return mechanism, which prevents effective branch prediction on Intel and AMD processors [11].

Limiting speculation has a potentially significant impact on performance, making it desirable to use these barriers only where necessary for security – and creating the opportunity for accidental omission by programmers.

Explicit flushes of architectural and microarchitectural state Where microarchitectures allow branch-predictor state to be shared between rings (e.g., between userspace and kernel), or between address spaces (e.g., between two user processes), an explicit invalidation of that state may be possible. This can be used during a context switch or on entry to the kernel to prevent malicious manipulation of speculation using the branch-predictor table. In general, architectures appear not to include portable interfaces to flush branch-predictor state, unlike other shared microarchitectural state such as Translation Lookaside Buffer (TLB) entries.

AMD has recommended that clearing untrusted values from registers when entering privileged mode, or sensitive code, will prevent speculative use of those values.

Implicit flushes of microarchitectural state Although explicit microarchitectural state invalidation may not be available in all architectures, it may be possible to reliably invalidate that state through architectural behaviors that are known to affect that state. For example, a well-defined sequence of branch or jump instructions may reliably flush the branch-predictor table on specific microarchitectures. These sequences can similarly be inserted in context switches or on entry to the kernel.

AMD suggests that flushing the branch-predictor table using a well-defined series of function calls (sized to reflect the microarchitectural table size – e.g., 32) will also be effective in preventing past branch behavior from affecting future branch prediction. This instruction sequence can be used during entry to privileged modes to prevent userspace behavior from influencing kernel branch prediction.

Unsharing user and kernel address spaces For performance reasons, it is common practice to allow user and kernel code to share the same architectural address space, relying on the ring mechanism and page permissions (or segments) to prevent undesired access. On microarchitectures where unprivileged code can manipulate the speculative memory accesses of privileged code despite that mechanism, another mitigation option is to place kernel memory in a different address space than user memory.

This can be accomplished by arranging for the majority of kernel code to execute with a different address-space identifier (ASID or PCID). Kernel Page Table Isolation (KPTI), previously known as Kernel Address Isolation to have Side-channels Efficiently Removed (KAISER), modifies the operating-system kernel to avoid leaving the kernel address space mapped when operating in userspace. The performance impact of this technique could be substantial, as it may increase the footprint of ASIDs or PCIDs, or require other further address-space manipulations – especially on architectures or microarchitectures without support for TLB entries tagged with address-space identifiers.

On Intel and AMD processors, use of the SMEP (Supervisor Mode Execution Protection) will prevent errant speculation of user instructions from kernel mode; SMAP (Supervisor Mode Access Protection) will prevent errant access to user data via speculated kernel loads and stores, reducing the tools available to attackers by virtue of use of user instructions or memory contents during speculative execution.

Architectural control-flow integrity Intel has observed that use of its recent Control-flow Enforcement Technology (CET) to implement Control-Flow Integrity (CFI) can reduce gadgets available to attackers during speculation.

1.2 Applicability to the CHERI Architecture

CHERI is an architectural security technique that relies on correct implementation in the microarchitecture. CHERI's features interact with the side-channel attacks described above in a manner dependent on the specific microarchitecture, both offering some mitigating aspects, and also requiring architectural change for CHERI.

Variant 1: Bounds-Check Bypass This attack depends on a mispredicted software bounds check resulting in a speculative load that is out of bounds, in turn enabling a cache-based timing side channel back to the attacker. With CHERI, bounds (and other) checks are performed atomically with memory access, based on capability information stored in the same register as the pointer value.

As long as data-value speculation is not performed on capabilities themselves, all necessary capability information required to validate a speculative load will be reliably available to the microarchitecture – eliminating the opportunity for malicious manipulation of speculative software-based bounds checking. It is therefore important that microarchitectures that speculatively perform memory access via capabilities also perform associated capability checks as well. This is a necessary condition before allowing any visible timing effects (e.g., cache perturbation).

It is unclear whether there are techniques to allow safe speculation based on predicted capability values, and as such we recommend against such speculation.

Variants 2: Branch Target Injection Branch predictors must make predictions without access to actual instructions or target addresses, whether represented as integer values or CHERI capabilities. Branch target injection may therefore be equally effective against CHERI as with traditional architectures, as the microarchitecture would not naturally prevent the branch predictor from injecting speculative targets from one domain into another, causing that domain to exercise its own rights to load data in a manner detectable by the attacking domain. The vulnerable domain may be the kernel, another process, or an in-address-space CHERI compartment.

Variants 3: Rogue Data Cache Load Where the values of capabilities themselves are not subject to being predicted, CHERI provides adequate information to the microarchitecture to prevent undesired memory access beyond the bounds (or in violation of other restrictions) in capabilities. As with Variant 1, the microarchitecture must ensure that any necessary capability checks have passed before allowing any side effects of a memory access via that capability to become visible via timing. This should also be possible with MMU protection, and indeed only certain commercial memory systems are vulnerable to this variant.

1.3 Implications for the CHERI Architecture

These observations motivate a number of thoughts on how the CHERI architecture should be implemented, and perhaps also might be changed.

1.3.1 Speculative Memory Access via Capabilities

Most critically, it is clear that memory accesses via capabilities must not have timing-visible side effects unless CHERI's checks (tag, bounds check, permission check, seal check, ...) have been satisfied by the microarchitecture. Otherwise, speculation across those checks may leak information about out-of-bounds or protected memory. However, if implemented in keeping with this design principle, CHERI offers substantial mitigation against Variants 1 and 3.

1.3.2 Speculated Capability Values

With CHERI, we are particularly concerned with the potential for data-value speculation (rather than simply branch prediction) to lead to violation of invariants – especially where the value of a capability might be speculated. This would mean that its bounds, permissions, and other state might themselves be predicted. In an ideal world, speculated values would be limited to capabilities actually held by the current protection domain – but it is not clear that this is microarchitecturally viable, even if considering only current register-file entries. It may be that speculating capability values has to be entirely disallowed in the presence of compartmentalization, which could have substantial cost.

1.3.3 Control-Flow Robustness in Speculation

Intel's observation that CET may assist with limiting branch-predictor-based attacks due to reducing the gadget space may also apply to capability-based limits on control flow. For example, indirect branches by integer offset rather than to a target capability will be constrained to the current program-counter capability's bounds. Similarly, speculated indirect branches to target capabilities will have their range limited by the permissions and bounds on the capability. It is not yet clear to what extent this provides effective attack mitigation, however.

1.3.4 Branch-Predictor State

A natural design choice for microarchitectural implementations of CHERI will be to retain current virtual-address-based branch predictors without also saving additional protection metadata (such as bounds) for those target addresses. It is therefore important that any bounds checks occur before a predicted target is inserted in the branch-predictor table to prevent predicted addresses outside of accessible bounds from being being used speculatively.

1.3.5 Sharing Microarchitectural State Between Compartments

For Variants 2 and 3, it is important that the microarchitecture must control sharing of microarchitectural state, not only across ring or process boundaries (i.e., userspace to kernel, or process to process), but also between CHERI compartments. For MMU-based isolation, it is possible to tag state – e.g., in the branch predictor – with the originating ring and process. With CHERI, however, the configuration of accessible capabilities can further define an in-process compartment.

One approach would be to limit speculation beyond potential non-monotonic operations in the architecture – specifically, exception delivery (increasingly understood for conventional architectures) and userspace capability invocation (not present in conventional architectures), as both of these grant access to capabilities not available to prior instructions. However, it is not clear that this is a desirable approach, as these are frequent operations occuring in performance-critical paths, and may not in fact require strong confidentiality protection.

1.4 Proposal: A CHERI Compartment Identifier (CID)

Another approach to controlling sharing of microarchitectural state between CHERI compartments is to explicitly identify compartments to the microarchitecture so that persisting microarchitectural state can be tagged and used only in appropriate contexts. This technique is already used with MMU-based compartmentalization by virtue of tagged TLBs: entries in the TLB are tagged with address-space identifiers preventing an entry from matching the wrong process. This would require introduction of an architectural *CHERI compartment identifier (CID)* to explicitly identify when microarchitectural state is (or isn't) allowed to be shared. As with the ring and address-space identifier, this could be included in the tag on branch target buffer entries (or any other microarchitectural state) to ensure that, for example, one compartment could not train the branch predictor in order to direct speculative execution in another compartment. We imagine that this mechanism would be in the form of a writable special register identifying the current CID, which could be set with a suitable capability authorizing use of a range of CIDs. CID changes would themselves require some care with respect to speculation, to ensure that those changes were not improperly speculated across.

In keeping with the virtualizability goals of CHERI, the authorizing capability would be able to specify bounds on what CIDs could be used, allowing domain-crossing code to be authorized to select the microarchitectural state of selected compartments it is authorized to enter. This approach comes with its challenges: for example, how should the state of a userspace domain switcher itself be protected, to prevent speculation across the switcher from revealing its own internal state, or the state of another compartment? One option here would be to also incorporate the sealed-capability object type into the concatenated index, as this namespace is also controlled. With exception-based domain switching, we believe that inclusion of the ring identifier should be sufficient. Another critical question is how large the CID must be in order to prevent collisions through reuse, as well as make partitioned delegation to multiple switchers efficient: we suspect at least a 24-bit ID would be desirable, if not far larger. This proposed size is based on the size of the object-type field in our 128-bit compressed-capability design, which in turn is based on an estimate on an upper bound on the number of unique sealed capability types that might be in use in a single address-space system. A larger size might be preferable, as unique code segments may be combined with additional unique data segments to produce a larger number of compartments.

An immediate concern to any microarchitect would be the size of the tag required on microarchitectural state. A large CID as well as an ASID and ring number may approach the entire storage capacity of current branch target entries. We might suggest an architectural table that maps a small number of active domains drawn from a larger architectural namespace to a small internal microarchitectural namespace of, perhaps, 8 bits. When a new domain is inserted that does not have a corresponding table entry, the kernel would replace an existing entry in the active domain table, removing all microarchitectural state for the replaced domain. With this mechanism, any partitioned microarchitectural state need be tagged with only a small number of bits to identify an arbitrary domain. Clearly, this approach will be efficient only if the time between insertion of new domains is large with respect to the cost of replacing a mapping in the active domain table, which seems likely for a table of even modest size.

There are a various tradeoffs around the use of a CID to partition microarchitectural state. Hard partitioning of that state may lead to performance loss where prior sharing led to performance gain – e.g., where branch-predictor state was shared beneficially between two compartments sharing common code. This concern also arises in the context of harder partitioning for MMU-based process schemes: forked processes executing common code (e.g., a language runtime or server application) may benefit from branch-predictor leakage between tasks, which would be prevented by partitioning. However, architectural CIDs need not map one-to-one with software compartments: if only integrity or availability is required (and not confidentiality), then a software compartment change can take place without requiring use of an independent microarchitectural state.

1.5 Future Side-Channel Challenges

Potential side-channel leakage is endemic to current microarchitectural optimization techniques – not just with respect to speculative execution, but also in any area where there is resource sharing rather than hard partitioning between mutually distrusting parties. Historically, cache side channels have been of particular interest because they have allowed effective attacks without any need for speculation – and also are a key linchpin for both Meltdown and Spectre. It is reasonable to anticipate that these and other side channels will remain relevant, and while effective mitigations have been identified for the current attacks, the broader picture is far less clear.

1.5.1 Tagging Does Not Close All Side Channels

More generally, constraining matching of branch-predictor entries does not entirely close the side channel, but simply makes it hard to maliciously impact choices in a victim task. Information will still leak between address spaces (or compartments) by virtue of entry displacement; while this has not been shown to be exploitable, history suggests that signal-analysis techniques often prove able to extract signal from noise, even when the signal-to-noise ratio is very low. Similarly, policy-based lookup and displacement constraints do not generalize well to all resource types – e.g., cache entries, bus contention, thread scheduling, etc.; this implies that many side channels will continue to remain, even if it is not yet clear how to exploit those channels.

1.5.2 Tagging Performance Overheads

There are also potential performance costs to tagging and lookup constraints, which have not yet been fully explored. For example, tagging branch-predictor entries with address-space identifiers may harm the performance of forked applications (such as the Android Dalvik language runtime, or the Apache web server), where accidental sharing between address spaces might lead to accurate branch prediction due to identical address-space layouts. This concern will also apply to future attempts to limit state sharing between address spaces – or, in CHERI, between compartments.

1.5.3 Principled Design and Microarchitectural Side Channels

We are intrigued by the possibility that a stronger expression of privilege (and privilege minimization) visible to the architecture but enforceable in the microarchitecture can be effective in mitigating microarchitectural side channels – a property that we did not foresee when designing CHERI. This possibility has arisen through CHERI's inclusion of fine-grained checking of bounds, permission, and other properties in an architecture-visible manner; with extensions to include a compartment ID, this would also allow software to limit speculation in the presence of granular software compartmentalization. If generalizable, this would reinforce the view that design principles such as the *principle of least privilege* remain a powerful approach to systems design when it comes to not just preventing vulnerabilities, but also mitigating their inevitable effects.

1.6 Conclusion

Microarchitectural side-channel attacks have long been known to the computer architecture and security communities, epitomized by cache-timing side-channel attacks on cryptography dating to the early 2000s. However, recent attacks such as Meltdown and Spectre have illustrated the power of those attacks against general-purpose systems software outside of more specialized environments and workloads. As an architectural security feature, CHERI is dependent on correct and secure implementation in the microarchitecture, especially with respect to side channels. By pushing greater knowledge of security-related constraints into the architecture, CHERI offers the microarchitecture the opportunity to mitigate some forms of side-channel attacks – for example, through its architectural bounds checking. To allow the microarchitecture to mitigate other forms of side-channel attacks, CHERI requires modest extensions to expose a more explicit notion of compartments (and compartmentalization) via the ISA. As understanding of these attacks grows, we anticipate further insights into the impact of microarchitectural side channels and architectural security features. Overall, we believe that the CHERI hardware-software system architecture will continue to provide a sound basis upon which to explore remediation of known vulnerabilities as well as avoidance of as yet unknown vulnerabilities.

Bibliography

[1] AMD. Software techniques for managing speculation on AMD processors. https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf, January 2018. Accessed 2018-01-31.

[2] D. J. Bernstein. Cache-timing attacks on AES. Technical report, University of Illinois at Chicago, 2005.

[3] J. Bonneau. Robust final-round cache-trace attacks against AES. *IACR Cryptology ePrint Archive*, 2006:374, 2006.

[4] J. Bonneau and I. Mironov. Cache-collision timing attacks against AES. In *in Proc. Cryptographic Hardware and Embedded Systems (CHES) 2006. Lecture Notes in Computer Science*, pages 201–215. Springer, 2006.

[5] R. Grisenthwaite. Cache speculation side-channels. https://developer.arm.com/support/security-update/download-the-whitepaper, January 2018. Version 1.1, Accessed 2018-01-31.

[6] J. Horn. Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.co.uk/2018/01/reading-privileged-memory-with-side.html, January 2018. Accessed: 2018-01-31.

[7] Intel. Intel analysis of speculative execution side channels. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, January 2018. Revision 1.0; Accessed 2018-01-31.

[8] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

[9] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[10] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, pages 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[11] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886, January 2018. Accessed 2018-01-31.

[12] C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[13] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for Unix. In *Proceedings of the 19th USENIX Security Symposium*. USENIX, August 2010.

[14] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson, J. Baldwin, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S. W. Moore, S. J. Murdoch, R. Norton, S. Son, and H. Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 6). Technical Report UCAM-CL-TR-907, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, Apr. 2017.