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

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").
Example: ARM Read-Read Hazard

- ARM ISA spec ambiguous regarding same-address Ld → Ld ordering:
  - Compiler’s job? Hardware job?
- C/C++ variables with atomic type require same-addr. Ld → Ld ordering
- ARM issued errata 1:
  - Rewrite compilers to insert fences (ordering instructions) with performance penalties
- ARM ISA had the right ordering instructions – just needed to use them.

Original: Alglave 2011
TriCheck Framework: Verifying Memory Event Ordering from Languages to Hardware

High-level Lang Litmus tests → HLL Mem Model Sim → Compare Outcomes

HLL->ISA Compiler Mappings → ISA Mem Model

ISA-level Litmus tests → uArch Mem Model

Permitted/Forbidden

Observable/Unobservable

<table>
<thead>
<tr>
<th></th>
<th>Obs.</th>
<th>Not obs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permit</td>
<td>ok</td>
<td>Over strict</td>
</tr>
<tr>
<td>Forbid</td>
<td>Bug</td>
<td>ok</td>
</tr>
</tbody>
</table>
TriCheck Framework: Verifying Memory Event Ordering from Languages to Hardware

High-level Lang Litmus tests → HLL Mem Model Sim → Permitted/Forbidden → Compare Outcomes

HLL-ISA Compiler Compiler Mappings

ISA Mem Model

ISA-level Litmus tests → uArch Mem Model → Observable/Unobservable

Iteratively Refine Design: HLL, Compiler, ISA, uArch

<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>
TriCheck Framework: RISC-V Case Study

1701 C11 Programs

High-level Lang Litmus tests → HLL Mem Model Sim

Permitted/Forbidden

Observable/Unobservable

Base RISC-V ISA:
144 buggy outcomes
Base+Atomics:
221 buggy outcomes

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)
CheckMate: From Memory Consistency Models to Security

Well-known cache side-channel attack

Flush+Reload

Widely-used hardware feature

Speculation

New exploit

2 new attacks

January 2018: Spectre & Meltdown:
Spectre and Meltdown: Details you need on those big chip flaws
Design flaws in processors from leading chipmakers could let attackers access sensitive information. How did this happen, and what's the fix?

Project Zero
News and updates from the Project Zero team at Google

Wednesday, January 3, 2018
Reading privileged memory with a side-channel
Posted by Jann Horn, Project Zero
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
**TL;DR**

Axiomatic specifications similar to Check tools

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.

[Trippel, Lustig, Martonosi. MICRO-51. October, 2018]
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
Exploit Programs: $\mu$hb Graphs featuring Exploit Patterns

1. Model subtle hardware-specific event orderings/interleavings: $\mu$hb graphs

2. Determine if an exploit is possible for a given implementation: cycle checks

Prime+Probe “exploit execution pattern”
Microarchitecture-Aware Program Synthesis

Microarchitecture

μhb Pattern
- Execute
- Store Buffer
- L1 ViCL Create

Load being sourced from the store buffer

Execution Constraints
- #cores = 1
- #threads = 1
- #instr ≤ 2

Microarchitecture

Fetch
Exec.
Commit

SB
Lds.
SB

L1
L1
Main Memory

Execution Constraints

Enumerate all possible execution graphs feat. μhb Graph

Check Mate

Core 0
W [x]→ 1
R [x]→ r0
Fetch
Execute
Commit
Store Buffer
L1 ViCL Create
L1 ViCL Expire
Main Memory
Complete
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 work addresses the problem of proving this correct with respect to RTL:

- 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
Relational Model Finding (RMF): A Natural Fit for Security Litmus Test Synthesis

- 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)

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

Flush+Reload Threat Pattern

L1 ViCL Create
L1 ViCL Expire

Spectre Security Litmus Test

<table>
<thead>
<tr>
<th>Initial conditions: ([x]=0, [y]=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Attacker T0</strong></td>
</tr>
<tr>
<td>(R [VA_{a1}] \rightarrow 0)</td>
</tr>
<tr>
<td>CLFLUSH ([VA_{a1}])</td>
</tr>
<tr>
<td>Branch (\rightarrow PT, NT)</td>
</tr>
<tr>
<td>(R [VA_{v0}] \rightarrow r1)</td>
</tr>
<tr>
<td>(R [f(r1)=VA_{a1}] \rightarrow 0)</td>
</tr>
<tr>
<td>(R [VA_{a1}] \rightarrow 0)</td>
</tr>
</tbody>
</table>
Prime & Probe Attack Pattern: Synthesizing MeltdownPrime & SpectrePrime

Prime+Probe

Microarchitecture feat. OOO execution & speculation

CheckMate
Is hardware susceptible to exploit?

Hardware-specific exploit programs (if susceptible)

L1 ViCL Create
L1 ViCL Expire

Attacker observes a cache hit
SpectrePrime uhb Graph

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

Attacker T0 \(\rightarrow\) 0

R \(\left[ VA_{a1} \right] \rightarrow 0\)

Branch \(\rightarrow\) PT, NT

W \(\left[ VA_{a1} \right] \rightarrow 0\)

Prime+Probe Threat Pattern

ViCLCreate

ViCLExpire

Spectre Security Litmus Test

<table>
<thead>
<tr>
<th>Initial conditions: ([x]=0, [y]=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attacker T0</td>
</tr>
<tr>
<td>R (\left[ VA_{a1} \right] \rightarrow 0)</td>
</tr>
<tr>
<td>Branch (\rightarrow) PT, NT</td>
</tr>
<tr>
<td>R (\left[ VA_{a0} \right] \rightarrow r1)</td>
</tr>
<tr>
<td>W (\left[ f(r1)=VA_{a1} \right] \rightarrow 0)</td>
</tr>
</tbody>
</table>
## 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</td>
<td>6.7</td>
<td>9.7</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flush+Reload</td>
<td></td>
<td></td>
<td></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</td>
<td>5.4</td>
<td>6.7</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Prime+Probe</td>
<td></td>
<td></td>
<td></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:
  • Move from ad hoc analysis to formal automated synthesis.
  • Span software, OS, and hardware for holistic hardware-aware analysis

[Trippel, Lustig, Martonosi. MICRO-51. October, 2018]
Acknowledgements

• CheckMate Co-Authors: Caroline Trippel, Princeton CS PhD student and Daniel 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
Verification Tools: http://check.cs.princeton.edu
Thank you!