


**Proceedings Paper:**
Dongol, B., Griffin, M., Popescu, A. orcid.org/0000-0001-8747-0619 et al. (1 more author) (2024) Relative security: formally modeling and (dis)proving resilience against semantic optimization vulnerabilities. In: 2024 IEEE 37th Computer Security Foundations Symposium (CSF). 2024 IEEE 37th Computer Security Foundations Symposium (CSF), 08-12 Jul 2024, Enschede, Netherlands. Institute of Electrical and Electronics Engineers (IEEE), pp. 403-418. ISBN: 9798350362046. ISSN: 1940-1434. EISSN: 2374-8303.

https://doi.org/10.1109/csf61375.2024.00027

# Relative Security: Formally Modeling and (Dis)Proving Resilience Against Semantic Optimization Vulnerabilities

Brijesh Dongol[†], Matt Griffin[†]
Andrei Popescu[*], Jamie Wright[*]

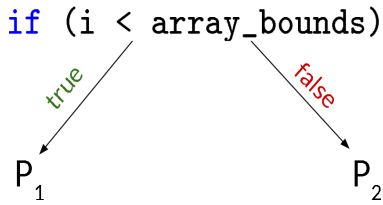
[†] University of Surrey
[*] University of Sheffield

# Speculative Execution

- Memory is much slower than the CPU
- CPU guesses instruction paths to keep busy

# Speculative Execution

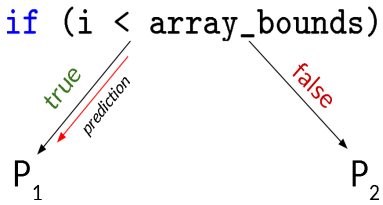
- Memory is much slower than the CPU
- CPU guesses instruction paths to keep busy
- *Branch Prediction*:



# Speculative Execution

- Memory is much slower than the CPU
- CPU guesses instruction paths to keep busy
- *Branch Prediction*:

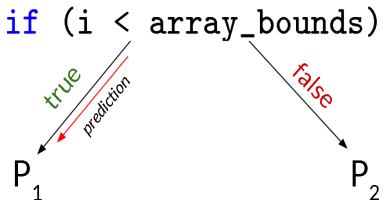
    1) Prediction correct...



# Speculative Execution

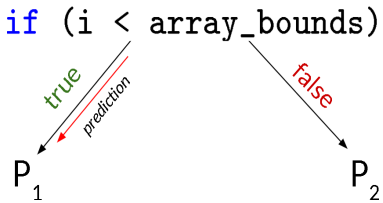
- Memory is much slower than the CPU
- CPU guesses instruction paths to keep busy
- *Branch Prediction*:

    1) Prediction correct... performance boost! 



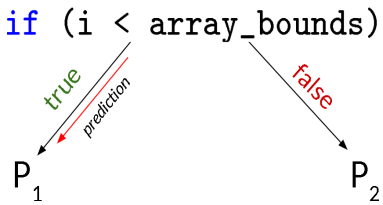
# Speculative Execution

- Memory is much slower than the CPU
- CPU guesses instruction paths to keep busy
- *Branch Prediction*:
  1) Prediction correct... performance boost! 
  2) Prediction incorrect...



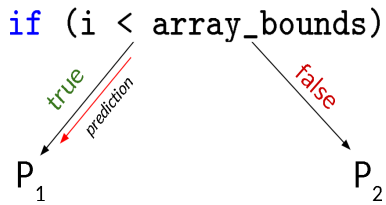
# Speculative Execution

- Memory is much slower than the CPU
- CPU guesses instruction paths to keep busy
- *Branch Prediction*:
  1) Prediction correct... performance boost! 
  2) Prediction incorrect... **Security Problem!!**



# Speculative Execution

- Memory is much slower than the CPU
- CPU guesses instruction paths to keep busy
- *Branch Prediction*:
  1) Prediction correct... performance boost!
  2) Prediction incorrect... **Security Problem!!**

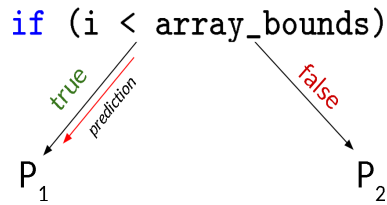



**Spectre**

P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom: *Spectre attacks: Exploiting speculative execution* in S&P. IEEE, 2019

# Speculative Execution

- Memory is much slower than the CPU
- CPU guesses instruction paths to keep busy
- *Branch Prediction*:
  1) Prediction correct... performance boost!
  2) Prediction incorrect... **Security Problem!!**



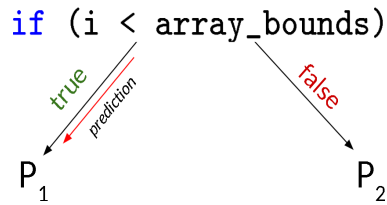

**Spectre**

- What if *i* contains attacker-controlled data?
- Speculative execution cannot be directly observed...

P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom: *Spectre attacks: Exploiting speculative execution* in S&P. IEEE, 2019

# Speculative Execution

- Memory is much slower than the CPU
- CPU guesses instruction paths to keep busy
- *Branch Prediction*:
  1) Prediction correct... performance boost!
  2) Prediction incorrect... **Security Problem!!**




**Spectre**

- What if *i* contains attacker-controlled data?
- Speculative execution cannot be directly observed...
- But side-channels can be exploited
  – Leaks data via CPU cache traces.

P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom: *Spectre attacks: Exploiting speculative execution* in S&P. IEEE, 2019

# History

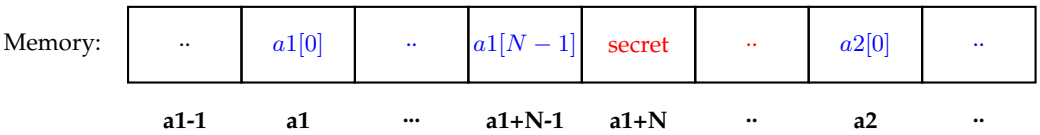
1) Problem uncovered in 2018 affecting *all* major processors (big news)
2) Some variants, e.g., Meltdown / Foreshadow have been fixed via hardware / microcode patches (though older machines are still vulnerable)
3) Spectre believed to be unpatchable; new variants continue to be discovered (Retbleed, NetSpectre, Speculative Store Bypass ...)

There are **63** CVE Records that match your search.

# A Dangerous Program

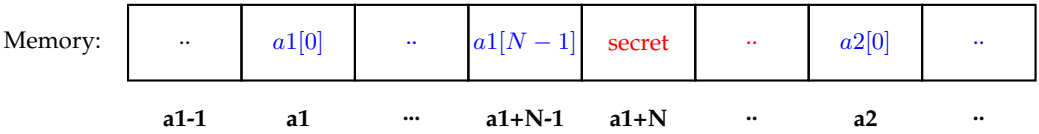
```
1  uint8_t function_v01(unsigned i) {
2     if (i < N) {
3       uint8_t v = a1[i];
4       return a2[v];
5     }
6     return 0;}
```

# A Dangerous Program




```
1  uint8_t function_v01(unsigned i) {
2     if (i < N) {
3       uint8_t v = a1[i];
4       return a2[v];
5     }
6     return 0;}
```

# A Dangerous Program




```
1  uint8_t function_v01(unsigned i) {
2    if (i < N) {
3      uint8_t v = a1[i];
4      return a2[v];
5    }
6    return 0;}
```

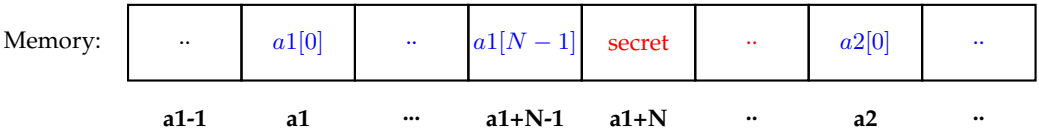

**Attacker:**

– Controls input

– Mistrains predictor

– Observes cache addresses

# A Dangerous Program




**Input: N**

```
1  uint8_t function_v01(unsigned i) {
2    if (i < N) {
3      uint8_t v = a1[i];
4      return a2[v];
5    }
6    return 0;}
```

Cache:

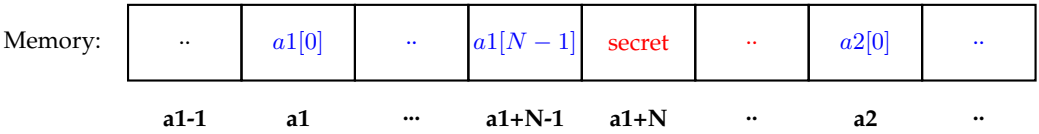
**Attacker:**

– Controls input - $i = N$

– Mistrains predictor

– Observes cache addresses

# A Dangerous Program




mis-predict →

```
1  uint8_t function_v01(unsigned i) {
2    if (i < N) {
3      uint8_t v = a1[i];
4      return a2[v];
5    }
6    return 0;}
```

Cache:

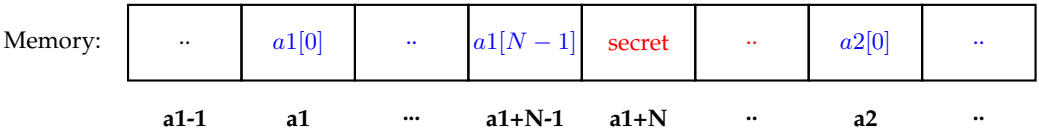
**Attacker:**

– Controls input - $i = N$

– Mistrains predictor

– Observes cache addresses

# A Dangerous Program




speculation

```
1  uint8_t function_v01(unsigned i) {
2    if (i < N) {
3      uint8_t v = a1[i];
4      return a2[v];
5    }
6    return 0;}
```

Cache:

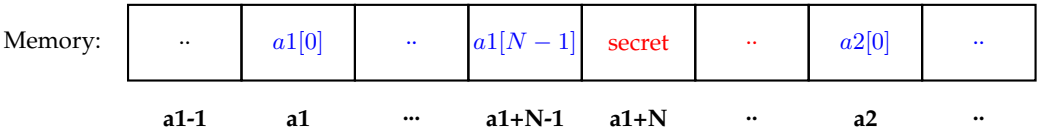

**Attacker:**

– Controls input - $i = N$

– Mistrains predictor

– Observes cache addresses

# A Dangerous Program




speculation →

```
1  uint8_t function_v01(unsigned i) {
2    if (i < N) {
3      uint8_t v = a1[i];
4      return a2[v];
5    }
6    return 0;}
```

Cache:

| a1+N | a2+secret | |
|---|---|---|

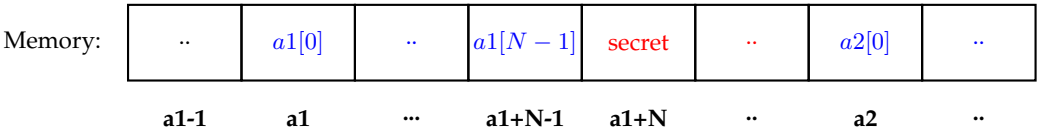
**Attacker:**

– Controls input - $i = N$

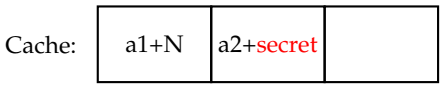
– Mistrains predictor

– Observes cache addresses

# A Dangerous Program




```
1  uint8_t function_v01(unsigned i) {
2     if (i < N) {
3       uint8_t v = a1[i];
4       return a2[v];
5     }
6     return 0;}
```



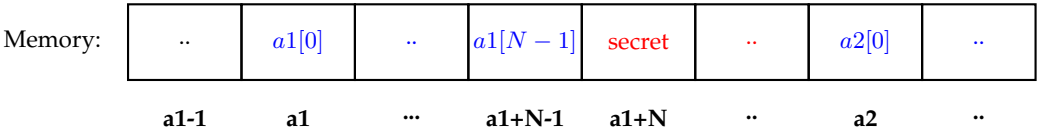

**Attacker:**

– Controls input - $i = N$

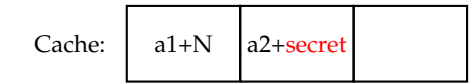
– Mistrains predictor

– Observes cache addresses

# A Dangerous Program




```
1  uint8_t function_v01(unsigned i) {
2     if (i < N) {
3       uint8_t v = a1[i];
4       return a2[v];
5     }
6     return 0;}
```




**SECRET IS REVEALED!**
*via probing cache address*

**Attacker:**

– Controls input - $i = N$

– Mistrains predictor

– Observes cache addresses

# Spectre Mitigation via Fences

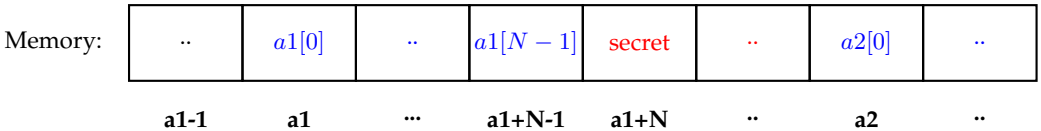
```
1  uint8_t v01_secure(unsigned i) {
2     if (i < N) {
3       _mm_lfence();//resolve spec
4       uint8_t v = a1[i];
5       return a2[v];
6     }
7     return 0;}
```

# Spectre Mitigation via Fences

```
1  uint8_t v01_secure(unsigned i) {
2     if (i < N) {
3       _mm_lfence();//resolve spec
4       uint8_t v = a1[i];
5       return a2[v];
6     }
7     return 0;}
```

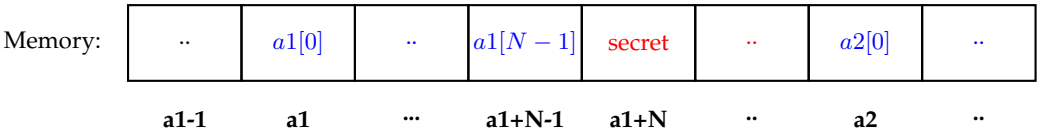
```
1  uint8_t v01_secure_2(unsigned i) {
2     if (i < N) {
3       uint8_t v = a1[i];
4       _mm_lfence();//resolve spec
5       return a2[v];
6     }
7     return 0;}
```

# Spectre Mitigation via Fences




```
1  uint8_t v01_secure_2(unsigned i) {
2     if (i < N) {
3       uint8_t v = a1[i];
4       _mm_lfence();//resolve spec
5       return a2[v];
6     }
7     return 0;}
```

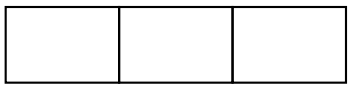
# Spectre Mitigation via Fences



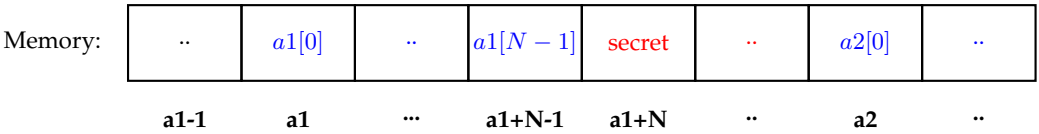

**Input: N**

```
1  uint8_t v01_secure_2(unsigned i) {
2    if (i < N) {
3      uint8_t v = a1[i];
4      _mm_lfence();//resolve spec
5      return a2[v];
6    }
7    return 0;}
```

Cache:



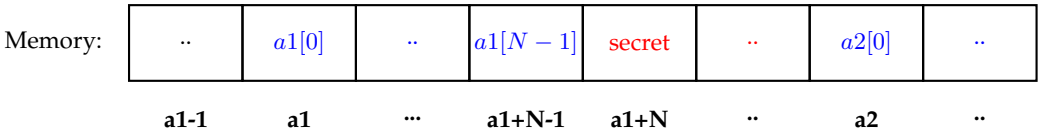
# Spectre Mitigation via Fences




mis-predict →

```
1  uint8_t v01_secure_2(unsigned i) {
2    if (i < N) {
3      uint8_t v = a1[i];
4      _mm_lfence();//resolve spec
5      return a2[v];
6    }
7    return 0;}
```

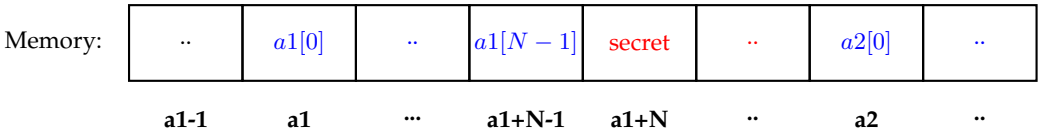
# Spectre Mitigation via Fences




speculation →

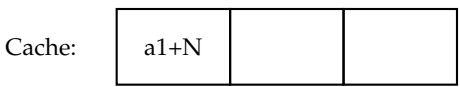
```
1  uint8_t v01_secure_2(unsigned i) {
2    if (i < N) {
3      uint8_t v = a1[i];
4      _mm_lfence();//resolve spec
5      return a2[v];
6    }
7    return 0;}
```

# Spectre Mitigation via Fences



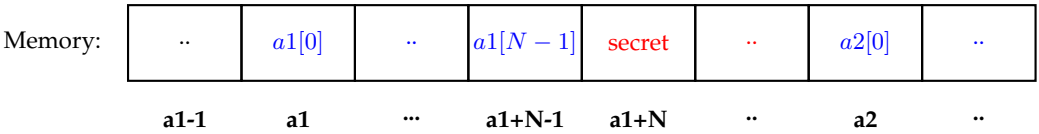

```
1  uint8_t v01_secure_2(unsigned i) {
2     if (i < N) {
3       uint8_t v = a1[i];
4       _mm_lfence();//resolve spec
5       return a2[v];
6     }
7     return 0;}
```

speculation →



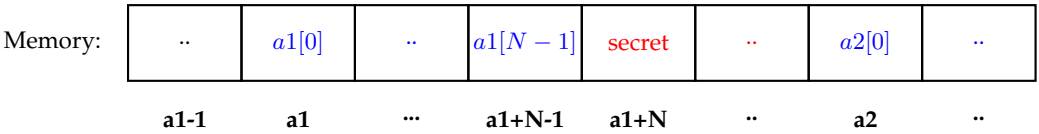

**FENCE HAS BEEN HIT!**

# Spectre Mitigation via Fences

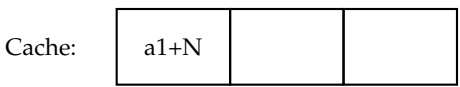



```
1  uint8_t v01_secure_2(unsigned i) {
2     if (i < N) {
3       uint8_t v = a1[i];
4       _mm_lfence();//resolve spec
5       return a2[v];
6     }
7     return 0;}
```

# Spectre Mitigation via Fences




```
1  uint8_t v01_secure_2(unsigned i) {
2     if (i < N) {
3       uint8_t v = a1[i];
4       _mm_lfence();//resolve spec
5       return a2[v];
6     }
7     return 0;}
```



# Spectre Mitigation via Fences

```
1  uint8_t v01_secure_2(unsigned i) {
2     if (i < N) {
3       uint8_t v = a1[i];
4       _mm_lfence();//resolve spec
5       return a2[v];
6     }
7     return 0;}
```

*How can we:*

*(a) characterise Spectre vulnerabilities and*

# Spectre Mitigation via Fences

```
1  uint8_t v01_secure_2(unsigned i) {
2     if (i < N) {
3       uint8_t v = a1[i];
4       _mm_lfence();//resolve spec
5       return a2[v];
6     }
7     return 0;}
```

*How can we:*

*(a) characterise Spectre vulnerabilities and*

*(b) prove their absence?*

# Related (and Inspiring) Work

| Tool | Interactive Attackers | Interactive Secret Uploading |
|---|---|---|
| Conditional NI[1] | No | Restricted To Initial State |
| Speculative NI/Spectector[2] | No | Restricted To Initial State |
| TPOD[3] | Yes | Yes |


[1]: Roberto Guanciale, Musard Balliu, and Mads Dam. Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In CCS, 2020.

[2]: Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In S&P, 2020.

[3]: Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In CSF, 2019.

# Related (and Inspiring) Work

| Tool | Interactive Attackers | Interactive Secret Uploading |
|---|---|---|
| Conditional NI[1] | No | Restricted To Initial State |
| Speculative NI/Spectector[2] | No | Restricted To Initial State |
| TPOD[3] | Yes | Yes |


[1]: Roberto Guanciale, Musard Balliu, and Mads Dam. Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In CCS, 2020.

[2]: Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In S&P, 2020.

[3]: Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In CSF, 2019.


**A comprehensive survey of the state-of-the-art:**
S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, D. Stefan: *SoK: Practical Foundations for Software Spectre Defenses*. IEEE S&P 2022.

# Our Contributions

**Relative Security**

General notion of information-flow security

- captures Spectre-like vulnerabilities
- works generally for transition systems (I/O automata)
- accounts for interactive attackers and interactive uploading of secrets

# Our Contributions

**Relative Security**

General notion of information-flow security

- captures Spectre-like vulnerabilities
- works generally for transition systems (I/O automata)
- accounts for interactive attackers and interactive uploading of secrets

**Unwinding Proof Methodology**

General *unwinding-style* (dis)proof methods for Relative Security

# Our Contributions

**Relative Security**

General notion of information-flow security

- captures Spectre-like vulnerabilities
- works generally for transition systems (I/O automata)
- accounts for interactive attackers and interactive uploading of secrets

**Unwinding Proof Methodology**

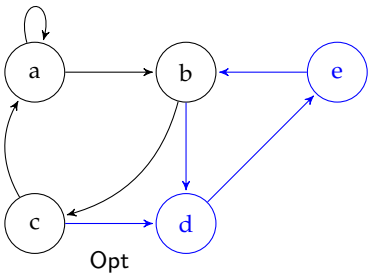
General *unwinding-style* (dis)proof methods for Relative Security

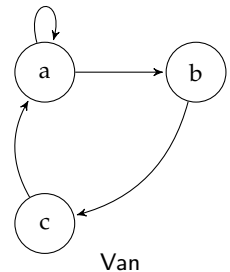
**Verified Examples**

- Instantiation to a C-like language with speculative semantics
- Case studies from the Spectre benchmark verified
- An Isabelle/HOL mechanization of the general framework and the case studies

# Relative Security

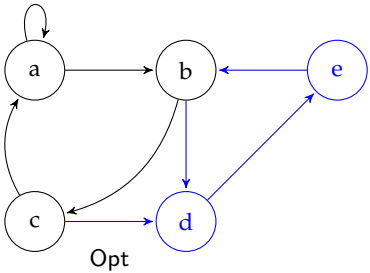
- We wish to compare two semantics of the same program $p$:
  A "vanilla" semantics and an "optimized" semantics (e.g. speculation)

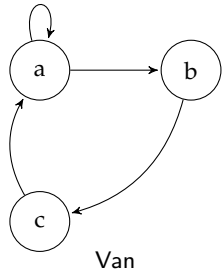


# Relative Security

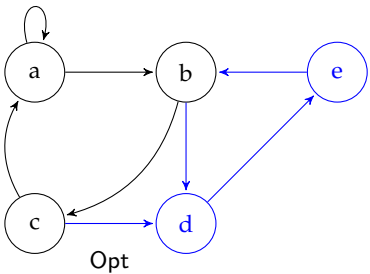
- We wish to compare two semantics of the same program $p$:
  A "vanilla" semantics and an "optimized" semantics (e.g. speculation)
- When no new leaks can be produced by $p$ through the optimization Opt

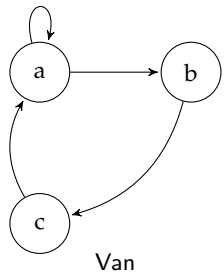


# Relative Security

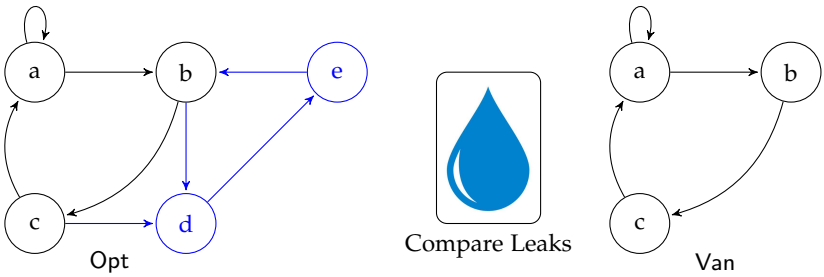
- We wish to compare two semantics of the same program $p$:
  A "vanilla" semantics and an "optimized" semantics (e.g. speculation)
- When no new leaks can be produced by $p$ through the optimization Opt
  Then $p$ satisfies *relative security* wrt Opt



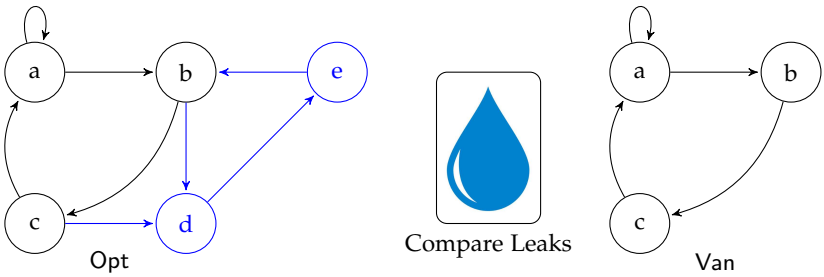
# Relative Security

- We wish to compare two semantics of the same program $p$:
  A "vanilla" semantics and an "optimized" semantics (e.g. speculation)
- When no new leaks can be produced by $p$ through the optimization Opt
  Then $p$ satisfies *relative security* wrt Opt



# Relative Security

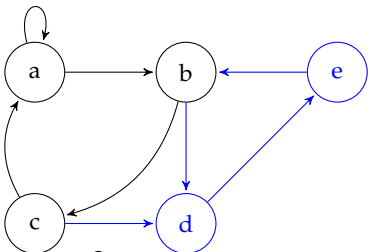
- We wish to compare two semantics of the same program $p$:
  A "vanilla" semantics and an "optimized" semantics (e.g. speculation)
- When no new leaks can be produced by $p$ through the optimization Opt
  Then $p$ satisfies *relative security* wrt Opt
  when
  **For all** possible leaks the semantics with Opt can produce



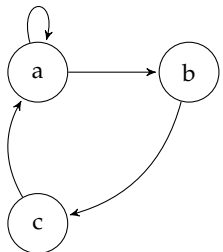
# Relative Security

- We wish to compare two semantics of the same program $p$:
  A "vanilla" semantics and an "optimized" semantics (e.g. speculation)
- When no new leaks can be produced by $p$ through the optimization Opt
  Then $p$ satisfies *relative security* wrt Opt
  when
  **For all** possible leaks the semantics with Opt can produce

  **There exist** traces without Opt which produces the same leak







# What is a leak?

# What is a leak?

A leak consists of two traces $\pi_1$ and $\pi_2$, such that an attacker can:

(a) distinguish between two secrets via observations while
(b) taking the same actions

# What is a leak?

A leak consists of two traces $\pi_1$ and $\pi_2$, such that an attacker can:

(a) distinguish between two secrets via observations while
(b) taking the same actions

```
input x
y ::= x + s
output y
```

# What is a leak?

A leak consists of two traces $\pi_1$ and $\pi_2$, such that an attacker can:

(a) distinguish between two secrets via observations while
(b) taking the same actions

$\mathcal{S}(\pi_1) = (s := 5)$
$\mathcal{S}(\pi_2) = (s := 4)$
$\mathcal{A}(\pi_1) = \mathcal{A}(\pi_2) = (x := 0)$

```
input x
y ::= x + s
output y
```

# What is a leak?

A leak consists of two traces $\pi_1$ and $\pi_2$, such that an attacker can:

(a) distinguish between two secrets via observations while
(b) taking the same actions

$\mathcal{S}(\pi_1) = (s := 5)$
$\mathcal{S}(\pi_2) = (s := 4)$
$\mathcal{A}(\pi_1) = \mathcal{A}(\pi_2) = (x := 0)$

```
input  x
y ::= x + s
output  y
```

$\mathcal{O}(\pi_1) \neq \mathcal{O}(\pi_2)$
$5 \neq 4$

# What is a leak?

A leak consists of two traces $\pi_1$ and $\pi_2$, such that an attacker can:

(a) distinguish between two secrets via observations while
(b) taking the same actions

$\mathcal{S}(\pi_1) = (s := 5)$
$\mathcal{S}(\pi_2) = (s := 4)$
$\mathcal{A}(\pi_1) = \mathcal{A}(\pi_2) = (x := 0)$

```
input x
y ::= x + s
output y
```

$\mathcal{O}(\pi_1) \neq \mathcal{O}(\pi_2)$
$5 \neq 4$

Traces $\pi_1$ and $\pi_2$ produce the leak $(sl_1, sl_2)$ iff...

$$\mathcal{S}(\pi_1) = sl_1 \ \wedge \ \mathcal{S}(\pi_2) = sl_2 \ \ \wedge \ \ \mathcal{A}(\pi_1) = \mathcal{A}(\pi_2) \ \wedge \ \mathcal{O}(\pi_1) \neq \mathcal{O}(\pi_2)$$

# What is a leak?

A leak consists of two traces $\pi_1$ and $\pi_2$, such that an attacker can:

(a) distinguish between two secrets via observations while
(b) taking the same actions

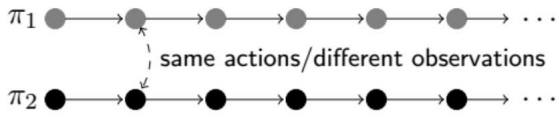
$\mathcal{S}(\pi_1) = (s := 5)$
$\mathcal{S}(\pi_2) = (s := 4)$
$\mathcal{A}(\pi_1) = \mathcal{A}(\pi_2) = (x := 0)$

input x
y ::= x + s
output y

$\mathcal{O}(\pi_1) \neq \mathcal{O}(\pi_2)$
$5 \neq 4$

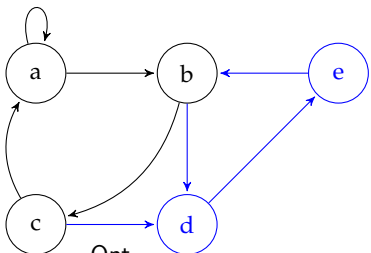
Traces $\pi_1$ and $\pi_2$ produce the leak $(sl_1, sl_2)$ iff...

$$\mathcal{S}(\pi_1) = sl_1 \ \wedge \ \mathcal{S}(\pi_2) = sl_2 \ \wedge \ \mathcal{A}(\pi_1) = \mathcal{A}(\pi_2) \ \wedge \ \mathcal{O}(\pi_1) \neq \mathcal{O}(\pi_2)$$

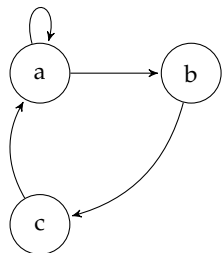


# Relative Security

- Let $\mathsf{Trace}_{\mathbf{van}}, \mathsf{Trace}_{\mathbf{opt}}$ be the set of traces of the vanilla and *optimized* system
- Recall: $\mathcal{S}(\pi_1) = sl_1 \ \wedge \ \mathcal{S}(\pi_2) = sl_2 \ \wedge \ \mathcal{A}(\pi_1) = \mathcal{A}(\pi_2) \ \wedge \ \mathcal{O}(\pi_1) \neq \mathcal{O}(\pi_2)$







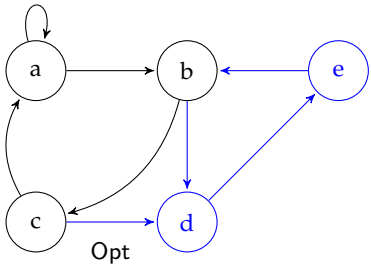
# Relative Security

- Let $\mathsf{Trace}_{\mathbf{van}}$, $\mathsf{Trace}_{\mathbf{opt}}$ be the set of traces of the vanilla and *optimized* system
- Recall: $\mathcal{S}(\pi_1) = sl_1 \ \wedge \ \mathcal{S}(\pi_2) = sl_2 \ \wedge \ \mathcal{A}(\pi_1) = \mathcal{A}(\pi_2) \ \wedge \ \mathcal{O}(\pi_1) \neq \mathcal{O}(\pi_2)$

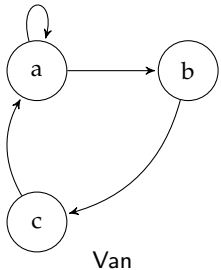
## Definition (Relative Security)

**For all** possible leaks the semantics with Opt can produce

**There exist** traces without Opt which produces the same leak







# Relative Security

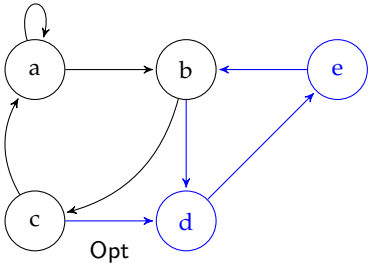
- Let $\mathsf{Trace}_{\mathbf{van}}$, $\mathsf{Trace}_{\mathbf{opt}}$ be the set of traces of the vanilla and *optimized* system
- Recall: $\mathcal{S}(\pi_1) = sl_1 \ \wedge \ \mathcal{S}(\pi_2) = sl_2 \ \wedge \ \mathcal{A}(\pi_1) = \mathcal{A}(\pi_2) \ \wedge \ \mathcal{O}(\pi_1) \neq \mathcal{O}(\pi_2)$

## Definition (Relative Security)

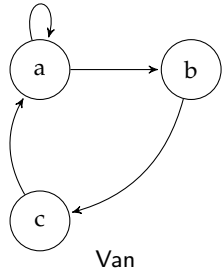
$\forall (sl_1, sl_2) \in \mathsf{Leak}.$

$\forall o_1, o_2 \in \mathsf{Trace}_{\mathbf{opt}}. \ \mathcal{S}(o_1) = sl_1 \ \wedge \ \mathcal{S}(o_2) = sl_2 \ \wedge \ \mathcal{A}(o_1) = \mathcal{A}(o_2) \wedge \ \mathcal{O}(o_1) \neq \mathcal{O}(o_2)$

**There exist** traces without Opt which produces the same leak







# Relative Security

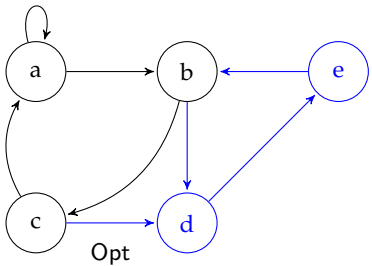
- Let $\mathsf{Trace}_{\mathbf{van}}$, $\mathsf{Trace}_{\mathbf{opt}}$ be the set of traces of the vanilla and *optimized* system
- Recall: $\mathcal{S}(\pi_1) = sl_1 \ \wedge \ \mathcal{S}(\pi_2) = sl_2 \ \wedge \ \mathcal{A}(\pi_1) = \mathcal{A}(\pi_2) \ \wedge \ \mathcal{O}(\pi_1) \neq \mathcal{O}(\pi_2)$

## Definition (Relative Security)

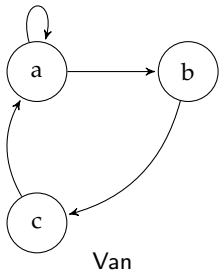
$\forall (sl_1, sl_2) \in \mathsf{Leak}.$

$$\forall o_1, o_2 \in \mathsf{Trace}_{\mathbf{opt}}.\ \mathcal{S}(o_1) = sl_1 \ \wedge \ \mathcal{S}(o_2) = sl_2 \ \wedge \ \mathcal{A}(o_1) = \mathcal{A}(o_2) \ \wedge \ \mathcal{O}(o_1) \neq \mathcal{O}(o_2)$$

$$\exists v_1, v_2 \in \mathsf{Trace}_{\mathbf{van}}.\ \mathcal{S}(v_1) = sl_1 \ \wedge \ \mathcal{S}(v_2) = sl_2 \ \wedge \ \mathcal{A}(v_1) = \mathcal{A}(v_2) \ \wedge \ \mathcal{O}(v_1) \neq \mathcal{O}(v_2)$$







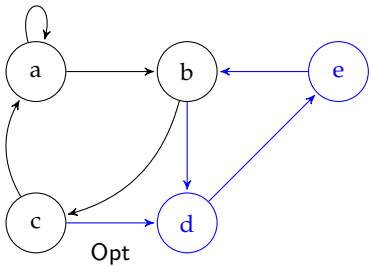
# Relative Security

- Let $\mathsf{Trace}_{\mathbf{van}}$, $\mathsf{Trace}_{\mathbf{opt}}$ be the set of traces of the vanilla and *optimized* system
- Recall: $\mathcal{S}(\pi_1) = sl_1 \ \wedge \ \mathcal{S}(\pi_2) = sl_2 \ \ \wedge \ \ \mathcal{A}(\pi_1) = \mathcal{A}(\pi_2) \ \wedge \ \mathcal{O}(\pi_1) \neq \mathcal{O}(\pi_2)$

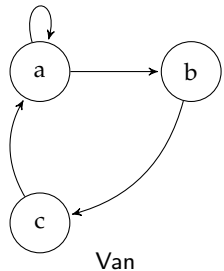
## Definition (Relative Security)

$\forall o_1, o_2 \in \mathsf{Trace}_{\mathbf{opt}}. \ \mathcal{A}(o_1) = \mathcal{A}(o_2) \wedge \ \mathcal{O}(o_1) \neq \mathcal{O}(o_2)$

$\exists v_1, v_2 \in \mathsf{Trace}_{\mathbf{van}}. \ \mathcal{S}(v_1) = \mathcal{S}(o_1) \ \wedge \ \mathcal{S}(v_2) = \mathcal{S}(o_2) \wedge \ \mathcal{A}(v_1) = \mathcal{A}(v_2) \wedge \ \mathcal{O}(v_1) \neq \mathcal{O}(v_2)$





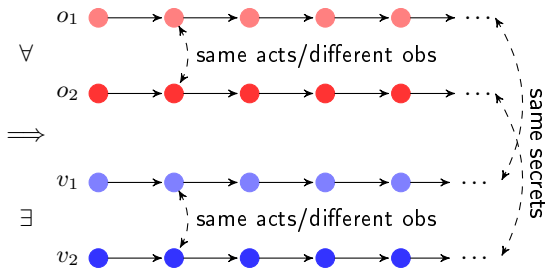


# Relative Security

## Definition (Relative Security)

$\forall o_1, o_2 \in \mathsf{Trace}_{\mathbf{opt}}.\ \mathcal{A}(o_1) = \mathcal{A}(o_2) \wedge\ \mathcal{O}(o_1) \neq \mathcal{O}(o_2)$

$\exists v_1, v_2 \in \mathsf{Trace}_{\mathbf{van}}.\ \mathcal{S}(v_1) = \mathcal{S}(o_1)\ \wedge\ \mathcal{S}(v_2) = \mathcal{S}(o_2) \wedge\ \mathcal{A}(v_1) = \mathcal{A}(v_2) \wedge\ \mathcal{O}(v_1) \neq \mathcal{O}(v_2)$



# Proving Relative Security

Use an *unwinding relation* over the states of the vanilla and optimised systems

Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–87, 1984.

# Proving Relative Security

Use an *unwinding relation* over the states of the vanilla and optimised systems
... A two player game!

Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–87, 1984.

# Proving Relative Security

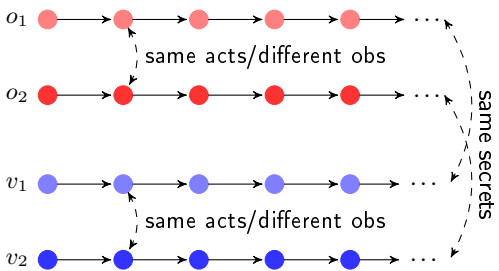
Use an *unwinding relation* over the states of the vanilla and optimised systems
... A two player game!

- Adversary controls optimized traces $o_1$ and $o_2$
- Protagonist build vanilla traces $v_1$ and $v_2$

# Proving Relative Security

Use an *unwinding relation* over the states of the vanilla and optimised systems
... A two player game!

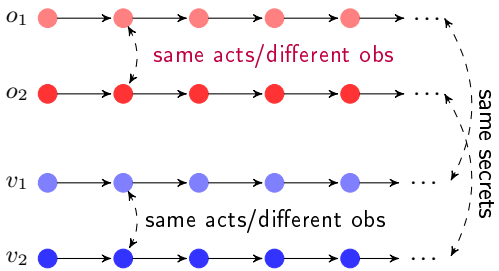
- Adversary controls optimized traces $o_1$ and $o_2$
- Protagonist build vanilla traces $v_1$ and $v_2$



# Proving Relative Security

Use an *unwinding relation* over the states of the vanilla and optimised systems
... A two player game!

- Adversary controls optimized traces $o_1$ and $o_2$
- Protagonist build vanilla traces $v_1$ and $v_2$

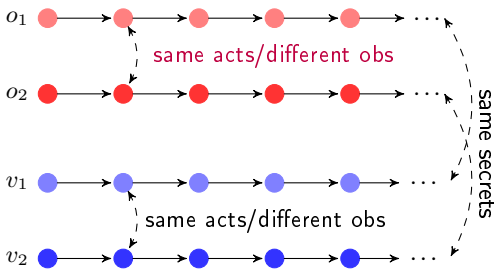



**Assume:** $o_1$ and $o_2$ – same acts/different obs

# Proving Relative Security

Use an *unwinding relation* over the states of the vanilla and optimised systems
... A two player game!

- Adversary controls optimized traces $o_1$ and $o_2$
- Protagonist build vanilla traces $v_1$ and $v_2$




**Assume:** $o_1$ and $o_2$ – same acts/different obs

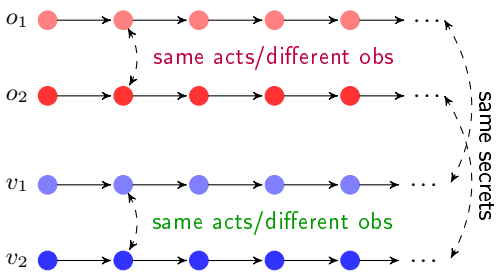
**Contracts:**

# Proving Relative Security

Use an *unwinding relation* over the states of the vanilla and optimised systems
... A two player game!

– Adversary controls optimized traces $o_1$ and $o_2$

– Protagonist build vanilla traces $v_1$ and $v_2$




**Assume:** $o_1$ and $o_2$ – same acts/different obs

**Contracts:**

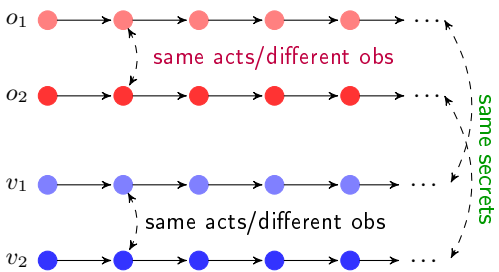
– *Interaction*: $v_1$ and $v_2$ – same acts/different obs

# Proving Relative Security

Use an *unwinding relation* over the states of the vanilla and optimised systems
... A two player game!

– Adversary controls optimized traces $o_1$ and $o_2$

– Protagonist build vanilla traces $v_1$ and $v_2$




**Assume:** $o_1$ and $o_2$ – same acts/different obs

**Contracts:**

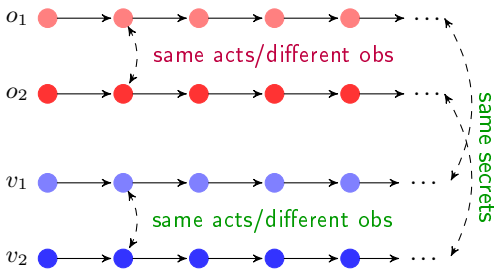
– *Interaction*: $v_1$ and $v_2$ – same acts/different obs

– *Secrecy*: same secrets for $o_1, v_1$ and $o_2, v_2$

# Proving Relative Security

Use an *unwinding relation* over the states of the vanilla and optimised systems
... A two player game!

- Adversary controls optimized traces $o_1$ and $o_2$
- Protagonist build vanilla traces $v_1$ and $v_2$




**Assume:** $o_1$ and $o_2$ – same acts/different obs

**Contracts:**

– *Interaction*: $v_1$ and $v_2$ – same acts/different obs

– *Secrecy*: same secrets for $o_1, v_1$ and $o_2, v_2$

**Assumption + Contracts** $\Longrightarrow$ *Relative Security*

# Proving Relative Security

Use an *unwinding relation* over the states of the vanilla and optimised systems
... A two player game!

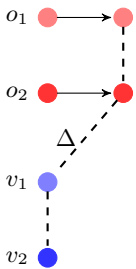
- Adversary controls optimized traces $o_1$ and $o_2$
- Protagonist build vanilla traces $v_1$ and $v_2$


$\Delta \implies$ **Assumption** + **Contracts**

# Proving Relative Security

Use an *unwinding relation* over the states of the vanilla and optimised systems
... A two player game!

- Adversary controls optimized traces $o_1$ and $o_2$
- Protagonist build vanilla traces $v_1$ and $v_2$ **... incrementally**

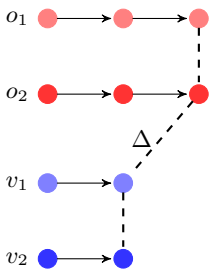



$\Delta \implies$ **Assumption + Contracts**

# Proving Relative Security

Use an *unwinding relation* over the states of the vanilla and optimised systems
... A two player game!

- Adversary controls optimized traces $o_1$ and $o_2$
- Protagonist build vanilla traces $v_1$ and $v_2$ **... incrementally**

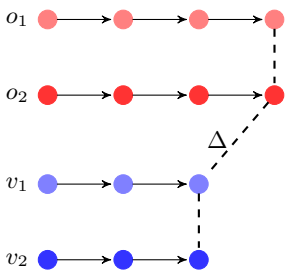



$\Delta \implies$ **Assumption + Contracts**

# Proving Relative Security

Use an *unwinding relation* over the states of the vanilla and optimised systems
... A two player game!

- Adversary controls optimized traces $o_1$ and $o_2$
- Protagonist build vanilla traces $v_1$ and $v_2$ **... incrementally**

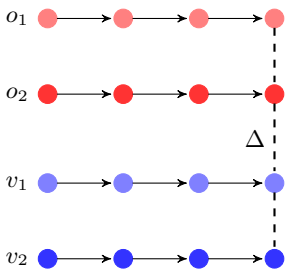



$\Delta \implies$ **Assumption + Contracts**

# Proving Relative Security

Use an *unwinding relation* over the states of the vanilla and optimised systems
... A two player game!

- Adversary controls optimized traces $o_1$ and $o_2$
- Protagonist build vanilla traces $v_1$ and $v_2$ **... incrementally**

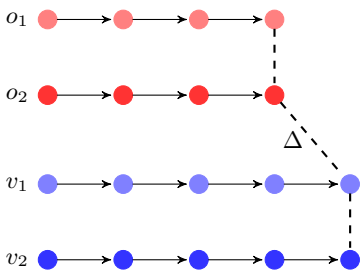



$\Delta \implies$ **Assumption + Contracts**

# Proving Relative Security

Use an *unwinding relation* over the states of the vanilla and optimised systems
... A two player game!

- Adversary controls optimized traces $o_1$ and $o_2$
- Protagonist build vanilla traces $v_1$ and $v_2$ **... incrementally**

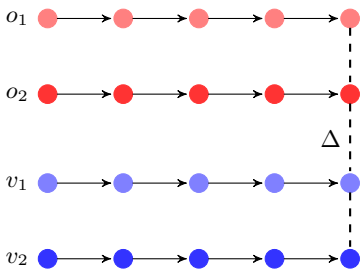



$\Delta \implies$ **Assumption + Contracts**

# Proving Relative Security

Use an *unwinding relation* over the states of the vanilla and optimised systems
... A two player game!

- Adversary controls optimized traces $o_1$ and $o_2$
- Protagonist build vanilla traces $v_1$ and $v_2$ **... incrementally**



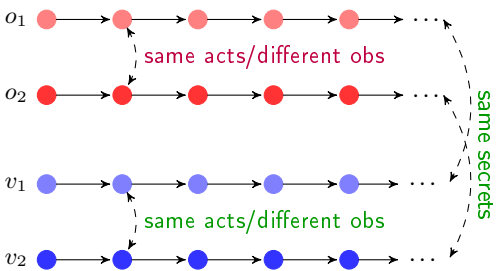

$\Delta \implies$ **Assumption + Contracts**

....

# Proving Relative Security

Use an *unwinding relation* over the states of the vanilla and optimised systems
... A two player game!

- Adversary controls optimized traces $o_1$ and $o_2$
- Protagonist build vanilla traces $v_1$ and $v_2$ **... incrementally**




$\Delta \implies$ **Assumption + Contracts**

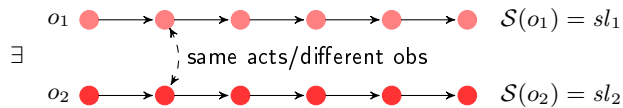
....

**Assumption + Contracts** $\implies$ *Relative Security*

# Disproving Relative Security

**Disproof**

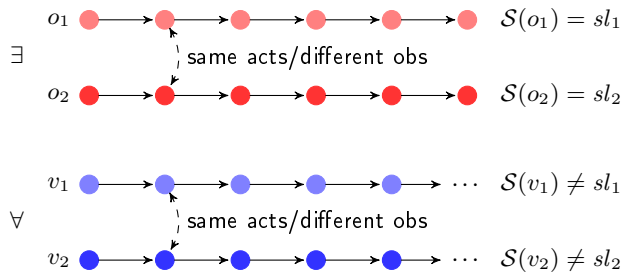
1) Provide traces ($o_1$, $o_2$) producing a concrete leak ($sl_1$, $sl_2$)



# Disproving Relative Security

**Disproof**

1) Provide traces ($o_1$, $o_2$) producing a concrete leak ($sl_1$, $sl_2$)
2) An unwinding, showing that there is no related pair $(v_1, v_2)$ producing the same secrets.



# Relative Security for Speculative Execution

```
1 uint8_t function_v01(unsigned i) {
2   if (i < N) {
3     uint8_t v = a1[i];
4     return a2[v];
5   }
6   return 0;}
```

Relatively Secure? (??)

# Relative Security for Speculative Execution

```
1 uint8_t function_v01(unsigned i) {
2   if (i < N) {
3     uint8_t v = a1[i];
4     return a2[v];
5   }
6   return 0;}
```

Not Relatively Secure (  )

# Relative Security for Speculative Execution

```
1  uint8_t function_v01(unsigned i) {
2     if (i < N) {
3       uint8_t v = a1[i];
4       return a2[v];
5     }
6     return 0;}
```

Not Relatively Secure (👎)

```
1  uint8_t v01_secure(unsigned i) {
2     if (i < N) {
3        _mm_lfence(); //resolve spec
4        uint8_t v = a1[i];
5        return a2[v];
6     }
7     return 0;}
```

Relatively Secure? (??)

# Relative Security for Speculative Execution

```
1  uint8_t function_v01(unsigned i) {
2     if (i < N) {
3       uint8_t v = a1[i];
4       return a2[v];
5     }
6     return 0;}
```

Not Relatively Secure (  )

```
1  uint8_t v01_secure(unsigned i) {
2     if (i < N) {
3        _mm_lfence(); //resolve spec
4        uint8_t v = a1[i];
5        return a2[v];
6     }
7     return 0;}
```

Relatively Secure! (  )

# Relative Security for Speculative Execution

```
1 uint8_t function_v01(unsigned i) {
2    if (i < N) {
3      uint8_t v = a1[i];
4      return a2[v];
5    }
6    return 0;}
```

Not Relatively Secure (  )

```
1 uint8_t v01_secure(unsigned i) {
2    if (i < N) {
3       _mm_lfence(); //resolve spec
4       uint8_t v = a1[i];
5       return a2[v];
6    }
7    return 0;}
```

Relatively Secure! (  )

```
1 uint8_t v01_secure_2(unsigned i) {
2    if (i < N) {
3       uint8_t v = a1[i];
4       _mm_lfence();//resolve spec
5       return a2[v];
6    }
7    return 0;}
```

Relatively secure? (??)

# Relative Security for Speculative Execution

```
1  uint8_t function_v01(unsigned i) {
2     if (i < N) {
3       uint8_t v = a1[i];
4       return a2[v];
5     }
6     return 0;}
```

Not Relatively Secure (  )

```
1  uint8_t v01_secure(unsigned i) {
2     if (i < N) {
3        _mm_lfence(); //resolve spec
4        uint8_t v = a1[i];
5        return a2[v];
6     }
7     return 0;}
```

Relatively Secure! (  )

```
1  uint8_t v01_secure_2(unsigned i) {
2     if (i < N) {
3        uint8_t v = a1[i];
4        _mm_lfence();//resolve spec
5        return a2[v];
6     }
7     return 0;}
```

Relatively Secure! (  )

# Relative Security vs TPOD

```
1  uint8_t cond_secure(unsigned i) {
2    if (i < N) {//Assuming N > 0...
3      uint8_t v = a1[0]; //ind. of i
4      return a2[v];
5    }
6    return 0;}
```

Relatively secure? (??) / TPOD (??)

# Relative Security vs TPOD

```
1  uint8_t cond_secure(unsigned i) {
2    if (i < N) {//Assuming N > 0...
3      uint8_t v = a1[0]; //ind. of i
4      return a2[v];
5    }
6    return 0;}
```

Relatively Secure! (  ) / TPOD (  )

# Conclusion

## Relative Security

New correctness condition **Relative Security**, characterising Spectre-like vulnerabilities

- works generally for any optimization vulnerability
- accounts for interactive attackers and interactive uploading of secrets

## Unwinding Proof Methodology

**Incremental unwinding** proofs to verify presence/absence of vulnerabilities

## Verified Examples

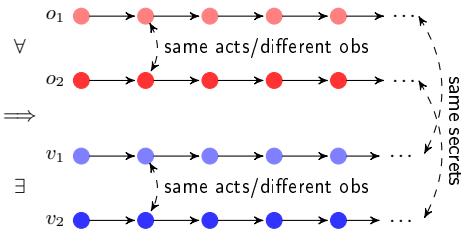
- Instantiation to a C-like language with speculative semantics
- Case studies from the Spectre benchmark verified
- An Isabelle/HOL mechanization of the general framework and the case studies

***Contact me: jwright8@sheffield.ac.uk***

# Appendix

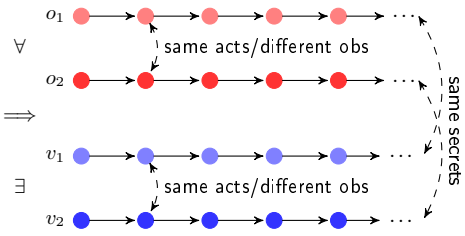
# Relative Security vs TPOD

*Relative Security*

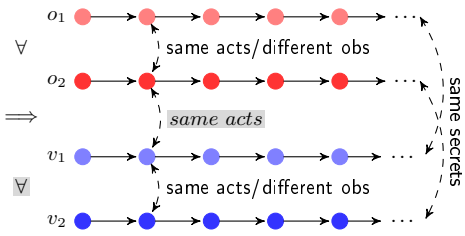


# Relative Security vs TPOD

*Relative Security*



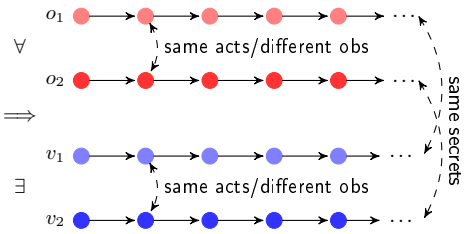

*Trace-property dependent observational nondeterminism (TPOD)*



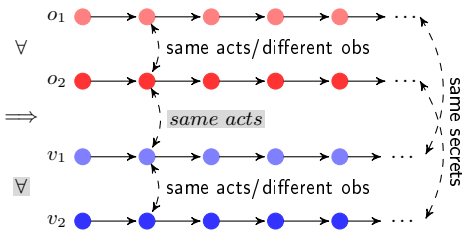

K. Cheang, C. Rasmussen, S. A. Seshia, and P. Subramanyan, "A formal approach to secure speculation," in CSF. IEEE, 2019

# Relative Security vs TPOD

*Relative Security*




*Trace-property dependent observational nondeterminism (TPOD)*




Key differences:

- TPOD requires same actions between optimised *and* vanilla system
- Any leak of $o_1, o_2$ is reproduced not by *some* traces $v_1, v_2$ but all traces that share secrets


K. Cheang, C. Rasmussen, S. A. Seshia, and P. Subramanyan, "A formal approach to secure speculation," in CSF. IEEE, 2019

# Relative Security vs. TPOD

```
1  uint8_t cond_secure(unsigned i) {
2    if (i < N) {//Assuming N > 0...
3      uint8_t v = a1[0]; //ind. of i
4      return a2[v];
5    }
6    return 0;}
```

Relatively secure? (??) / TPOD (??)

# Relative Security vs. TPOD

```
1  uint8_t cond_secure(unsigned i) {
2    if (i < N) {//Assuming N > 0...
3      uint8_t v = a1[0]; //ind. of i
4      return a2[v];
5    }
6    return 0;}
```

Relatively Secure! (  ) / TPOD (  )

# Example proof of security

```
1  uint8_t v01_secure(unsigned i) {
2     if (i < N) {
3       _mm_lfence();//resolve spec
4       uint8_t v = a1[i];
5       return a2[v];
6     }
7     return 0;}
```

$\Delta_0$

| | Spec. | $v_1 = v_2 = o_1^0 = o_2^0$ | $o_1^1 = o_2^1$ | Memory invariants | Read locs. invariants |
|---|---|---|---|---|---|
| $\Delta_0$ | No | 1 | – | $\mathcal{S}(v_1) = \mathcal{S}(o_1) \wedge$ $\mathcal{S}(v_2) = \mathcal{S}(o_2)$ | $v_1 = o_1 \wedge$ $v_2 = o_2$ |

# Example proof of security

```
1  uint8_t v01_secure(unsigned i) {
2     if (i < N) {
3       _mm_lfence();//resolve spec
4       uint8_t v = a1[i];
5       return a2[v];
6     }
7     return 0;}
```

$\Delta_0 \longrightarrow \Delta_1$

| | Spec. | $v_1 = v_2 = o_1^0 = o_2^0$ | $o_1^1 = o_2^1$ | Memory invariants | Read locs. invars ($\mathcal{O}$) |
|---|---|---|---|---|---|
| $\Delta_0$ | No | 1 | – | $\mathcal{S}(v_1) = \mathcal{S}(o_1) \wedge$ $\mathcal{S}(v_2) = \mathcal{S}(o_2)$ | $v_1 = o_1 \wedge$ $v_2 = o_2$ |
| $\Delta_1$ | No | 2–7 | – | $\cdots \wedge$ $v_1 =_i o_1$ | $\cdots$ |

# Example proof of security

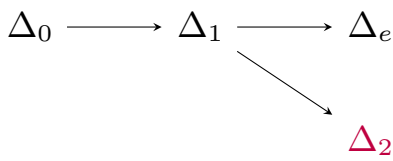
```
1  uint8_t v01_secure(unsigned i) {
2     if (i < N) {
3       _mm_lfence();//resolve spec
4       uint8_t v = a1[i];
5       return a2[v];
6     }
7     return 0;}
```

$$\Delta_0 \longrightarrow \Delta_1 \longrightarrow \Delta_e$$

| | Spec. | $v_1 = v_2 = o_1^0 = o_2^0$ | $o_1^1 = o_2^1$ | Memory invariants | Read locs. invars ($\mathcal{O}$) |
|---|---|---|---|---|---|
| $\Delta_0$ | No | 1 | – | $\mathcal{S}(v_1) = \mathcal{S}(o_1) \wedge$ $\mathcal{S}(v_2) = \mathcal{S}(o_2)$ | $v_1 = o_1 \wedge$ $v_2 = o_2$ |
| $\Delta_1$ | No | 2–7 | – | $\cdots \wedge$ $v_1 =_i o_1$ | $\cdots$ |
| $\Delta_e$ | No | return | – | $\cdots$ | $\cdots$ |

# Example proof of security

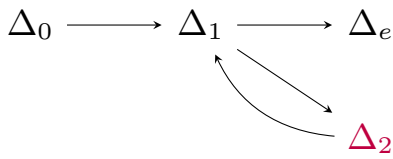
```
1  uint8_t v01_secure(unsigned i) {
2     if (i < N) {
3       _mm_lfence();//resolve spec
4       uint8_t v = a1[i];
5       return a2[v];
6     }
7     return 0;}
```




| | Spec. | $v_1 = v_2 = o_1^0 = o_2^0$ | $o_1^1 = o_2^1$ | Memory invariants | Read locs. invars ($\mathcal{O}$) |
|---|---|---|---|---|---|
| $\Delta_0$ | No | 1 | – | $\mathcal{S}(v_1) = \mathcal{S}(o_1) \wedge$ $\mathcal{S}(v_2) = \mathcal{S}(o_2)$ | $v_1 = o_1 \wedge$ $v_2 = o_2$ |
| $\Delta_1$ | No | 2–7 | – | $\cdots \wedge$ $v_1 =_i o_1$ | $\cdots$ |
| $\Delta_2$ | Yes | 3 | 7 | $\cdots$ | $\cdots$ |
| $\Delta_e$ | No | return | – | $\cdots$ | $\cdots$ |

# Example proof of security

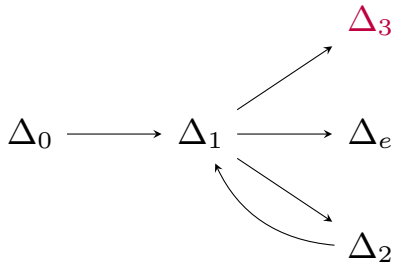
```
1  uint8_t v01_secure(unsigned i) {
2     if (i < N) {
3       _mm_lfence();//resolve spec
4       uint8_t v = a1[i];
5       return a2[v];
6     }
7     return 0;}
```




| | Spec. | $v_1 = v_2 = o_1^0 = o_2^0$ | $o_1^1 = o_2^1$ | Memory invariants | Read locs. invars ($\mathcal{O}$) |
|---|---|---|---|---|---|
| $\Delta_0$ | No | 1 | – | $\mathcal{S}(v_1) = \mathcal{S}(o_1) \wedge$ $\mathcal{S}(v_2) = \mathcal{S}(o_2)$ | $v_1 = o_1 \wedge$ $v_2 = o_2$ |
| $\Delta_1$ | No | 2–7 | – | $\cdots \wedge$ $v_1 =_i o_1$ | $\cdots$ |
| $\Delta_2$ | Yes | 3 | 7 | $\cdots$ | $\cdots$ |
| $\Delta_e$ | No | return | – | $\cdots$ | $\cdots$ |

# Example proof of security

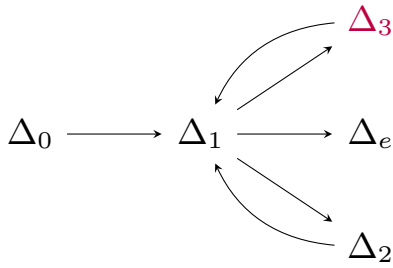
```
1  uint8_t v01_secure(unsigned i) {
2    if (i < N) {
3      _mm_lfence();//resolve spec
4      uint8_t v = a1[i];
5      return a2[v];
6    }
7    return 0;}
```




| | Spec. | $v_1 = v_2 = o_1^0 = o_2^0$ | $o_1^1 = o_2^1$ | Memory invariants | Read locs. invars ($\mathcal{O}$) |
|---|---|---|---|---|---|
| $\Delta_0$ | No | 1 | – | $\mathcal{S}(v_1) = \mathcal{S}(o_1) \wedge$ $\mathcal{S}(v_2) = \mathcal{S}(o_2)$ | $v_1 = o_1 \wedge$ $v_2 = o_2$ |
| $\Delta_1$ | No | 2–7 | – | $\cdots \wedge$ $v_1 =_i o_1$ | $\cdots$ |
| $\Delta_2$ | Yes | 3 | 7 | $\cdots$ | $\cdots$ |
| $\Delta_3$ | Yes | 7 | 3 | $\cdots$ | $\cdots$ |
| $\Delta_e$ | No | return | – | $\cdots$ | $\cdots$ |

# Example proof of security

```
1  uint8_t v01_secure(unsigned i) {
2    if (i < N) {
3      _mm_lfence();//resolve spec
4      uint8_t v = a1[i];
5      return a2[v];
6    }
7    return 0;}
```




| | Spec. | $v_1 = v_2 = o_1^0 = o_2^0$ | $o_1^1 = o_2^1$ | Memory invariants | Read locs. invars ($\mathcal{O}$) |
|---|---|---|---|---|---|
| $\Delta_0$ | No | 1 | – | $\mathcal{S}(v_1) = \mathcal{S}(o_1) \wedge$ $\mathcal{S}(v_2) = \mathcal{S}(o_2)$ | $v_1 = o_1 \wedge$ $v_2 = o_2$ |
| $\Delta_1$ | No | 2–7 | – | $\cdots \wedge$ $v_1 =_i o_1$ | $\cdots$ |
| $\Delta_2$ | Yes | 3 | 7 | $\cdots$ | $\cdots$ |
| $\Delta_3$ | Yes | 7 | 3 | $\cdots$ | $\cdots$ |
| $\Delta_e$ | No | return | – | $\cdots$ | $\cdots$ |

# A simple language with speculative semantics - Syntax

$$\text{Exp} ::= \text{Lit} \mid \text{Var} \mid \text{Exp Op Exp} \mid \ldots$$

$$\text{BExp} ::= \text{true} \mid \text{false} \mid \text{not BExp} \mid \ldots$$

$$\text{Com} ::= \text{Fence} \mid \text{IfJump BExp pc pc} \mid \text{I/O} \ldots$$

# A simple language with speculative semantics - State

**Configuration**:

- Program Counter
- Variable memory
- Array Memory
- Heap
- Pointer

**Predictor**:

- Mispred
- Resolve
- Update

Normal Semantics – (Config, InputBuffer, ReadLocations)

# A simple language with speculative semantics - State

**Configuration**:

- Program Counter
- Variable memory
- Array Memory
- Heap
- Pointer

**Predictor**:

- Mispred
- Resolve
- Update

Normal Semantics – (Config, InputBuffer, ReadLocations)

Speculative Semantics = *Normal Semantics* + **Predictor** + **Speculative Configs**

# A simple language with speculative semantics - Semantics

StartOrFenceOrOutput

$$\frac{c_{pc} \in \{\mathsf{Start}, \mathsf{Fence}\} \cup \{\mathsf{Output}_{och}\ e \mid e \in \mathsf{Exp}\}}{((pc, \mu), inp) \Rightarrow_B ((pc + 1, \mu), inp)}$$

VarAssign

$$\frac{c_{pc} = (x = e)}{((pc, \mu), inp) \Rightarrow_B ((pc + 1, \mu[x \leftarrow [\![e]\!](\mu)]), inp)}$$

AVarAssign

$$\frac{c_{pc} = (a[e] = e')}{((pc, \mu), inp) \Rightarrow_B ((pc + 1, \mu[(a, [\![e]\!](\mu)) \leftarrow [\![e']\!](\mu)]), inp)}$$

Input

$$\frac{c_{pc} = (\mathsf{Input}_{ich}\ x) \qquad inp_{ich} = i \cdot is'}{((pc, \mu), inp) \Rightarrow_B ((pc + 1, \mu[x \leftarrow i]), inp[ich \leftarrow is'])}$$

Jump

$$\frac{c_{pc} = (\mathsf{Jump}\ pc')}{((pc, \mu), inp) \Rightarrow_B ((pc', \mu), inp)}$$

IfJump

$$\frac{c_{pc} = (\mathsf{IfJump}\ b\ pc_1\ pc_2) \qquad pc' = (\text{if } [\![b]\!](\mu) \text{ then } pc_1 \text{ else } pc_2)}{((pc, \mu), inp) \Rightarrow_B ((pc', \mu), inp)}$$

# A simple language with speculative semantics - Extended Semantics

IfJumpMispred

$$\frac{c_{pc} = (\mathsf{IfJump}\ b\ pc_1\ pc_2) \qquad pc' = (\mathsf{if}\ [\![b]\!](\mu)\ \mathsf{then}\ pc_2\ \mathsf{else}\ pc_1)}{((pc, \mu), inp) \Rightarrow_M ((pc', \mu), inp)}$$

Standard

$$\frac{\begin{array}{c}\neg\,\mathsf{isCond}(cfg_k)\ \vee\ \neg\,\mathsf{mispred}(ps, pcs) \\ (k > 0 \longrightarrow \neg\,\mathsf{isIOorFence}(cfg_k)\ \wedge\ \neg\,\mathsf{resolve}(ps, pcs)) \\ (cfg_k, inp) \Rightarrow_B (cfg', inp') \\ C' = cfg_0 \cdot \ldots \cdot cfg_{k-1} \cdot cfg' \qquad L' = L \cup \mathsf{readLocs}(cfg_k)\end{array}}{(ps, cfg_0 \cdot \ldots \cdot cfg_k, inp, L) \Rightarrow_S (ps, C', inp', L')}$$

Mispred

$$\frac{\begin{array}{c}\mathsf{isCond}(cfg_k) \qquad \mathsf{mispred}(ps, pcs) \\ (cfg_k, inp) \Rightarrow_B (cfg', inp') \qquad (cfg_k, inp) \Rightarrow_M (cfg'', inp'') \\ C' = cfg_0 \cdot \ldots \cdot cfg_{k-1} \cdot cfg' \cdot cfg'' \qquad L' = L \cup \mathsf{readLocs}(cfg_k)\end{array}}{(ps, cfg_0 \cdot \ldots \cdot cfg_k, inp, L) \Rightarrow_S (\mathsf{update}(ps, pcs), C', inp', L')}$$

Resolve

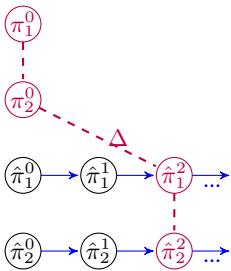
$$\frac{k > 0 \qquad \mathsf{resolve}(ps, pcs) \qquad C' = cfg_0 \cdot \ldots \cdot cfg_{k-1}}{(ps, cfg_0 \cdot \ldots \cdot cfg_k, inp, L) \Rightarrow_S (\mathsf{update}(ps, pcs), C', inp, L)}$$

Fence

$$\frac{k > 0 \qquad \neg\,\mathsf{resolve}(ps, pcs) \qquad \mathsf{isFence}(cfg_k)}{(ps, cfg_0 \cdot \ldots \cdot cfg_k, inp, L) \Rightarrow_S (pcs, cfg_0, inp, L)}$$

# A problem with infinite traces

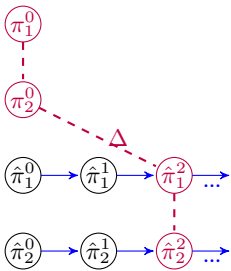
What if the player makes infinite independent steps?




Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–87, 1984.

# A problem with infinite traces

What if the player makes infinite independent steps?




We include a timer in our unwinding which decreases with every proactive step (and resets to $\infty$ when reacting)

Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–87, 1984.