Speculative execution is powerful 😊 ...

char A[16]

if (idx < 16)
    x = load A[idx]
compute(x)

Speculate instead of stalling!

Good prediction: performance gain!
Bad prediction (transient executions): revert changes and continue.
Processor speculates on branch targets, store-to-load dependencies, etc.
... but leads to Spectre attacks 😞

Changes to *microarchitectural state* (e.g. cache) are not reverted!

**Idea.** Force victim to *leak secret data* during transient execution and recover them with *microarchitectural attacks*
Even constant-time programs are vulnerable to Spectre 😞!

**Constant-time**
- Protection against (non-transient) microarchitectural attacks
- No secret-dependent control flow & memory accesses
- Used in many cryptographic implementations

**Constant-Time in the Spectre Era**
- Speculative semantics for software defenses
  - Hard to reason about
  - Accommodate new speculation mechanisms?
Secure Speculation for Constant-Time!

Developers should not care about speculations

Hardware should not speculatively leak secrets

But still be efficient and enables speculation

*Hardware defense: Secure speculation for constant-time!*
How do I know that my defense works?
Hardware-Software Contracts for Secure Speculation

Marco Guarnieri*, Boris Köpf†, Jan Reineke‡, and Pepe Vila*

*IMDEA Software Institute †Microsoft Research ‡Saarland University

Formalize hardware leakage as a **contract**

<table>
<thead>
<tr>
<th><strong>Software side</strong></th>
<th><strong>Hardware side</strong></th>
</tr>
</thead>
</table>
| Program **secure software wrt. contract**  
  • Secure **software** design  
  • Verification  
  • Compilation | Hardware complies with contract  
  • *Formally express guarantees of hardware defenses* |
Hardware-Software Contracts for Secure Speculation

Marco Guarnieri*, Boris Köpf†, Jan Reineke‡, and Pepe Vila*

*IMDEA Software Institute †Microsoft Research ‡Saarland University

Formalize hardware leakage as a contract

Software side

Hardware side

Program secure software wrt. contract

Secure software design

Verification

Compilation

Hardware complies with contract

• Formally express guarantees of hardware defenses

No secure speculation for constant-time programs!
Hardware Secrecy Tracking (HST)

- Inform hardware of what is secret
- Track secret taint in hardware
- Do not leak tainted values during speculation

ConTExT: A Generic Approach for Mitigating Spectre

- Michael Schwarz¹, Moritz Lipp¹, Claudio Camella¹, Robert Schilling¹,², Florian Kargl¹, Daniel Grass¹

  ¹Graz University of Technology ²Know- Center GmbH

SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks

- Jacob Fustos
  University of Kansas

- Farzad Farshchi
  University of Kansas

- Heechul Yun
  University of Kansas

Speculative Privacy Tracking (SPT): Leaking Information From Speculative Execution Without Compromising Privacy

- Rutvik Choudhary
  UIUC, USA

- Jiyong Yu
  UIUC, USA

- Christopher W. Fletcher
  UIUC, USA

- Adam Morrison
  Tel Aviv University, Israel
Hardware Secrecy Tracking (HST)

- Inform hardware of what is secret
- Track secret taint in hardware
- Do not leak tainted values during speculation

Technical implementation details & evaluation

No end-to-end formal security guarantee for constant-time programs
Challenges

• Account for all existing speculation mechanisms
• Account for futuristic speculation mechanisms
• Account for declassification
• Adapt HW/SW contract framework for these new features
• Evaluation: hardware costs?
Our contributions

ProSpeCT: Formal processor model with HST
- **Proof:** constant-time programs do not leak secrets
- Generic: all Spectre variants + LVI
- Allows for *declassification*

First to consider Load Value Speculation
- Novel insight: sometimes need to rollback *correct* speculations for security

Implementation in a RISC-V microarchitecture
- First synthesizable implementation
- Evaluation: hardware cost, performance, annotations
ProSpeCT
Secure Speculation for Constant-Time
**Spectre-v1.** Exploit branch prediction

```c
char A[16]
char secret
if (idx < 16)
    x = load A[idx]
    leak(x)

Consider idx = 16
```

No defense

Mispredicted

x = secret

secret is transiently leaked!
Illustration with Spectre-v1

**Spectre-v1.** Exploit branch prediction

```c
char A[16]  // public memory
char secret  // secret memory
if (idx < 16)
  x = load A[idx]
leak(x)
```

Consider idx = 16

**ProSpeCT**

Developer annotates secret memory
**Illustration with Spectre-v1**

**Spectre-v1.** Exploit branch prediction

```c
char A[16]  // public memory
cchar secret  // secret memory
if (idx < 16)
    x = load A[idx]
leak(x)
```

Consider $idx = 16$

**ProSpeCT**

- Developer annotates secret memory
- Prediction
  - $x = \text{secret}:H$

**Prediction**

Spectre-v1. Exploit branch prediction
Illustration with Spectre-v1

**Spectre-v1.** Exploit branch prediction

```c
char A[16]  // public memory
char secret  // secret memory
if (idx < 16)
  x = load A[idx]
leak(x)
```

Consider `idx = 16`

**ProSpeCT**

- Developer annotates secret memory
- Prediction
- `x = secret:H`
- secret is not forwarded to `leak`
**LVI.** Inject values at faulting loads

```
char A[16]
char secret
x = load idx
y = load A[x]
leak(y)
```

**No defense**

Attacker injects \( x = 16 \)

\( y = \text{secret} \)

\( \text{secret is transiently leaked!} \)

*Akin to Load Value Prediction*
Illustration with LVI

```c
char A[16] // public memory
char secret // secret memory
x = load idx
y = load A[x]
leak(y)
```

**ProSpeCT**
Developer annotates secret memory

_Akin to Load Value Prediction_
Illustration with LVI

```c
char A[16] // public memory
char secret // secret memory
x = load idx
y = load A[x]
leak(y)
```

**ProSpeCT**
- Developer annotates secret memory
- Attacker injects `x = 16`
- `y = secret:H`

_Akin to Load Value Prediction_
Illustration with LVI

```
char A[16]  // public memory
char secret // secret memory
x = load idx
y = load A[x]
leak(y)
```

**ProSpeCT**
- Developer annotates secret memory
- Attacker injects $x = 16$
- $y = \text{secret:} H$
- secret is not forwarded to leak

*Akin to Load Value Prediction*
Design Choices

<table>
<thead>
<tr>
<th>Software side</th>
<th>Hardware side</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Label secret memory</td>
<td>• Track security labels</td>
</tr>
<tr>
<td>• Constant-time program</td>
<td>• Secrets do not speculatively flow to insecure instructions</td>
</tr>
<tr>
<td>• Secret written to public memory is declassified</td>
<td>• Predictions do not leak secrets</td>
</tr>
</tbody>
</table>

Code without secret $\Rightarrow$ **free speculation**
Constant-time programs $\Rightarrow$ **only block mispredictions**
Semantics of out-of-order speculative processor with HST

\[(a, \mu) \xrightarrow{d} (a', \mu')\]

- architectural state
- microarchitectural context
- declassification trace
Semantics of out-of-order speculative processor with HST

$$(a, \mu) \xrightarrow{d} (a', \mu')$$

Abstract microarchitectural context $\mu$
+ Functions $update, predict, next$  \quad \{ \text{Attacker observations} \quad \text{Attacker influence} \}$
Semantics of out-of-order speculative processor with HST

\[(a, \mu) \xrightarrow{d} (a', \mu')\]

Abstract microarchitectural context \(\mu\)
+ Functions update, predict, next

At each step: \(\mu\) is updated with all public values
→ predictions can depend on any public value
Secure Speculation for Constant-Time Policy

Security (no declassification).

For all constant-time program (architectural semantics)

if \( a_0 = \text{public } a'_0 \) and \( (a_0, \mu) \rightarrow^n (a_n, \mu_n) \)

then \( (a'_0, \mu) \rightarrow^n (a'_n, \mu'_n) \) and \( \mu_n = \mu'_n \)

Architectural semantics = hardware software security contract
Secure Speculation for Constant-Time Policy

Security (with declassification).

For all constant-time program up to declassification

if $a_0 =_{\text{public}} a'_0$ and $(a_0, \mu) \xrightarrow{d} (a_n, \mu_n)$

then $(a'_0, \mu), d \xleftarrow{n} (a'_n, \mu'_n)$ and $\mu_n = \mu'_n$

Declassify ciphertext while still protecting plaintext
Load Prediction: Rollback correct executions?

```
char secret // secret memory
x = load secret
y = x + 4
```

**Execution 1:** secret=0

```
x = 0 (?); y = 4
```

**Prediction:** load value to 0

**Execution 2:** secret=1

```
x = 0 (?); y = 4
```
Load Prediction: Rollback correct executions?

```
char secret // secret memory
x = load secret
y = x + 4
```

### Execution 1: secret=0

Predict load value to 0

- $x = 0$ (?)
- $y = 4$

Resolve

- $x = 0$
- $y = 4$

Commit if secret = 0 vs Rollback if secret ≠ 0

$\Rightarrow$ Implicit resolution-based channel

### Execution 2: secret=1

Predict load value to 0

- $x = 0$ (?)
- $y = 4$

Resolve

- $x = 1$
**Load Prediction: Rollback correct executions?**

```c
char secret; // secret memory
x = load secret
y = x + 4
```

<table>
<thead>
<tr>
<th>Execution 1: secret=0</th>
<th>Execution 2: secret=1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x = 0 (?); y = 4</code></td>
<td><code>x = 0 (?); y = 4</code></td>
</tr>
<tr>
<td><code>x = 0:H</code></td>
<td><code>x = 1:H</code></td>
</tr>
</tbody>
</table>

**Predict load value to 0**

**Resolve**

**Solution:** Always rollback when actual value is `secret`
Implementation and Evaluation
Prototype Risc-V implementation

• Firsts synthesizable implementation

• On top of Proteus modular RiSC-V processor

• Open-sourced on github!

• Limitation
  • Only branch prediction
  • Secrets not forwarded *at all* during speculation (conservative)
Evaluation: Labelling Secrets

Inform hardware about secrets?

Secret are labelled in source and co-located in binary
Boundaries stored in CSRs
  - Currently supporting up to 2 separate regions
  - Easy to change

Evaluation: is annotation easy?

Need to mark secret in source
Need avoid stack spilling!

<table>
<thead>
<tr>
<th></th>
<th>LoC</th>
<th>S</th>
<th>A_m</th>
<th>A_a</th>
<th>I</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>djbsort</td>
<td>246</td>
<td>L</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>Constant-time sort</td>
</tr>
<tr>
<td>sha256</td>
<td>1795</td>
<td>L</td>
<td>34</td>
<td>0</td>
<td>6</td>
<td>Hash function</td>
</tr>
<tr>
<td>chacha20</td>
<td>1864</td>
<td>L</td>
<td>51</td>
<td>0</td>
<td>6</td>
<td>Encryption</td>
</tr>
<tr>
<td>curve25519</td>
<td>3026</td>
<td>H</td>
<td>967</td>
<td>0</td>
<td>6</td>
<td>Elliptic curve</td>
</tr>
</tbody>
</table>
Evaluation: Hardware

Hardware implementation

• Proteus is written in SpinalHDL
• ≈5000 lines of Scala code
• Changes for ProSpeCT: ≈ 400 lines

Hardware costs

• LUTs: 16,847 → 19,728 (+17%)
• Registers: 11,913 → 12,600 (+6%)
• Critical path: 30.1 ns → 30.7 ns (+2%)
Runtime Overhead

**Benchmark** [1]

- Amount of secret
- Speculation-heavy public computations / crypto

<table>
<thead>
<tr>
<th>spec/crypto</th>
<th>25/75</th>
<th>50/50</th>
<th>75/25</th>
<th>90/10</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Secret</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>All</td>
<td>110%</td>
<td>125%</td>
<td>136%</td>
<td>145%</td>
</tr>
</tbody>
</table>

**Conclusion**

Results similar to [1]

Precise annotation + restricted secret computations = **Low overhead**

[1] Jacob Fustos, Farzad Farshchi, and Heechul Yun. “SpectreGuard: An Efficient Data-Centric Defense Mechanism against Spectre Attacks”. In: DAC. 2019
Conclusion

Hardware Secrecy Tracking

Software informs hardware about secret

Strong security guarantees

ProSpeCT $\implies$ end-to-end security for constant-time programs

Low overhead

ProSpeCT $\implies$ no runtime overhead on public data

Check our paper on arXiv!
Future Work?

**Formal model**
- Cryptographic security down to the hardware?

**Compiler-support**
- Separate secret from public memory
- Ensure no unintentional declassification

**Validate RISC-V implementation**
- Contract-based CPU testing (e.g., Revizor, Scam-V)?
- Hardware-fuzzing / Model checking / Formal methods?
Icons made by Freepik from www.flaticon.com

Diamond icons created by Vectors Market – Flaticon www.flaticon.com/free-icons/diamond

Hard work icon created by monkik – Flaticon www.flaticon.com/free-icons/hard-work