“Who ya gonna call?”

Cybersecurity for the Spectre Era

Calvin Deutschbein

- Spectre targets **hardware** (all Intel processors since 1995)
- Spectre leaves **no traces** in traditional logs
- Spectre went **undetected for over two decades**

Learn more: [meltdownattack.com/](https://meltdownattack.com/)
What is Spectre?

The Meltdown and Spectre exploits use "speculative execution?" What's that?

You know the trolley problem? Well, for a while now, CPUs have basically been sending trolleys down both paths, quantum-style, while awaiting your choice. Then the unneeded "phantom" trolley disappears.

The phantom trolley isn't supposed to touch anyone. But it turns out you can still use it to do stuff, and it can drive through walls.
The Branch Prediction Problem in 1995
To avoid delays, x86 chips “guess” what’s next
But what if chips guess an unused branch?
Speculative execution can access memory...

*This is the point of speculative execution.
Speculative execution can access memory...and bypass memory protections*

OS permissions

*more of an accident
Exploiting the Vulnerability

Spectre is a vulnerability - it provides an entry point for an adversary.

- Spectre: “Branch misprediction may leave observable side effects”

Adversaries must exploit the vulnerability to gain access to secure data.

- Exploit: “Using observable side effects to access secure data.”

The mere presence of the vulnerability in on hardware not running code capable of exploiting the vulnerability will not result in a security violation.
How adversaries can exploit Spectre

Hello I am a trustworthy website.

Can you help me with this Trolley problem?

I don't trust you but my memory protections mean I can help
How adversaries can exploit Spectre

How about these 10 trolley problems followed by a timed memory access to this specific address?

Of course! Surely there is no way this will leak secure data.
I requested, and was denied, to read the passwords folder 10 times. But the third time I was denied 50 times faster!

I can use another attack to read the values in cache after that request.
Thus, an attacker gains access to secure data.
Three attack surfaces:

- **Timing**
  - Timing cache hits
- **Microarchitecture**
  - Branch prediction
  - Below operating system or assembly
- **Multiple Executions**
  - Multiple runs expose timing differences
So… "Who ya gonna call?"
Mining Behavior

My research shows the technique of specification mining can find:

- **Temporal** properties, for timing
- **Closed source CISC** architecture properties, for microarchitecture
- **Hyperproperties**, properties over multiple traces of execution
Defining Secure Behavior

For x86-64, no specification exists - so we create one.
Specification Mining

- Miners accept as input **traces of execution**.
  - For example, the debug output of an x86-64 processor booting Linux.
- Miners find **properties** that hold over the traces.
  - For example, “if reset is active, then the privilege level is supervisor”.
  - RESET==0 \Rightarrow \text{CURRENT\_PRIVILEGE\_LEVEL==0}
- Miners contain powerful **inference engines** for high performance.
Undine: Mining Temporal Properties

Can **linear temporal logic** properties that model secure behavior be discovered using specification mining?

A library of typed templates for my miner, Undine, enable it to find security temporal properties, including properties using \( G \) (Globally) or \( X \) (Next) operators.
Difficulties Finding Security Properties

- Too Many Properties
- Properties Not Security Related
- Do Not Capture Semantic Info
Without separate events there are many properties

Sample Trace

reg_a==1
reg_b==1
reg_c==0
reg_d==0
reg_a==reg_b
reg_c==reg_d

Mined 30 G(x \rightarrow y)

reg_a==1 \rightarrow reg_b==1
...
reg_a==1 \rightarrow reg_c==reg_d
...
reg_c==reg_d \rightarrow reg_a==reg_b
Templates Refine to Security Properties

Sample Trace

\[
\begin{align*}
\text{reg}_a &= 1 \\
\text{reg}_b &= 1 \\
\text{reg}_c &= 0 \\
\text{reg}_d &= 0 \\
\text{reg}_a &= \text{reg}_b \\
\text{reg}_c &= \text{reg}_d
\end{align*}
\]

Mined 8 G(\(R \rightarrow R-R\))

\[
\begin{align*}
\text{reg}_a &= 1 & \rightarrow & \text{reg}_a &= \text{reg}_b \\
\text{reg}_a &= 1 & \rightarrow & \text{reg}_c &= \text{reg}_d \\
\text{reg}_b &= 1 & \rightarrow & \text{reg}_a &= \text{reg}_b \\
\ldots \\
\text{reg}_f &= 0 & \rightarrow & \text{reg}_c &= \text{reg}_d
\end{align*}
\]
Register Roles Refine Further

Sample Trace

\[
\begin{align*}
\text{reg}_a &= 1 \\
\text{reg}_b &= 1 \\
\text{reg}_c &= \text{reg}_d
\end{align*}
\]

Mined 2 G(R → R-R)

\[
\begin{align*}
\text{reg}_a &= 1 & \rightarrow & \text{reg}_c &= \text{reg}_d \\
\text{reg}_b &= 1 & \rightarrow & \text{reg}_c &= \text{reg}_d
\end{align*}
\]
Register Slices Uncover Semantic Meaning

Sample Trace

\[
\begin{align*}
\text{reg}_a &= 7 \\
\text{#tick} \\
\text{reg}_a &= 3 \\
\text{#tick} \\
\text{reg}_a &= 5 \\
\end{align*}
\]

Mining \( G(a) \) 

<no properties>
Register Slices Uncover Semantic Meaning

Sample Trace

\[ \text{reg}_a[0]==1 \]
\[ \text{reg}_a[1]==1 \]
#tick
\[ \text{reg}_a[0]==1 \]
\[ \text{reg}_a[1]==1 \]
#tick
\[ \text{reg}_a[0]==1 \]
\[ \text{reg}_a[1]==0 \]

Mining G(a)

\[ \text{reg}_a[0]==1 \]
Tested on 3 Processors

OR1200  mor1kx  RISC-V
Undine: Mining Temporal Properties

Undine can discover linear temporal logic security properties such as those related to correct initialization of a system using a library of typed templates.
Mining Behavior

My research shows the technique of specification mining can find:

- **Temporal** properties, for timing
- **Closed source CISC** architecture properties, for microarchitecture
- **Hyperproperties**, properties over multiple traces of execution

Closed Source CISC

(that’s me)

Temporal

Hyperproperties
How can properties that model secure behavior of closed source complex instruction set computer (CISC) designs be discovered using specification mining?

Mining for control signals in the design then mining preconditioned on those control signals yields security properties of the design.
Recall: Undine Tested on 3 Processors

OR1200

mor1kx

RISC-V
All were Open Source and RISC! x86 is neither!
The x86 specification has many control signals...
Control Signals Partition the Space
So I created a tool to find properties using signals.
Front End: Registers Placed in Groups
Property Refinement

- Hundreds of Registers
- Tens of Registers
- Hundreds of Control Bits
- 24 Control Signals
- 11 Control Signals
Control Signals Partition the Space

Preconditions capturing changes to signals capture transitions between different modes of the processor.

Preconditions holding signals constant capture the behavior defined by a control bit taking on a certain value.
Astarte: Mining Closed Source CISC

Specification mining can discover security properties preconditioned on control signals in closed source CISC designs.
Mining Behavior

My research shows the technique of specification mining can find:

- **Temporal** properties, for timing
- **Closed source CISC architecture** properties, for microarchitecture
- **Hyperproperties**, properties over multiple traces of execution
Isadora: Mining Hyperproperties (Current work)

How can hyperproperties that model secure behavior of designs be discovered using specification mining?
Hyperproperties

Sets of Sets of Traces, or Sets of Properties
Example: GMNI (Noninterference)

“High” could be OS, “Low” could be adversary
Instrumentation

To find hyperproperties, use Information Flow Tracking (IFT) instrumentation.

- IFT creates a shadow register for all design registers to track information flow
- GMNI is an information flow hyperproperty
Problem Statement

How can Information Flow Tracking (IFT) and specification mining determine

**where** and **when**

interference occurs in a design from any arbitrary source?
Tracking Information Flow

- Given a source, registers can be in one of three categories:
  - Always a sink: \( \text{source} = \Rightarrow \text{sink} \) ("flows to")
  - Never a sink: \( \text{source} =/\Rightarrow \text{sink} \) ("does not flow to")
  - Conditionally a sink: \( \text{source} =/\Rightarrow \text{sink} \) UNLESS \(<\text{boolean expression}>\)
Research Technique Sketch

Verilog Design → Tortuga → VCD(s) → MINER First Pass / Second Pass → (Hyper) Properties

Source Register

VCD Miner Front End

ID’ed regs

Daikon
Trace Detail

More than traces!

1. Specify Source
2. Generate Trace and IFT
3. Look at relevant regs
Miner Detail

1. Input Traces
2. Run Miner
3. Get Output
4. Flag interesting shadow_*
   a. shadow_* is IFT state
5. (Re-)Run Miner
6. Output Information Flow
   a. “Always, never, maybe”
Research Technique Sketch

Verilog Design → Tortuga → VCD(s) → MINER

First Pass / Second Pass

VCD Miner Front End

MINER

(Hyper) Properties

Source Register

Daikon

ID'ed regs
Mining in Practice

● Test using *write-address* register
  ○ Always sink 003 regs
  ○ Never sink 189 regs
  ○ Conditional sink 037 regs

● Secondary mining passes can determine conditions under which the 37 conditional sinks are affected by the source register
Isadora: Mining Hyperproperties (Current work)

Hyperproperties that model secure behavior of designs be discovered using specification mining along with Information Flow Tracking (IFT).
Mining Behavior

My research shows the technique of specification mining can find:

- **Temporal** properties, such as correct initialization
- **Closed source CISC** architecture properties, such those over x86-64
- **Hyperproperties**, properties over multiple traces of execution
“Who ya gonna call?”

Cybersecurity for the Spectre Era

Any Questions?