Hardware-Aware Security Verification and Synthesis

Margaret Martonosi
H. T. Adams ‘35 Professor
Dept. of Computer Science
Princeton University

Joint work with Caroline Trippel, Princeton CS PhD student
and Dr. Daniel Lustig, NVIDIA
The Check Suite: An Ecosystem of Tools For Verifying Memory Consistency Model Implementations

High-Level Languages (HLL)
- Compiler
- OS
Architecture (ISA)
Microarchitecture
RTL (e.g. Verilog)

TriCheck [ASPLOS ‘17] [IEEE MICRO Top Picks]
COATCheck [ASPLOS ‘16] [IEEE MICRO Top Picks]
PipeCheck [Micro ‘14] [IEEE MICRO Top Picks]
CCI Check [Micro ‘15] [Nominated for Best Paper Award]
RTLCheck [Micro ‘17] [IEEE MICRO Top Picks Honorable Mention]

Our Approach
• Axiomatic specifications -> Happens-before graphs
• Check Happens-Before Graphs via Efficient SMT solvers
  • Cyclic => A->B->C->A... Can’t happen
  • Acyclic => Scenario is observable
Check: Formal, Axiomatic Models and Interfaces

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), "PPO").
TriCheck Framework: Verifying Memory Event Ordering from Languages to Hardware

- High-level Lang Litmus tests
- HLL Mem Model Sim
- HLL->ISA Compiler Mappings
- ISA Mem Model
- ISA-level Litmus tests
- uArch Mem Model

Permitted/Forbidden

Compare Outcomes

<table>
<thead>
<tr>
<th>Permit</th>
<th>Obs.</th>
<th>Not obs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ok</td>
<td></td>
<td>Over strict</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Forbid</th>
<th>Obs.</th>
<th>Not obs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bug</td>
<td></td>
<td>ok</td>
</tr>
</tbody>
</table>

Observable/Unobservable
TriCheck Framework: Verifying Memory Event Ordering from Languages to Hardware

- High-level Lang Litmus tests
- HLL Mem Model Sim
- ISA Mem Model
- uArch Mem Model
- ISA-level Litmus tests

Permitted/Forbidden

Compare Outcomes

<table>
<thead>
<tr>
<th>Obs.</th>
<th>Not Obs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permit</td>
<td>ok</td>
</tr>
<tr>
<td>Forbid</td>
<td>Bug</td>
</tr>
</tbody>
</table>

Observable/Unobservable

Iteratively Refine Design: HLL, Compiler, ISA, uArch
Conclusion: Draft RISC-V spec could not serve as a legal C11 compiler target.

Status: RISC-V Memory Model Working Group formed to address these issues. Just voted to ratify new, improved RISC-V memory model.

7 Distinct RISC-V Implementations (All abide by RISC-V specifications, but vary in reordering / performance)

Base RISC-V ISA: 144 buggy outcomes
Base+Atomics: 221 buggy outcomes
CheckMate: From Memory Consistency Models to Security

- **Well-known** cache side-channel attack
- **Widely-used** hardware feature
- **New exploit**
- **2 new attacks**

**Flush+Reload** + **Speculation** = **New exploit**

January 2018: Spectre & Meltdown

**Project Zero**

News and updates from the Project Zero team at Google

**TRIPLE MELTDOWN: HOW SO MANY RESEARCHERS FOUND A 20-YEAR-OLD CHIP FLAW AT THE SAME TIME**
Attack Discovery & Synthesis: What We Would Like

1. Specify system to study
   - Formal interface and specification of given system implementation

2. Specify attack pattern
   - E.g. Subtle event sequences during program’s execution

3. Synthesis
   - Either output synthesized attacks. Or determine that none are possible
Attack Discovery & Synthesis: CheckMate TL;DR

1. Specify system to study
2. Specify attack pattern
3. Synthesis

- **What we did**: Developed a tool to do this, based on the uHB graphs from previous sections.
- **Results**: Automatically synthesized Spectre and Meltdown, as well as two new distinct exploits and many variants.

In more detail...
CheckMate Methodology

1. Frame classes of attacks as patterns of event interleavings?
   - Essentially a snippet out of a happens-before graph

2. Specify hardware using uSpec axioms
   - Determine if attack is realizable on a given hardware implementation
Microarchitecture-Aware Program Synthesis

Enumerate all possible execution graphs with pattern

Core 0
W [x]→1 R [x]→r0
Fetch
Execute
Commit
Store Buffer
L1 ViCL Create
L1 ViCL Expire
Main Memory
Complete

Load being sourced from the store buffer

#cores = 1
#threads = 1
#instr ≤ 2

Microarchitecture

μhb Pattern
Execute
Store Buffer
L1 ViCL Create

Execution Constraints

μhb Graph

Check Mate
Microarchitecture-Aware Program Synthesis

**Microarchitecture Specification**

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), "PPO").

Prior Check tools work addresses many of these issues

- SW/OS/HW events and locations
- SW/OS/HW ordering details
- Hardware optimizations, e.g., speculation
- Processes and resource-sharing
- Memory hierarchies and cache coherence protocols
A relational model is a set of constraints on an abstract system (for CheckMate, a μhb graph) of:

- Set of abstract objects (for CheckMate, μhb graph nodes)
- Set of N-dimensional relations (for example, 2D μhb graph edges relation connecting 2 nodes)

For CheckMate, the constraints are a μhb pattern of interest

RMF attempts to find and satisfying “instance” (or μhb graph)

Implementation: Alloy DSL maps RMF problems onto Kodkod model-finder, which in turn uses off-the-shelf SAT solvers

CheckMate Tool maps μspec HW/OS spec to Alloy
Spectre (Exploits Speculation)

Flush+Reload Threat Pattern

Spectre Security Litmus Test

Initial conditions: \([x]=0, [y]=0\)

<table>
<thead>
<tr>
<th>Action</th>
<th>Initial State</th>
<th>Final State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush</td>
<td></td>
<td>[VAa1] (\rightarrow) 0</td>
</tr>
<tr>
<td>Reload</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch PT, NT</td>
<td>[VAa0] (\rightarrow) 0</td>
<td></td>
</tr>
<tr>
<td>R f(r1) = VAa1</td>
<td></td>
<td>[VAa1] (\rightarrow) 0</td>
</tr>
<tr>
<td>R [VAa1] (\rightarrow) 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Prime&Probe Attack Pattern: Synthesizing MeltdownPrime & SpectrePrime

CheckMate
Is hardware susceptible to exploit?

Hardware-specific exploit programs (if susceptible)

Prime+Probe
Microarchitecture feat. OOO execution & speculation

L1 ViCL Create
L1 ViCL Expire

Attacker observes a cache hit
SpectrePrime uhb Graph

Initial conditions: [x]=0, [y]=0

<table>
<thead>
<tr>
<th>Attacker T0</th>
<th>Attacker T0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R [VAa1] → 0</td>
<td>R [VAa1] → 0</td>
</tr>
<tr>
<td>Branch → PT,NT</td>
<td>Branch → PT,NT</td>
</tr>
<tr>
<td>R [VAS0] → r1</td>
<td>R [VAS0] → r1</td>
</tr>
<tr>
<td>W [f(r1)=VAa1] → 0</td>
<td>W [f(r1)=VAa1] → 0</td>
</tr>
<tr>
<td>Prime</td>
<td>Probe</td>
</tr>
</tbody>
</table>

Prime+Probe Threat Pattern

ViCLCreate

ViCLEXpire
Overall Results: What exploits get synthesized? And how long does it take?

<table>
<thead>
<tr>
<th>Exploit Pattern</th>
<th>#Instrs (RMF Bound)</th>
<th>Output Attack</th>
<th>Minutes to synthesize 1st exploit</th>
<th>Minutes to synthesize all exploits</th>
<th>#Exploits Synthesized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush +Reload</td>
<td>4</td>
<td>Traditional Flush+Reload</td>
<td>6.7</td>
<td>9.7</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Meltdown</td>
<td>27.8</td>
<td>59.2</td>
<td>572</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Spectre</td>
<td>101.0</td>
<td>198.0</td>
<td>1144</td>
</tr>
<tr>
<td>Prime +Probe</td>
<td>3</td>
<td>Traditional Prime+Probe</td>
<td>5.4</td>
<td>6.7</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>MeltdownPrime</td>
<td>17.0</td>
<td>8.2</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>SpectrePrime</td>
<td>71.8</td>
<td>76.7</td>
<td>24</td>
</tr>
</tbody>
</table>
CheckMate: Takeaways

• New Variants reported: SpectrePrime and MeltdownPrime
  • Speculative cacheline invalidations versus speculative cache pollution
  • Software mitigation is the same as for Meltdown & Spectre

• Key overall philosophy:
  • Event ordering in security exploit patterns aligns strongly with MCM analysis
  • Move from ad hoc analysis to formal automated synthesis.
  • Span software, OS, and hardware for holistic hardware-aware analysis

Acknowledgements

• CheckMate: Caroline Trippel (Princeton CS PhD student) and Dan Lustig (NVIDIA)

• Funding: NSF, NVIDIA Graduate Fellowship

• Check Tools, additional co-authors: Yatin Manerkar, Abhishek Bhattacharjee, Michael Pellauer, Geet Sethi

Me: http://www.princeton.edu/~mrm
Group Papers: http://mrmgroup.cs.princeton.edu
Check and CheckMate Tools: http://check.cs.princeton.edu
Thank you!