Verification of Spectre Mitigations using Hypersimulations

Jonathan Baumann

August 09, 2023

UNIVERSITÄT
DES
SAARLANDES

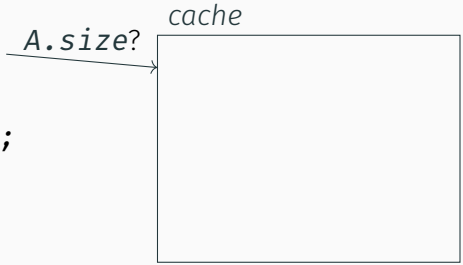
Spectre

```
if (i < A.size && i > 0) {
        int x = A[i];
        int y = B[x * 256];
}
```

cache

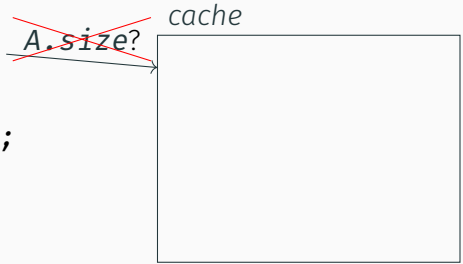
Spectre

```
▶ if (i < A.size && i > 0) {
        int x = A[i];
        int y = B[x * 256];
  }
```



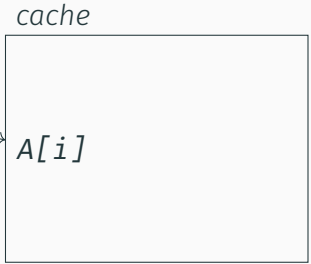
Spectre

```
if (i < A.size && i > 0) {
        int x = A[i];
        int y = B[x * 256];
}
```



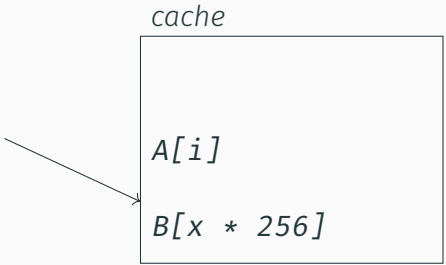
Spectre

```
▷ if (i < A.size && i > 0) {
      ▶ int x = A[i];
        int y = B[x * 256];
  }
```



Spectre

```
▷ if (i < A.size && i > 0) {
        int x = A[i];
      ▶ int y = B[x * 256];
  }
```



Spectre

```
▷ if (i < A.size && i > 0) {
        int x = A[i];
        int y = B[x * 256];
▶ }
```

cache

A[i]
*B[x * 256]*

Spectre

```
if (i < A.size && i > 0) {
        int x = A[i];
        int y = B[x * 256];
}
```

cache

A.size
A[i]
*B[x * 256]*

Spectre

```
if (i < A.size && i > 0) {
        int x = A[i];
        int y = B[x * 256];
}
```

cache

A.size
A[i]
*B[x * 256]*

Spectre Mitigations

- Hardware Mitigations

Spectre Mitigations

- Hardware Mitigations

$\Longrightarrow$ costly, can only protect future hardware

Spectre Mitigations

- Hardware Mitigations
- Software mitigations

Spectre Mitigations

- Hardware Mitigations
- Software mitigations
  - Program Analysis and Repair

Spectre Mitigations

- Hardware Mitigations
- Software mitigations
  - Program Analysis and Repair
    - oo7: taint tracking

Spectre Mitigations

- Hardware Mitigations
- Software mitigations
  - Program Analysis and Repair
    - oo7: taint tracking
    - SPECTECTOR: symbolic execution

Spectre Mitigations

- Hardware Mitigations
- Software mitigations
  - Program Analysis and Repair
  - Compiler Countermeasures

Spectre Mitigations

- Hardware Mitigations
- Software mitigations
  - Program Analysis and Repair
  - Compiler Countermeasures
    - Intel: Fence instructions after branches

Spectre Mitigations

- Hardware Mitigations
- Software mitigations
  - Program Analysis and Repair
  - Compiler Countermeasures
    - Intel: Fence instructions after branches
    - Speculative Load Hardening: additional data dependencies

Spectre Mitigations

- Hardware Mitigations
- Software mitigations
  - Program Analysis and Repair
  - Compiler Countermeasures
    - Intel: Fence instructions after branches
    - Speculative Load Hardening: additional data dependencies
    - MSVC: Fence instructions, pattern-based

Contributions

- Method of verifying compiler-based spectre mitigations

Contributions

- Method of verifying compiler-based spectre mitigations
  - applicable to a variety of leakage models

Contributions

- Method of verifying compiler-based spectre mitigations
  - applicable to a variety of leakage models
  - capable of verifying more mitigations

Contributions

- Method of verifying compiler-based spectre mitigations
  - applicable to a variety of leakage models
  - capable of verifying more mitigations
- Verify Intel's mitigation for three leakage models

Contributions

- Method of verifying compiler-based spectre mitigations
  - applicable to a variety of leakage models
  - capable of verifying more mitigations
- Verify Intel's mitigation for three leakage models
- Incompatibility between speculative load hardening and always-mispredict semantics

Contributions

- Method of verifying compiler-based spectre mitigations
  - applicable to a variety of leakage models
  - capable of verifying more mitigations
- Verify Intel's mitigation for three leakage models
- Incompatibility between speculative load hardening and always-mispredict semantics
- Design and verify a mitigation that could not be verified previously

Contributions

- Method of verifying compiler-based spectre mitigations
  - applicable to a variety of leakage models
  - capable of verifying more mitigations
- Verify Intel's mitigation for three leakage models
- Incompatibility between speculative load hardening and always-mispredict semantics
- Design and verify a mitigation that could not be verified previously
- All work formalized in the Coq Proof Assistant

Verifying Spectre Mitigations: Speculative Noninterference [Guarnieri et al., 2018]

Noninterference

Same public inputs $\Longrightarrow$ Same observations

Verifying Spectre Mitigations: Speculative Noninterference [Guarnieri et al., 2018]

Noninterference

Same public inputs $\Longrightarrow$ Same observations

Speculative Noninterference

Same observations under nonspeculative execution $\Longrightarrow$ Same observations under speculative execution

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

Applying a mitigation: Compilation

Language without speculative execution $\longrightarrow$ Language with speculative execution

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

Applying a mitigation: Compilation

Language without speculative execution $\longrightarrow$ Language with speculative execution

- Without speculation, speculative noninterference is always satisfied

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

Applying a mitigation: Compilation

Language without speculative execution $\longrightarrow$ Language with speculative execution

- Without speculation, speculative noninterference is always satisfied
- Speculative noninterference should be preserved

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

Applying a mitigation: Compilation

Language without speculative execution $\longrightarrow$ Language with speculative execution

- Without speculation, speculative noninterference is always satisfied
- Speculative noninterference should be preserved
- Approximate using speculative safety

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

- Speculative safety only defined for constant-time leakage model

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

- Speculative safety only defined for constant-time leakage model
  - not applicable to other leakage models

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

- Speculative safety only defined for constant-time leakage model
  - not applicable to other leakage models
- Speculatively noninterferent code may not be speculatively safe:

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

- Speculative safety only defined for constant-time leakage model
  - not applicable to other leakage models
- Speculatively noninterferent code may not be speculatively safe:

$$\texttt{read}\ [1]\ x;\ \texttt{if}\ x \geq 0\ \texttt{then}\ \texttt{read}\ [x]\ y\ \texttt{else}\ \texttt{read}\ [x]\ y\ \texttt{end};\ \texttt{write}\ [1]\ y$$

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

- Speculative safety only defined for constant-time leakage model
  - not applicable to other leakage models
- Speculatively noninterferent code may not be speculatively safe:

$$\texttt{read}\ [1]\ x;\ \texttt{if}\ x \geq 0\ \texttt{then read}\ [x]\ y\ \texttt{else read}\ [x]\ y\ \texttt{end};\ \texttt{write}\ [1]\ y$$

  - not all mitigations can be verified

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

- Speculative safety only defined for constant-time leakage model
  - not applicable to other leakage models
- Speculatively noninterferent code may not be speculatively safe:

$$\texttt{read}\ [1]\ x;\ \texttt{if}\ x \geq 0\ \texttt{then read}\ [x]\ y\ \texttt{else read}\ [x]\ y\ \texttt{end};\ \texttt{write}\ [1]\ y$$

  - not all mitigations can be verified
- Our approach does not have these limitations

Verifying Spectre Mitigations: Preservation of Leakage Equivalence

$$
\begin{tikzcd}
\text{Source Speculative Noninterference} \arrow[r, Leftrightarrow] \arrow[d, Rightarrow, color=red] & \text{Source Property} \arrow[d, Rightarrow] \\
\text{Target Speculative Noninterference} & \text{Target Property} \arrow[l, Leftarrow]
\end{tikzcd}
$$

Verifying Spectre Mitigations: Preservation of Leakage Equivalence

$$\begin{tikzcd}
\text{Source Speculative Noninterference} \arrow[r, Leftrightarrow] \arrow[d, Rightarrow] & \text{Source Property} \arrow[d, Rightarrow] \\
\text{Target Speculative Noninterference} & \text{Target Property} \arrow[l, Leftarrow]
\end{tikzcd}$$

Speculative Noninterference

Same observations under nonspeculative execution $\Longrightarrow$ Same observations under speculative execution

Verifying Spectre Mitigations: Preservation of Leakage Equivalence

$$
\begin{tikzcd}
\text{Source Speculative Noninterference} \arrow[r, Leftrightarrow] \arrow[d, Rightarrow] & \text{Source Property} \arrow[d, Rightarrow] \\
\text{Target Speculative Noninterference} & \text{Target Property} \arrow[l, Leftarrow]
\end{tikzcd}
$$

Speculative Noninterference

Same observations under nonspeculative execution of source program $\Longrightarrow$ Same observations under speculative execution of mitigated program

Verifying Spectre Mitigations: Preservation of Leakage Equivalence

Speculative Noninterference

Source Property

$\Downarrow$

Target Property

Speculative Noninterference

Same observations under nonspeculative execution of source program $\Longrightarrow$ Same observations under speculative execution of mitigated program

Verifying Spectre Mitigations: Preservation of Leakage Equivalence

Speculative Noninterference

Source Leakage Equivalence

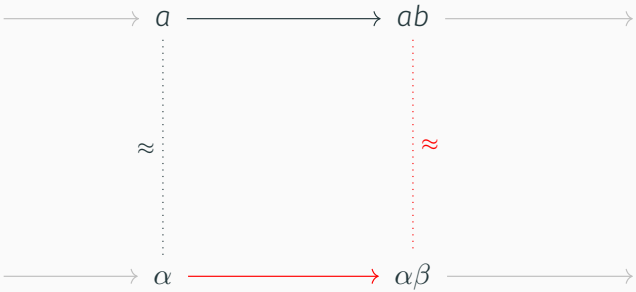
$\Downarrow$

Target Leakage Equivalence

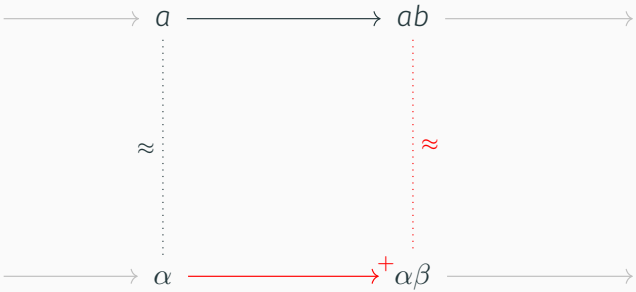
Speculative Noninterference

Leakage equivalence under nonspeculative execution of source program $\Longrightarrow$ Leakage Equivalence under speculative execution of mitigated program

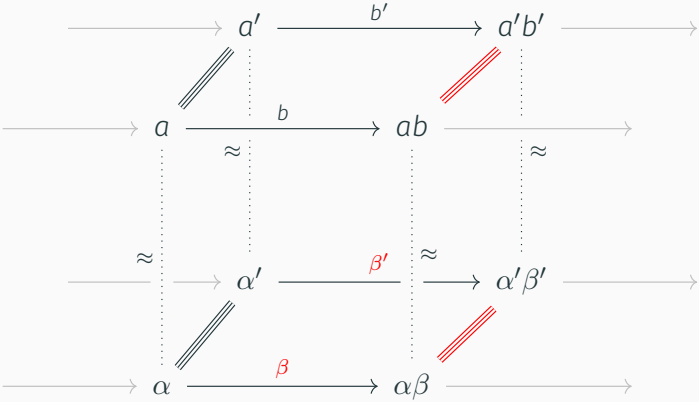
Preservation of Hyperproperties: Constant-Time Simulations [Barthe et al., 2018]



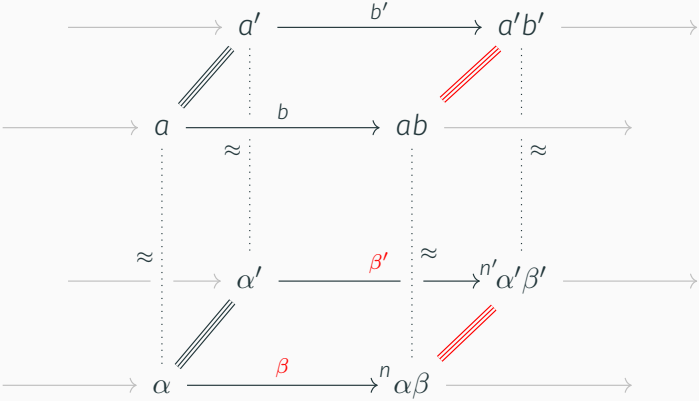
Preservation of Hyperproperties: Constant-Time Simulations [Barthe et al., 2018]



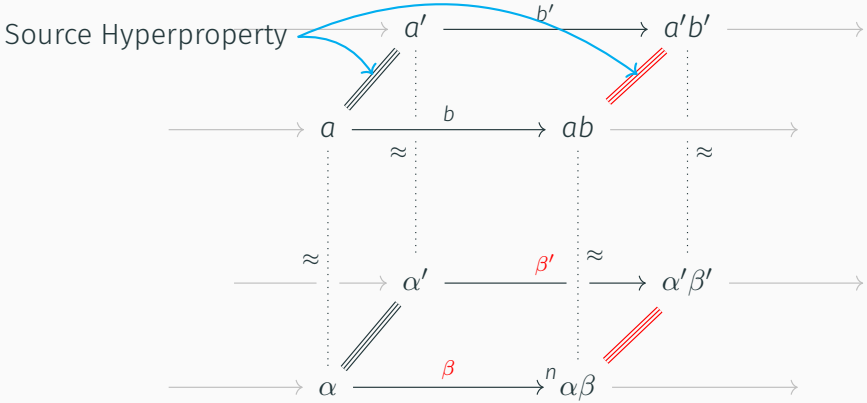
Preservation of Hyperproperties: Constant-Time Simulations [Barthe et al., 2018]



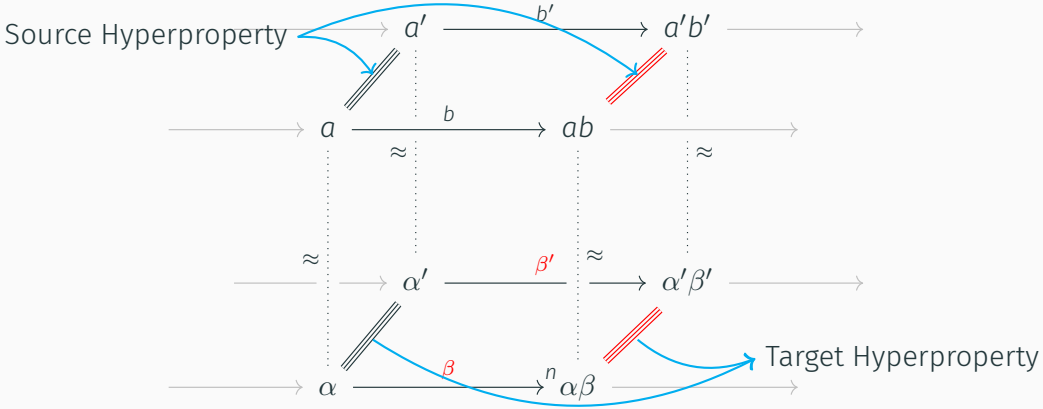
Preservation of Hyperproperties: Constant-Time Simulations [Barthe et al., 2018]



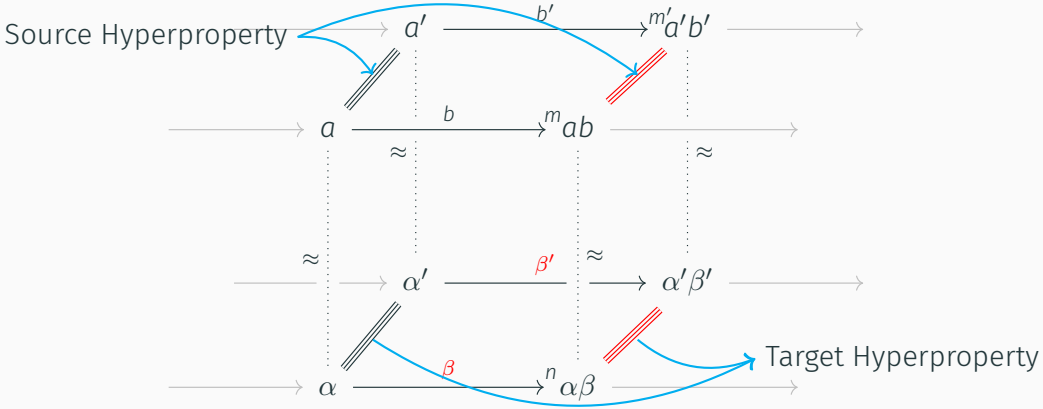
Preservation of Hyperproperties: Constant-Time Simulations [Barthe et al., 2018]



Preservation of Hyperproperties: Constant-Time Simulations [Barthe et al., 2018]



Preservation of Hyperproperties: Hyper-Simulations [Rosemann, 2023]



Leakage Equivalence as a Property of Trace Prefixes

- Traces are infinite

Leakage Equivalence as a Property of Trace Prefixes

- Traces are infinite
- Hypersimulation handles finite prefixes

Leakage Equivalence as a Property of Trace Prefixes

- Traces are infinite
- Hypersimulation handles finite prefixes

$\Longrightarrow$ Define leakage equivalence with respect to trace prefixes

Leakage Equivalence as a Property of Trace Prefixes

- Traces are infinite
- Hypersimulation handles finite prefixes

$\Longrightarrow$ Define leakage equivalence with respect to trace prefixes

Leakage Equivalence Traces t_1 and t_2 are leakage equivalent iff

Leakage Equivalence as a Property of Trace Prefixes

- Traces are infinite
- Hypersimulation handles finite prefixes

$\Longrightarrow$ Define leakage equivalence with respect to trace prefixes

Leakage Equivalence Traces t_1 and t_2 are leakage equivalent iff

- for all finite prefixes p_1 and p_2

Leakage Equivalence as a Property of Trace Prefixes

- Traces are infinite
- Hypersimulation handles finite prefixes

$\Longrightarrow$ Define leakage equivalence with respect to trace prefixes

Leakage Equivalence Traces t_1 and t_2 are leakage equivalent iff

- for all finite prefixes p_1 and p_2
- when computing the leakages $\ell(p_1)$ and $\ell(p_2)$

Leakage Equivalence as a Property of Trace Prefixes

- Traces are infinite
- Hypersimulation handles finite prefixes

$\Longrightarrow$ Define leakage equivalence with respect to trace prefixes

Leakage Equivalence Traces t_1 and t_2 are leakage equivalent iff

- for all finite prefixes p_1 and p_2
- when computing the leakages $\ell(p_1)$ and $\ell(p_2)$
- $\ell(p_1)$ is a prefix of $\ell(p_2)$ or vice versa

Leakage Equivalence as a Property of Trace Prefixes

- Traces are infinite
- Hypersimulation handles finite prefixes

$\Longrightarrow$ Define leakage equivalence with respect to trace prefixes

Leakage Equivalence Traces t_1 and t_2 are leakage equivalent iff

- for all finite prefixes p_1 and p_2
- when computing the leakages $\ell(p_1)$ and $\ell(p_2)$
- $\ell(p_1)$ is a prefix of $\ell(p_2)$ or vice versa
- we can extend the prefixes such that the leakages are equal

Leakage Equivalence as a Property of Trace Prefixes

- Traces are infinite
- Hypersimulation handles finite prefixes

$\Longrightarrow$ Define leakage equivalence with respect to trace prefixes

Leakage Equivalence Traces t_1 and t_2 are leakage equivalent iff

$$p_1 \lesssim^{+}_{\ell i} p_2 \quad \begin{cases} \text{for all finite prefixes } p_1 \text{ and } p_2 \\ \text{when computing the leakages } \ell(p_1) \text{ and } \ell(p_2) \\ \ell(p_1) \text{ is a prefix of } \ell(p_2) \text{ or vice versa} \\ \text{we can extend the prefixes such that the leakages are equal} \end{cases}$$

Proving Preservation of Leakage Equivalence

- Find simulation relation

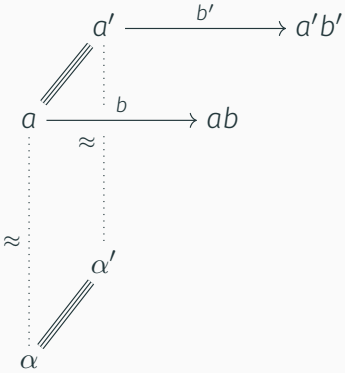
Proving Preservation of Leakage Equivalence

- Find simulation relation
- Pick source and target relations

Proving Preservation of Leakage Equivalence

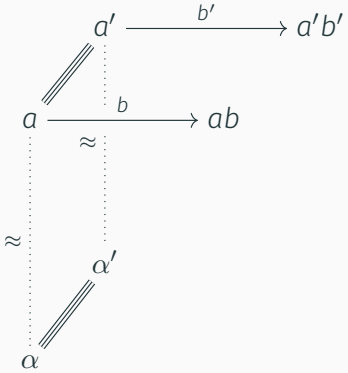
- Find simulation relation
- Pick source and target relations
  - typically: trace prefixes should have the *exact same* leakage

Proving Preservation of Leakage Equivalence



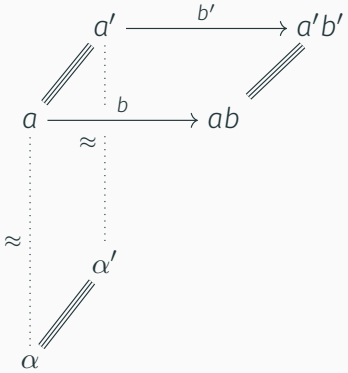
- Find simulation relation
- Pick source and target relations
  - typically: trace prefixes should have the *exact same* leakage
- Determine which source states should be related

Proving Preservation of Leakage Equivalence



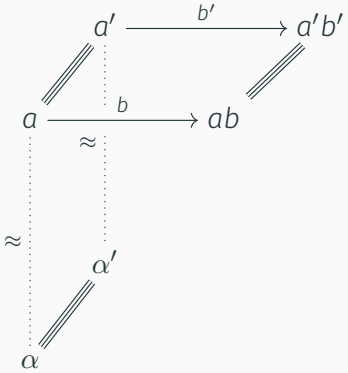
- Find simulation relation
- Pick source and target relations
  - typically: trace prefixes should have the *exact same* leakage
- Determine which source states should be related
  - typically: synchronize on leakage-producing states

Proving Preservation of Leakage Equivalence



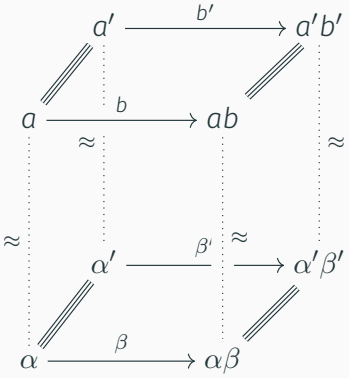
- Find simulation relation
- Pick source and target relations
  - typically: trace prefixes should have the *exact same* leakage
- Determine which source states should be related
  - typically: synchronize on leakage-producing states
- Prove that source relation is satisfied after hyper-step

Proving Preservation of Leakage Equivalence



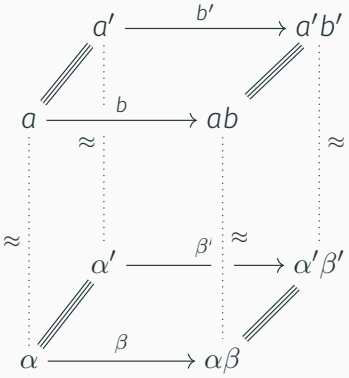
- Find simulation relation
- Pick source and target relations
  - typically: trace prefixes should have the *exact same* leakage
- Determine which source states should be related
  - typically: synchronize on leakage-producing states
- Prove that source relation is satisfied after hyper-step
  - follows directly from leakage equivalence

Proving Preservation of Leakage Equivalence



- Find simulation relation
- Pick source and target relations
  - typically: trace prefixes should have the *exact same* leakage
- Determine which source states should be related
  - typically: synchronize on leakage-producing states
- Prove that source relation is satisfied after hyper-step
  - follows directly from leakage equivalence
- Prove that target relation is satisfied after hyper-step

Proving Preservation of Leakage Equivalence

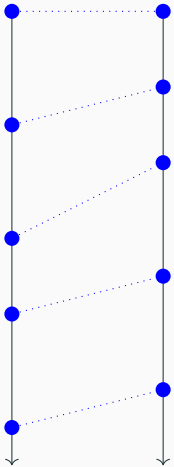


- Find simulation relation
- Pick source and target relations
  - typically: trace prefixes should have the *exact same* leakage
- Determine which source states should be related
  - typically: synchronize on leakage-producing states
- Prove that source relation is satisfied after hyper-step
  - follows directly from leakage equivalence
- Prove that target relation is satisfied after hyper-step
- Conclude that target traces are leakage equivalent

Proving Preservation of Leakage Equivalence

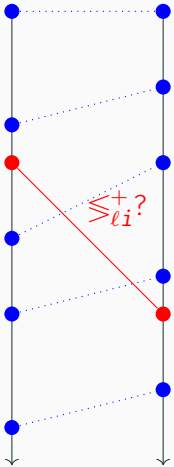
Proving Preservation of Leakage Equivalence

- From hypersimulation: target relation *always holds at some future point*



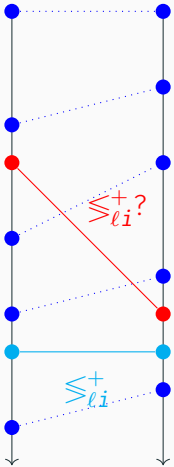
Proving Preservation of Leakage Equivalence

- From hypersimulation: target relation *always holds at some future point*
- $\lesssim^{+}_{\ell i}$ must be satisfied *at every point during execution*



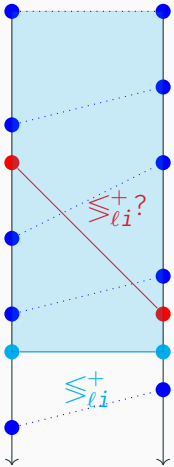
Proving Preservation of Leakage Equivalence

- From hypersimulation: target relation *always holds at some future point*
- $\lesssim^{+}_{\ell i}$ must be satisfied *at every point during execution*
- Leakage equivalence is a safety hyperproperty



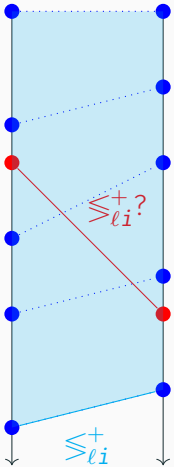
Proving Preservation of Leakage Equivalence

- From hypersimulation: target relation *always holds at some future point*
- $\lesssim^{+}_{\ell i}$ must be satisfied *at every point during execution*
- Leakage equivalence is a safety hyperproperty
  - $\lesssim^{+}_{\ell i}$ satisfied at some point
    $\implies$ satisfied at all prior points



Proving Preservation of Leakage Equivalence

- From hypersimulation: target relation *always holds at some future point*
- $\lesssim^{+}_{\ell i}$ must be satisfied *at every point during execution*
- Leakage equivalence is a safety hyperproperty
  - $\lesssim^{+}_{\ell i}$ satisfied at some point
    $\implies$ satisfied at all prior points
- Target relation implies $\lesssim^{+}_{\ell i}$
  $\implies$ $\lesssim^{+}_{\ell i}$ must hold everywhere



Modelling Speculative Execution: Toy Language

$$
\begin{aligned}
Expr \ni e ::= \; & n && n \in \mathbb{N} \\
\mid \; & x && x \text{ is a variable name} \\
\mid \; & e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1 < e_2 \mid e_1 = e_2 \mid !e
\end{aligned}
$$

Modelling Speculative Execution: Toy Language

$$
\begin{aligned}
\mathit{Expr} \ni e ::= {} & n && n \in \mathbb{N} \\
\mid {} & x && x \text{ is a variable name} \\
\mid {} & e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1 < e_2 \mid e_1 = e_2 \mid !e
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Stmt} \ni s ::= {} & \texttt{skip} && \mid x := e \\
\mid {} & \texttt{fence} && \mid \texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end} \\
\mid {} & \texttt{read}\ [e]\ x && \mid \texttt{while}\ e\ \texttt{do}\ p_1\ \texttt{finally}\ p_2\ \texttt{end} \\
\mid {} & \texttt{write}\ [e]\ x
\end{aligned}
$$

Modelling Speculative Execution: Speculative Semantics

- Note: we translate `while` to `if`

Modelling Speculative Execution: Speculative Semantics

- Note: we translate `while` to `if`
- Stack of states to allow rollback

Modelling Speculative Execution: Speculative Semantics

- Note: we translate `while` to `if`
- Stack of states to allow rollback
- Speculation window determines how long to follow mispredicted paths

Modelling Speculative Execution: Speculative Semantics

- Note: we translate `while` to `if`
- Stack of states to allow rollback
- Speculation window determines how long to follow mispredicted paths
- Always-Mispredict semantics [Guarnieri et al., 2018] captures all possible combinations of branch predictions

Modelling Speculative Execution: Speculative Semantics

$$\left\langle \bot \mid [x \mapsto -1] \mid [-1 \mapsto 42] \mid \begin{array}{l} \texttt{if}\ x > 0 \\ \texttt{then}\ \texttt{if}\ x < 10 \\ \qquad \texttt{then}\ \texttt{read}\ [x]\ y \\ \qquad \texttt{else}\ \texttt{fence} \\ \qquad \texttt{end} \\ \texttt{else}\ \texttt{skip} \\ \texttt{end} \end{array} \right\rangle$$

Modelling Speculative Execution: Speculative Semantics

$$\frac{\text{no-rollback}(n) \qquad [\![b]\!]_{\mathcal{V}} = 0}{\begin{array}{l}\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end}; p\rangle :: S \\ \rightarrow_{\text{sp}} \langle \text{wndw}(n) \mid \mathcal{V} \mid \mathcal{H} \mid p_1 \mathbin{+\!\!+} p\rangle :: \langle \text{decr}(n) \mid \mathcal{V} \mid \mathcal{H} \mid p_2 \mathbin{+\!\!+} p\rangle :: S\end{array}} \ \textsc{AMIfFalse}$$

$$\left\langle \bot \mid [x \mapsto -1] \mid [-1 \mapsto 42] \mid \begin{array}{l}\texttt{if}\ x > 0 \\ \texttt{then}\ \texttt{if}\ x < 10 \\ \qquad \texttt{then}\ \texttt{read}\ [x]\ y \\ \qquad \texttt{else}\ \texttt{fence} \\ \qquad \texttt{end} \\ \texttt{else}\ \texttt{skip} \\ \texttt{end}\end{array} \right\rangle$$

Modelling Speculative Execution: Speculative Semantics

$$\frac{\text{no-rollback}(n) \qquad [\![b]\!]_{\mathcal{V}} = 0}{\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end}; p\rangle :: S \to_{\text{sp}} \langle \text{wndw}(n) \mid \mathcal{V} \mid \mathcal{H} \mid p_1 \mathbin{+\!\!+} p\rangle :: \langle \text{decr}(n) \mid \mathcal{V} \mid \mathcal{H} \mid p_2 \mathbin{+\!\!+} p\rangle :: S}\ \textsc{AMIfFalse}$$

```
if x > 0
then if x < 10
     then read [x] y
     else fence
     end
else skip
end
```

$$\left\langle 16 \mid [x \mapsto -1] \mid [-1 \mapsto 42] \mid \begin{array}{l} \texttt{if}\ x < 10 \\ \texttt{then read}\ [x]\ y \\ \texttt{else fence end} \end{array} \right\rangle$$

$$\langle \bot \mid [x \mapsto -1] \mid [-1 \mapsto 42] \mid \texttt{skip}\rangle$$

Modelling Speculative Execution: Speculative Semantics

```
if x > 0
then if x < 10
     then read [x] y
     else fence
     end
else skip
end
```

$$\langle 15 \mid [x \mapsto -1] \mid [-1 \mapsto 42] \mid \mathtt{read}\ [x]\ y \rangle$$

$$\langle 15 \mid [x \mapsto -1] \mid [-1 \mapsto 42] \mid \mathtt{fence} \rangle$$

$$\langle \bot \mid [x \mapsto -1] \mid [-1 \mapsto 42] \mid \mathtt{skip} \rangle$$

Modelling Speculative Execution: Speculative Semantics

```
if x > 0
then if x < 10
     then read [x] y
     else fence
     end
else skip
end
```

$$\langle 14 \mid [x \mapsto -1, y \mapsto 42] \mid [-1 \mapsto 42] \mid [] \rangle$$

$$\langle 15 \mid [x \mapsto -1] \mid [-1 \mapsto 42] \mid \texttt{fence} \rangle$$

$$\langle \bot \mid [x \mapsto -1] \mid [-1 \mapsto 42] \mid \texttt{skip} \rangle$$

Modelling Speculative Execution: Speculative Semantics

```
if x > 0
then if x < 10
     then read [x] y
     else fence
     end
else skip
end
```

$$\langle 15 \mid [x \mapsto -1] \mid [-1 \mapsto 42] \mid \texttt{fence} \rangle$$

$$\langle \bot \mid [x \mapsto -1] \mid [-1 \mapsto 42] \mid \texttt{skip} \rangle$$

Modelling Speculative Execution: Speculative Semantics

```
if x > 0
then if x < 10
     then read [x] y
     else fence
     end
else skip
end
```

$$\langle 0 \mid [x \mapsto -1] \mid [-1 \mapsto 42] \mid [] \rangle$$

$$\langle \bot \mid [x \mapsto -1] \mid [-1 \mapsto 42] \mid \mathit{skip} \rangle$$

Modelling Speculative Execution: Speculative Semantics

```
if x > 0
then if x < 10
     then read [x] y
     else fence
     end
else skip
end
```

$$\langle \bot \mid [x \mapsto -1] \mid [-1 \mapsto 42] \mid \mathit{skip} \rangle$$

Modelling Speculative Execution: Speculative Semantics

```
if x > 0
then if x < 10
     then read [x] y
     else fence
     end
else skip
end
```

$$\langle \bot \mid [x \mapsto -1] \mid [-1 \mapsto 42] \mid [] \rangle$$

Modelling Side-Channel Vulnerabilities: Leakage Models

Leakage models: functions from sequences of states to sequences of observations

Modelling Side-Channel Vulnerabilities: Leakage Models

Leakage models: functions from sequences of states to sequences of observations

$$\begin{tikzcd}
\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{while}\ldots\rangle \arrow[r] & \langle \mathcal{V} \mid \mathcal{H} \mid \texttt{if}\ldots\rangle \arrow[r] \arrow[d] & \langle \mathcal{V} \mid \mathcal{H} \mid \texttt{read}\ldots\rangle \arrow[r] \arrow[d] & \langle \mathcal{V} \mid \mathcal{H} \mid \texttt{write}\ldots\rangle \arrow[r] \arrow[d] & \cdots \\
 & \text{branch } \top \arrow[r] & \text{read } \mathit{addr} \arrow[r] & \text{write } \mathit{addr} \arrow[r] & \cdots
\end{tikzcd}$$

ℓ_{ct} control flow, memory accesses

Modelling Side-Channel Vulnerabilities: Leakage Models

Leakage models: functions from sequences of states to sequences of observations

$$\begin{tikzcd}
\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{while}\ldots\rangle \arrow[r] \arrow[d] & \langle \mathcal{V} \mid \mathcal{H} \mid \texttt{if}\ldots\rangle \arrow[r] & \langle \mathcal{V} \mid \mathcal{H} \mid \texttt{read}\ldots\rangle \arrow[r] \arrow[d] & \langle \mathcal{V} \mid \mathcal{H} \mid \texttt{write}\ldots\rangle \arrow[r] \arrow[d] & \cdots \\
\text{loop} \arrow[rr] & & \text{read } \mathit{addr} \arrow[r] & \text{write } \mathit{addr} \arrow[r] & \cdots
\end{tikzcd}$$

ℓ_{ct} control flow, memory accesses

ℓ_{lm} loop headers, memory accesses

Modelling Side-Channel Vulnerabilities: Leakage Models

Leakage models: functions from sequences of states to sequences of observations

$$\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{while}\ldots\rangle \to \langle \mathcal{V} \mid \mathcal{H} \mid \texttt{if}\ldots\rangle \to \langle \mathcal{V} \mid \mathcal{H} \mid \texttt{read}\ldots\rangle \to \langle \mathcal{V} \mid \mathcal{H} \mid \texttt{write}\ldots\rangle \to \cdots$$

$$\longrightarrow \qquad \longrightarrow \text{read } \mathit{addr} \longrightarrow \text{write } \mathit{addr} \longrightarrow \cdots$$

ℓ_{ct} control flow, memory accesses

ℓ_{lm} loop headers, memory accesses

ℓ_{mem} only memory accesses

Verifying Intel's Mitigation

Mitigation proposed by Intel [Intel, 2018, 2021]:
Insertion of Fence instructions after every branch

Verifying Intel's Mitigation

Mitigation proposed by Intel [Intel, 2018, 2021]:
Insertion of Fence instructions after every branch

$$
\begin{aligned}
(\!|\texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end}; p|\!)_{\text{fence}} &= \texttt{if}\ b\ \texttt{then fence}; (\!|p_1|\!)_{\text{fence}} \\
&\quad \texttt{else fence}; (\!|p_2|\!)_{\text{fence}}\ \texttt{end}; (\!|p|\!)_{\text{fence}} \\
(\!|\texttt{while}\ b\ \texttt{do}\ p_1\ \texttt{finally}\ p_2\ \texttt{end}; p|\!)_{\text{fence}} &= \texttt{while}\ b\ \texttt{do fence}; (\!|p_1|\!)_{\text{fence}} \\
&\quad \texttt{finally fence}; (\!|p_2|\!)_{\text{fence}}\ \texttt{end}; (\!|p|\!)_{\text{fence}} \\
(\!|s; p|\!)_{\text{fence}} &= s; (\!|p|\!)_{\text{fence}} \qquad \text{otherwise} \\
(\!|[]|\!)_{\text{fence}} &= []
\end{aligned}
$$

Verifying Intel's Mitigation

$(\!|\cdot|\!)_{\mathsf{fence}}$ is proven secure under

Verifying Intel's Mitigation

$(\!|\cdot|\!)_{\mathsf{fence}}$ is proven secure under

ℓ_{ct} Constant-time leakage model
(*confirming result by Patrignani and Guarnieri [2021]*)

Verifying Intel's Mitigation

$(\!|\cdot|\!)_{\mathsf{fence}}$ is proven secure under

- ℓ_{ct} Constant-time leakage model
  (*confirming result by Patrignani and Guarnieri [2021]*)
- ℓ_{lm} Leakage model including loop headers, but not all control flow
  (*new result*)

Verifying Intel's Mitigation

$(\!|\cdot|\!)_{\mathsf{fence}}$ is proven secure under

- ℓ_{ct} Constant-time leakage model
  (*confirming result by Patrignani and Guarnieri [2021]*)
- ℓ_{lm} Leakage model including loop headers, but not all control flow
  (*new result*)
- ℓ_{mem} Leakage model without any control flow
  (*new result*)

Issues with Speculative Load Hardening

Speculative Load Hardening [Carruth, 2018]:
Protect memory access with artificial data dependencies

Issues with Speculative Load Hardening

Speculative Load Hardening [Carruth, 2018]:
Protect memory access with artificial data dependencies

- Special register accumulates path conditions

Issues with Speculative Load Hardening

Speculative Load Hardening [Carruth, 2018]:
Protect memory access with artificial data dependencies

- Special register accumulates path conditions
  - data dependency on branch conditions

Issues with Speculative Load Hardening

Speculative Load Hardening [Carruth, 2018]:
Protect memory access with artificial data dependencies

- Special register accumulates path conditions
  - data dependency on branch conditions
  - detects speculative execution

Issues with Speculative Load Hardening

Speculative Load Hardening [Carruth, 2018]:
Protect memory access with artificial data dependencies

- Special register accumulates path conditions
  - data dependency on branch conditions
  - detects speculative execution
- Memory accesses compute address using special register

Issues with Speculative Load Hardening

Speculative Load Hardening [Carruth, 2018]:
Protect memory access with artificial data dependencies

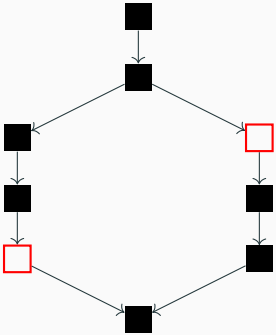
- Special register accumulates path conditions
  - data dependency on branch conditions
  - detects speculative execution
- Memory accesses compute address using special register
  - original address during nonspeculative execution

Issues with Speculative Load Hardening

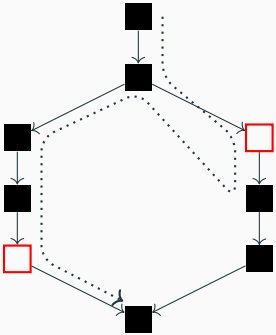
Speculative Load Hardening [Carruth, 2018]:
Protect memory access with artificial data dependencies

- Special register accumulates path conditions
  - data dependency on branch conditions
  - detects speculative execution
- Memory accesses compute address using special register
  - original address during nonspeculative execution
  - known safe address during speculative execution

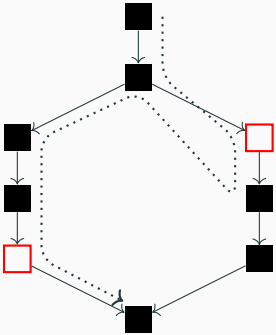
Issues with Speculative Load Hardening

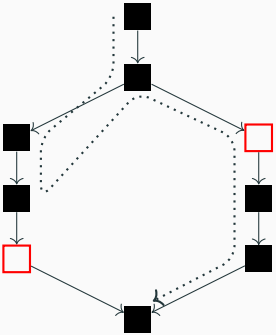


Issues with Speculative Load Hardening



Issues with Speculative Load Hardening





Issues with Speculative Load Hardening

- In practice, data dependency should trigger rollback

Issues with Speculative Load Hardening

- In practice, data dependency should trigger rollback
- Always-Mispredict semantics do not model this

Issues with Speculative Load Hardening

- In practice, data dependency should trigger rollback
- Always-Mispredict semantics do not model this
- More accurate semantics *with vendor guarantees* needed

Relaxed Insertion of Fences

$$\texttt{read}\ [1]\ x;\ \texttt{if}\ x \geq 0\ \texttt{then}\ \texttt{read}\ [x]\ y\ \texttt{else}\ \texttt{read}\ [x]\ y\ \texttt{end};\ \texttt{write}\ [1]\ y$$

Relaxed Insertion of Fences

$$\texttt{read}\ [1]\ x;\ \texttt{if}\ x \geq 0\ \texttt{then}\ \texttt{read}\ [x]\ y\ \texttt{else}\ \texttt{read}\ [x]\ y\ \texttt{end};\ \texttt{write}\ [1]\ y$$

Relaxed insertion of fences:

Relaxed Insertion of Fences

$$\texttt{read}\ [1]\ x;\ \texttt{if}\ x \geq 0\ \texttt{then}\ \texttt{read}\ [x]\ y\ \texttt{else}\ \texttt{read}\ [x]\ y\ \texttt{end};\ \texttt{write}\ [1]\ y$$

Relaxed insertion of fences:

- At every branch, check the sequence of ***read*** and ***write*** instructions

Relaxed Insertion of Fences

$$\texttt{read}\ [1]\ x;\ \texttt{if}\ x \geq 0\ \texttt{then}\ \texttt{read}\ [x]\ y\ \texttt{else}\ \texttt{read}\ [x]\ y\ \texttt{end};\ \texttt{write}\ [1]\ y$$

Relaxed insertion of fences:

- At every branch, check the sequence of `read` and `write` instructions
- Insert a `fence` instruction if the sequences are different

Relaxed Insertion of Fences

$$\texttt{read}\ [1]\ x;\ \texttt{if}\ x \geq 0\ \texttt{then}\ \texttt{read}\ [x]\ y\ \texttt{else}\ \texttt{read}\ [x]\ y\ \texttt{end};\ \texttt{write}\ [1]\ y$$

Relaxed insertion of fences:

- At every branch, check the sequence of `read` and `write` instructions
- Insert a `fence` instruction if the sequences are different
- Insert a `fence` instruction in case of nested branches

Relaxed Insertion of Fences

$$\texttt{read}\ [1]\ x;\ \texttt{if}\ x \geq 0\ \texttt{then}\ \texttt{read}\ [x]\ y\ \texttt{else}\ \texttt{read}\ [x]\ y\ \texttt{end};\ \texttt{write}\ [1]\ y$$

Relaxed insertion of fences:

- At every branch, check the sequence of `read` and `write` instructions
- Insert a `fence` instruction if the sequences are different
- Insert a `fence` instruction in case of nested branches

→ Proof-of-concept implementation has some additional restrictions

Relaxed Insertion of Fences

- Mitigated code may not be speculatively safe

Relaxed Insertion of Fences

- Mitigated code may not be speculatively safe
  - not verifiable by previous approach

Relaxed Insertion of Fences

- Mitigated code may not be speculatively safe
  - not verifiable by previous approach
- Verified under constant-time leakage model

Relaxed Insertion of Fences

- Mitigated code may not be speculatively safe
  - not verifiable by previous approach
- Verified under constant-time leakage model
- Restrictions can be lifted with future work

Relaxed Insertion of Fences

- Mitigated code may not be speculatively safe
  - not verifiable by previous approach
- Verified under constant-time leakage model
- Restrictions can be lifted with future work

$\Longrightarrow$ reasonable mitigations that were not covered before

Further Work

Patrignani and Guarnieri [2021] support

Further Work

Patrignani and Guarnieri [2021] support

- function calls

Further Work

Patrignani and Guarnieri [2021] support

- function calls
- linking against attacker-defined code

Further Work

Patrignani and Guarnieri [2021] support

- function calls
- linking against attacker-defined code

Other applications:

Further Work

Patrignani and Guarnieri [2021] support

- function calls
- linking against attacker-defined code

Other applications:

- semantics closer to actual hardware

Further Work

Patrignani and Guarnieri [2021] support

- function calls
- linking against attacker-defined code

Other applications:

- semantics closer to actual hardware
- semantics for different spectre variations [Fabian et al., 2022]

Further Work

Patrignani and Guarnieri [2021] support

- function calls
- linking against attacker-defined code

Other applications:

- semantics closer to actual hardware
- semantics for different spectre variations [Fabian et al., 2022]
- more precise semantics based on hardware guarantees

Conclusion

Our approach

Conclusion

Our approach

- confirms prior work [Patrignani and Guarnieri, 2021]

Conclusion

Our approach

- confirms prior work [Patrignani and Guarnieri, 2021]
- enables reasoning about more leakage models

Conclusion

Our approach

- confirms prior work [Patrignani and Guarnieri, 2021]
- enables reasoning about more leakage models
  - control flow does not need to be included

Conclusion

Our approach

- confirms prior work [Patrignani and Guarnieri, 2021]
- enables reasoning about more leakage models
  - control flow does not need to be included
- verifies mitigations that could not be verified before

Conclusion

Our approach

- confirms prior work [Patrignani and Guarnieri, 2021]
- enables reasoning about more leakage models
  - control flow does not need to be included
- verifies mitigations that could not be verified before
  - such mitigations are reasonable, not contrived examples

References i

Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 328–343. IEEE Computer Society, 2018. doi: 10.1109/CSF.2018.00031. URL *https://doi.org/10.1109/CSF.2018.00031*.

Chandler Carruth. Speculative Load Hardening, 2018. URL *https://llvm.org/docs/SpeculativeLoadHardening.html*.

References ii

Xaver Fabian, Marco Guarnieri, and Marco Patrignani. Automatic detection of speculative execution combinations. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 965–978. ACM, 2022. doi: 10.1145/3548606.3560555. URL *https://doi.org/10.1145/3548606.3560555*.

Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: principled detection of speculative information flows. *CoRR*, abs/1812.08639, 2018. URL *http://arxiv.org/abs/1812.08639*.

References iii

Intel. Using Intel compilers to mitigate speculative execution side-channel issues, 2018. URL *https://www.intel.com/content/www/us/en/developer/articles/troubleshooting/using-intel-compilers-to-mitigate-speculative-execution-side-ch html*.

Intel. Intel C++ compiler classic developer guide and reference, 2021. URL *https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/compiler-options/code-generation-options/mconditional-branch-qconditional-branch.html*.

References iv

Marco Patrignani and Marco Guarnieri. Exorcising spectres with secure compilers. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 445–461. ACM, 2021. doi: 10.1145/3460120.3484534. URL *https://doi.org/10.1145/3460120.3484534*.

Julian Rosemann. Private communications, not yet released. All relevant proofs are included in the Coq development, 2023.

Nonspeculative Semantics i

$$\frac{}{\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{skip}; p\rangle \to_{\mathsf{ns}} \langle \mathcal{V} \mid \mathcal{H} \mid p\rangle}\ \textsc{Skip}$$

$$\frac{}{\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{fence}; p\rangle \to_{\mathsf{ns}} \langle \mathcal{V} \mid \mathcal{H} \mid p\rangle}\ \textsc{Fence}$$

$$\frac{[\![e]\!]_{\mathcal{V}} = v}{\langle \mathcal{V} \mid \mathcal{H} \mid x := e; p\rangle \to_{\mathsf{ns}} \langle \mathcal{V}[x \mapsto v] \mid \mathcal{H} \mid p\rangle}\ \textsc{Assign}$$

$$\frac{[\![a]\!]_{\mathcal{V}} = a' \qquad \mathcal{H}(a') = v}{\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{read}\,[a]\,x; p\rangle \to_{\mathsf{ns}} \langle \mathcal{V}[x \mapsto v] \mid \mathcal{H} \mid p\rangle}\ \textsc{Read}$$

$$\frac{[\![a]\!]_{\mathcal{V}} = a' \qquad \mathcal{V}(x) = v}{\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{write}\,[a]\,x; p\rangle \to_{\mathsf{ns}} \langle \mathcal{V} \mid \mathcal{H}[a' \mapsto v] \mid p\rangle}\ \textsc{Write}$$

Nonspeculative Semantics ii

$$\frac{[\![b]\!]_{\mathcal{V}} \neq 0}{\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end}; p\rangle \to_{\mathrm{ns}} \langle \mathcal{V} \mid \mathcal{H} \mid p_1 +\!\!+ p\rangle}\ \textsc{IfTrue}$$

$$\frac{[\![b]\!]_{\mathcal{V}} = 0}{\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end}; p\rangle \to_{\mathrm{ns}} \langle \mathcal{V} \mid \mathcal{H} \mid p_2 +\!\!+ p\rangle}\ \textsc{IfFalse}$$

$$\frac{}{\begin{array}{c}\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{while}\ b\ \texttt{do}\ p_1\ \texttt{finally}\ p_2\ \texttt{end}; p\rangle \\ \to_{\mathrm{ns}} \langle \mathcal{V} \mid \mathcal{H} \mid \texttt{if}\ b\ \texttt{then}\ (p_1 +\!\!+ \texttt{while}\ b\ \texttt{do}\ p_1\ \texttt{finally}\ p_2\ \texttt{end})\ \texttt{else}\ p_2\ \texttt{end}; p\rangle\end{array}}\ \textsc{While}$$

$$\frac{}{\langle \mathcal{V} \mid \mathcal{H} \mid [\,]\rangle \to_{\mathrm{ns}} \langle \mathcal{V} \mid \mathcal{H} \mid [\,]\rangle}\ \textsc{Term}$$

Speculative Semantics i

$$\text{decr}(n) := \begin{cases} \bot & \text{if } n = \bot \\ n - 1 & \text{otherwise} \end{cases} \qquad \text{wndw}(n) := \begin{cases} \omega & \text{if } n = \bot \\ n - 1 & \text{otherwise} \end{cases}$$

$$\text{no-rollback}(n) := n > 0 \vee n = \bot \qquad \text{zero-out}(n) := \begin{cases} \bot & \text{if } n = \bot \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\text{no-rollback}(n)}{\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{skip}; p\rangle :: S \rightarrow_{\text{sp}} \langle \text{decr}(n) \mid \mathcal{V} \mid \mathcal{H} \mid p\rangle :: S}\ \text{AMSkip}$$

Speculative Semantics ii

$$\frac{\text{no-rollback}(n)}{\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{fence}; p\rangle :: S \rightarrow_{\text{sp}} \langle \text{zero-out}(n) \mid \mathcal{V} \mid \mathcal{H} \mid p\rangle :: S} \ \textsc{AMFence}$$

$$\frac{\text{no-rollback}(n) \qquad [\![e]\!]_{\mathcal{V}} = v}{\langle n \mid \mathcal{V} \mid \mathcal{H} \mid x := e; p\rangle :: S \rightarrow_{\text{sp}} \langle \text{decr}(n) \mid \mathcal{V}[x \mapsto v] \mid \mathcal{H} \mid p\rangle :: S} \ \textsc{AMAssign}$$

$$\frac{\text{no-rollback}(n) \qquad [\![a]\!]_{\mathcal{V}} = a' \qquad \mathcal{H}(a') = v}{\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{read}\ [a]\ x; p\rangle :: S \rightarrow_{\text{sp}} \langle \text{decr}(n) \mid \mathcal{V}[x \mapsto v] \mid \mathcal{H} \mid p\rangle :: S} \ \textsc{AMRead}$$

$$\frac{\text{no-rollback}(n) \qquad [\![a]\!]_{\mathcal{V}} = a' \qquad \mathcal{V}(x) = v}{\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{write}\ [a]\ x; p\rangle :: S \rightarrow_{\text{sp}} \langle \text{decr}(n) \mid \mathcal{V} \mid \mathcal{H}[a' \mapsto v] \mid p\rangle :: S} \ \textsc{AMWrite}$$

Speculative Semantics iii

$$\frac{\text{no-rollback}(n)}{\begin{array}{c}\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{while}\ e\ \texttt{do}\ p_1\ \texttt{finally}\ p_2\ \texttt{end};\rangle :: S \\ \rightarrow_{\text{sp}} \langle \text{decr}(n) \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{if}\ e\ \texttt{then}\ (p_1 +\!\!+ [\texttt{while}\ e\ \texttt{do}\ p_1\ \texttt{finally}\ p_2\ \texttt{end}])\ \texttt{else}\ p_2\ \texttt{end}; p\rangle \\ :: S\end{array}}\ \textsc{AMWhile}$$

$$\frac{\text{no-rollback}(n) \qquad [\![b]\!]_{\mathcal{V}} \neq 0}{\begin{array}{c}\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end}; p\rangle :: S \\ \rightarrow_{\text{sp}} \langle \text{wndw}(n) \mid \mathcal{V} \mid \mathcal{H} \mid p_2 +\!\!+ p\rangle :: \langle \text{decr}(n) \mid \mathcal{V} \mid \mathcal{H} \mid p_1 +\!\!+ p\rangle :: S\end{array}}\ \textsc{AMIfTrue}$$

Speculative Semantics iv

$$\frac{\text{no-rollback}(n) \qquad [\![b]\!]_{\mathcal{V}} = 0}{\begin{array}{c}\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end}; p\rangle :: S \\ \rightarrow_{\text{sp}} \langle \text{wndw}(n) \mid \mathcal{V} \mid \mathcal{H} \mid p_1 +\!\!+ p\rangle :: \langle \text{decr}(n) \mid \mathcal{V} \mid \mathcal{H} \mid p_2 +\!\!+ p\rangle :: S\end{array}}\ \textsc{AMIfFalse}$$

$$\frac{}{\langle 0 \mid \mathcal{V} \mid \mathcal{H} \mid p\rangle :: S \rightarrow_{\text{sp}} S}\ \textsc{AMRollback} \qquad \frac{n \neq \bot \wedge n > 0}{\langle n \mid \mathcal{V} \mid \mathcal{H} \mid []\rangle :: S \rightarrow_{\text{sp}} S}\ \textsc{AMRollbackT}$$

$$\frac{}{\langle \bot \mid \mathcal{V} \mid \mathcal{H} \mid []\rangle :: S \rightarrow_{\text{sp}} \langle \bot \mid \mathcal{V} \mid \mathcal{H} \mid []\rangle :: S}\ \textsc{AMTerm} \qquad \frac{}{[] \rightarrow_{\text{sp}} []}\ \textsc{AMTerm'}$$

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

Extend the semantics with taint tracking (Safe / Unsafe)

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

Extend the semantics with taint tracking (Safe / Unsafe)

Speculating? + Secret Data?

Observation Taint

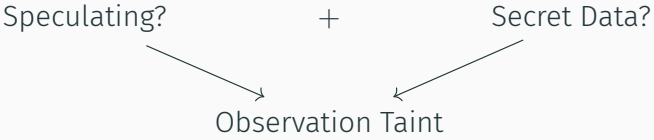
Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

Extend the semantics with taint tracking (Safe / Unsafe)

Speculative safety: All executions of a program only produce safe observations.

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

Extend the semantics with taint tracking (Safe / Unsafe)



Speculative safety: All executions of a program only produce safe observations.

Speculative safety implies speculative noninterference

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

- Taint tracking ignores implicit flows

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

- Taint tracking ignores implicit flows
  - only correct when control flow is observable to the attacker

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

- Taint tracking ignores implicit flows
  - only correct when control flow is observable to the attacker
- Speculatively noninterferent code may not be speculatively safe:

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

- Taint tracking ignores implicit flows
  - only correct when control flow is observable to the attacker
- Speculatively noninterferent code may not be speculatively safe:

$$\texttt{read}\ [1]\ x;\ \texttt{if}\ x \geq 0\ \texttt{then}\ \texttt{read}\ [x]\ y\ \texttt{else}\ \texttt{read}\ [x]\ y\ \texttt{end};\ \texttt{write}\ [1]\ y$$

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

- Taint tracking ignores implicit flows
  - only correct when control flow is observable to the attacker
- Speculatively noninterferent code may not be speculatively safe:

$$\texttt{read}\ [1]\ x;\ \texttt{if}\ x \geq 0\ \texttt{then}\ \texttt{read}\ [x]\ y\ \texttt{else}\ \texttt{read}\ [x]\ y\ \texttt{end};\ \texttt{write}\ [1]\ y$$

  - Speculative reads may be tainted unsafe

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

- Taint tracking ignores implicit flows
  - only correct when control flow is observable to the attacker
- Speculatively noninterferent code may not be speculatively safe:

$$\texttt{read}\ [1]\ x;\ \texttt{if}\ x \geq 0\ \texttt{then}\ \texttt{read}\ [x]\ y\ \texttt{else}\ \texttt{read}\ [x]\ y\ \texttt{end};\ \texttt{write}\ [1]\ y$$

  - Speculative reads may be tainted unsafe
  - Speculatively noninterferent, as all the same read will be performed nonspeculatively

Patrignani and Guarnieri [2021]: Verifying Mitigations Using Speculative Safety

- Taint tracking ignores implicit flows
  - only correct when control flow is observable to the attacker
- Speculatively noninterferent code may not be speculatively safe:

$$\texttt{read}\ [1]\ x;\ \texttt{if}\ x \geq 0\ \texttt{then read}\ [x]\ y\ \texttt{else read}\ [x]\ y\ \texttt{end};\ \texttt{write}\ [1]\ y$$

  - Speculative reads may be tainted unsafe
  - Speculatively noninterferent, as all the same read will be performed nonspeculatively
- Our approach does not have these limitations

Relaxed Insertion of Fences

Additional simplifying restrictions:

Relaxed Insertion of Fences

Additional simplifying restrictions:

- Assignments within the speculation window

Relaxed Insertion of Fences

Additional simplifying restrictions:

- Assignments within the speculation window
- Manually inserted `fence` instructions within the speculation window

Relaxed Insertion of Fences

Additional simplifying restrictions:

- Assignments within the speculation window
- Manually inserted `fence` instructions within the speculation window
- Branches shorter than the speculation window