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

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

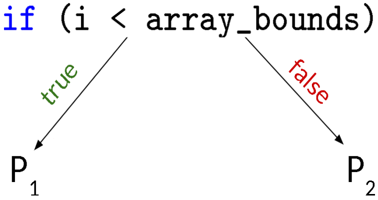
[†] University of Surrey
[*] University of Sheffield

Speculative Execution

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

Speculative Execution

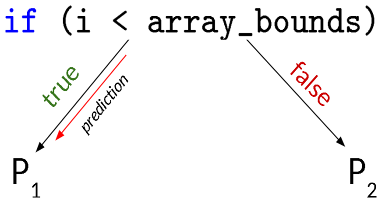
- Memory is much slower than the CPU
- CPU guesses instruction paths to keep busy
- *Branch Prediction*:



Speculative Execution

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

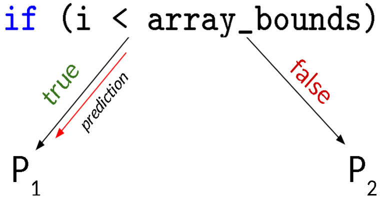
  1) Prediction correct...



Speculative Execution

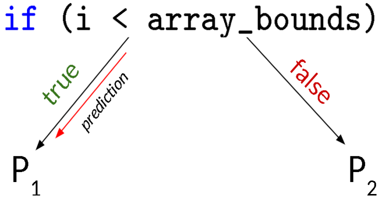
- Memory is much slower than the CPU
- CPU guesses instruction paths to keep busy
- *Branch Prediction*:

  1) Prediction correct... performance boost!



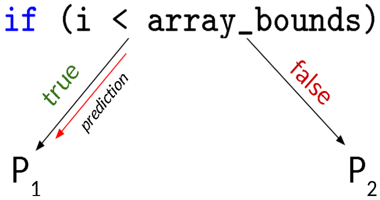
Speculative Execution

- 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

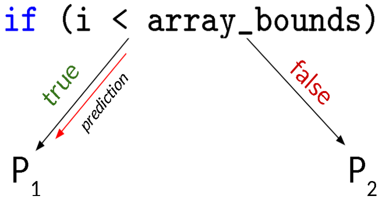
- Memory is much slower than the CPU
- CPU guesses instruction paths to keep busy
- *Branch Prediction*:
    1) Prediction correct... performance boost!
    2) Prediction incorrect... **Security Problem!!**



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!!**

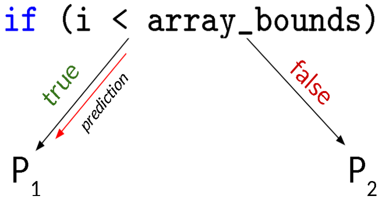


Spectre

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!!**



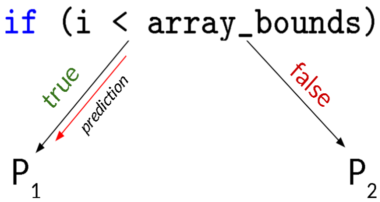
Spectre

- 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

```c
uint8_t function_v01(unsigned i) {
   if (i < N) {
     uint8_t v = a1[i];
     return a2[v];
   }
   return 0;}
```

A Dangerous Program

Memory:	..	$a1[0]$	..	$a1[N-1]$	secret	..	$a2[0]$	..
	a1-1	**a1**	**...**	**a1+N-1**	**a1+N**	**..**	**a2**	**..**

```
uint8_t function_v01(unsigned i) {
   if (i < N) {
     uint8_t v = a1[i];
     return a2[v];
   }
   return 0;}
```

Cache:			

A Dangerous Program

Memory:

..	$a1[0]$	..	$a1[N-1]$	secret	..	$a2[0]$	..
a1-1	**a1**	**...**	**a1+N-1**	**a1+N**	**..**	**a2**	**..**

```
uint8_t function_v01(unsigned i) {
  if (i < N) {
    uint8_t v = a1[i];
    return a2[v];
  }
  return 0;}
```

Cache:

Attacker:

– Controls input

– Mistrains predictor

– Observes cache addresses

A Dangerous Program

Memory:	..	$a1[0]$	..	$a1[N-1]$	secret	..	$a2[0]$	..
	a1-1	**a1**	**...**	**a1+N-1**	**a1+N**	**..**	**a2**	**..**

Input: N →

```c
uint8_t function_v01(unsigned i) {
  if (i < N) {
    uint8_t v = a1[i];
    return a2[v];
  }
  return 0;}
```

Cache:

Attacker:

– Controls input - $i = N$

– Mistrains predictor

– Observes cache addresses

A Dangerous Program

Memory:

..	$a1[0]$	..	$a1[N-1]$	secret	..	$a2[0]$	..
a1-1	**a1**	**...**	**a1+N-1**	**a1+N**	**..**	**a2**	**..**

mis-predict

```
uint8_t function_v01(unsigned i) {
   if (i < N) {
     uint8_t v = a1[i];
     return a2[v];
   }
   return 0;}
```

Cache:

Attacker:

– Controls input - $i = N$

– Mistrains predictor

– Observes cache addresses

A Dangerous Program

Memory:	..	$a1[0]$	..	$a1[N-1]$	secret	..	$a2[0]$	..
	a1-1	**a1**	**...**	**a1+N-1**	**a1+N**	**..**	**a2**	**..**

```c
uint8_t function_v01(unsigned i) {
  if (i < N) {
    uint8_t v = a1[i];
    return a2[v];
  }
  return 0;}
```

speculation

Cache:	a1+N		

Attacker:

– Controls input - $i = N$

– Mistrains predictor

– Observes cache addresses

A Dangerous Program

Memory:	..	$a1[0]$	..	$a1[N-1]$	secret	..	$a2[0]$	..
	a1-1	**a1**	**...**	**a1+N-1**	**a1+N**	**..**	**a2**	**..**

speculation

```
uint8_t function_v01(unsigned i) {
   if (i < N) {
     uint8_t v = a1[i];
     return a2[v];
   }
   return 0;}
```

Cache:	a1+N	a2+secret	

Attacker:

– Controls input - $i = N$

– Mistrains predictor

– Observes cache addresses

A Dangerous Program

Memory:	..	$a1[0]$	..	$a1[N-1]$	secret	..	$a2[0]$	..
	a1-1	**a1**	**...**	**a1+N-1**	**a1+N**	**..**	**a2**	**..**

```
uint8_t function_v01(unsigned i) {
   if (i < N) {
     uint8_t v = a1[i];
     return a2[v];
   }
   return 0;}
```

Cache:	a1+N	a2+secret	

Attacker:

– Controls input - $i = N$

– Mistrains predictor

– Observes cache addresses

A Dangerous Program

Memory:	..	$a1[0]$	..	$a1[N-1]$	secret	..	$a2[0]$	..
	a1-1	**a1**	**...**	**a1+N-1**	**a1+N**	**..**	**a2**	**..**

```
uint8_t function_v01(unsigned i) {
  if (i < N) {
    uint8_t v = a1[i];
    return a2[v];
  }
  return 0;}
```

Cache:	a1+N	a2+secret	

SECRET IS REVEALED!

via probing cache address

Attacker:

– Controls input - $i = N$

– Mistrains predictor

– Observes cache addresses

Spectre Mitigation via Fences

```
uint8_t v01_secure(unsigned i) {
   if (i < N) {
     _mm_lfence();//resolve spec
     uint8_t v = a1[i];
     return a2[v];
   }
   return 0;}
```

Spectre Mitigation via Fences

```c
uint8_t v01_secure(unsigned i) {
  if (i < N) {
    _mm_lfence();//resolve spec
    uint8_t v = a1[i];
    return a2[v];
  }
  return 0;}
```

```c
uint8_t v01_secure_2(unsigned i) {
  if (i < N) {
    uint8_t v = a1[i];
    _mm_lfence();//resolve spec
    return a2[v];
  }
  return 0;}
```

Spectre Mitigation via Fences

Memory:

	··	$a1[0]$	··	$a1[N-1]$	secret	··	$a2[0]$	··
	a1-1	**a1**	**···**	**a1+N-1**	**a1+N**	**··**	**a2**	**··**

```c
uint8_t v01_secure_2(unsigned i) {
   if (i < N) {
     uint8_t v = a1[i];
     _mm_lfence();//resolve spec
     return a2[v];
   }
   return 0;}
```

Cache:

Spectre Mitigation via Fences

Memory:

··	$a1[0]$	··	$a1[N-1]$	secret	··	$a2[0]$	··
a1-1	**a1**	**···**	**a1+N-1**	**a1+N**	**··**	**a2**	**··**

Input: N

```
uint8_t v01_secure_2(unsigned i) {
   if (i < N) {
     uint8_t v = a1[i];
     _mm_lfence();//resolve spec
     return a2[v];
   }
   return 0;}
```

Cache:

Spectre Mitigation via Fences

Memory:

..	$a1[0]$	..	$a1[N-1]$	secret	..	$a2[0]$	..
a1-1	**a1**	**...**	**a1+N-1**	**a1+N**	**..**	**a2**	**..**

mis-predict →

```
uint8_t v01_secure_2(unsigned i) {
  if (i < N) {
    uint8_t v = a1[i];
    _mm_lfence();//resolve spec
    return a2[v];
  }
  return 0;}
```

Cache:

Spectre Mitigation via Fences

Memory:

..	$a1[0]$	..	$a1[N-1]$	secret	..	$a2[0]$	..
a1-1	**a1**	**...**	**a1+N-1**	**a1+N**	**..**	**a2**	**..**

speculation

```
uint8_t v01_secure_2(unsigned i) {
   if (i < N) {
     uint8_t v = a1[i];
     _mm_lfence();//resolve spec
     return a2[v];
   }
   return 0;}
```

Cache:

a1+N		

Spectre Mitigation via Fences

Memory:

..	$a1[0]$	..	$a1[N-1]$	secret	..	$a2[0]$	..
a1-1	**a1**	**...**	**a1+N-1**	**a1+N**	**..**	**a2**	**..**

speculation

```
uint8_t v01_secure_2(unsigned i) {
  if (i < N) {
    uint8_t v = a1[i];
    _mm_lfence();//resolve spec
    return a2[v];
  }
  return 0;}
```

Cache:

a1+N		

FENCE HAS BEEN HIT!

Spectre Mitigation via Fences

Memory:	··	$a1[0]$	··	$a1[N-1]$	secret	··	$a2[0]$	··
	a1-1	**a1**	**···**	**a1+N-1**	**a1+N**	**··**	**a2**	**··**

```
uint8_t v01_secure_2(unsigned i) {
  if (i < N) {
    uint8_t v = a1[i];
    _mm_lfence();//resolve spec
    return a2[v];
  }
  return 0;}
```

Cache:	a1+N		

Spectre Mitigation via Fences

Memory:	··	$a1[0]$	··	$a1[N-1]$	secret	··	$a2[0]$	··
	a1-1	**a1**	**···**	**a1+N-1**	**a1+N**	**··**	**a2**	**··**

```c
uint8_t v01_secure_2(unsigned i) {
  if (i < N) {
    uint8_t v = a1[i];
    _mm_lfence();//resolve spec
    return a2[v];
  }
  return 0;}
```

Cache:	a1+N		

Spectre Mitigation via Fences

```
uint8_t v01_secure_2(unsigned i) {
   if (i < N) {
     uint8_t v = a1[i];
     _mm_lfence();//resolve spec
     return a2[v];
   }
   return 0;}
```

How can we:

(a) characterise Spectre vulnerabilities and

Spectre Mitigation via Fences

```
uint8_t v01_secure_2(unsigned i) {
   if (i < N) {
     uint8_t v = a1[i];
     _mm_lfence();//resolve spec
     return a2[v];
   }
   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

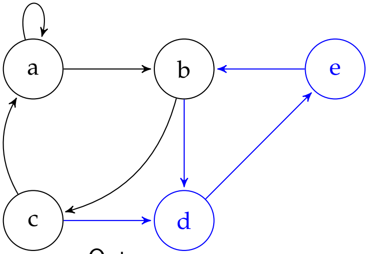
– Instantiation to a C-like language with speculative semantics

– Case studies from the Spectre benchmark verified

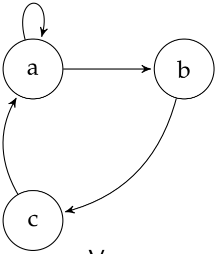
– An Isabelle/HOL mechanization of the general framework and the case studies

Relative Security

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



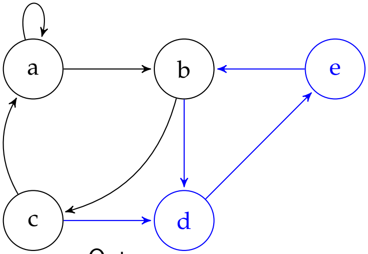
Opt



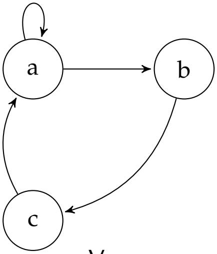
Van

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



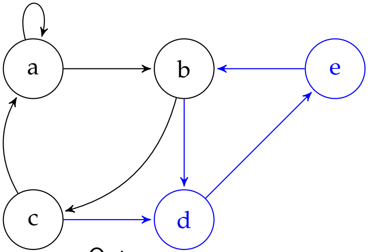
Opt



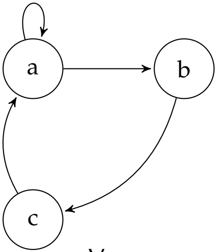
Van

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



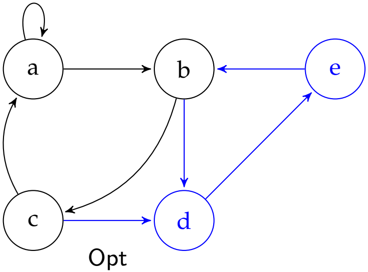
Opt

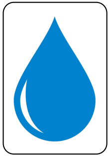


Van

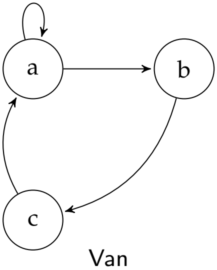
Relative Security

- 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



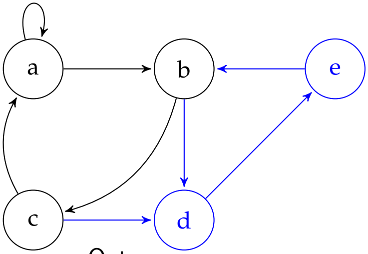


Compare Leaks

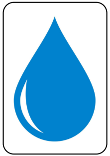


Relative Security

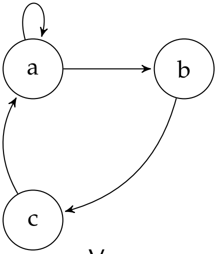
- We wish to compare two semantics of the same program p:
  A "vanilla" semantics and an "optimized" semantics (e.g. speculation)
- When no new leaks can be produced by p through the optimization Opt
  Then p satisfies *relative security* wrt Opt
  when
  For all possible leaks the semantics with Opt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



Opt



Compare Leaks

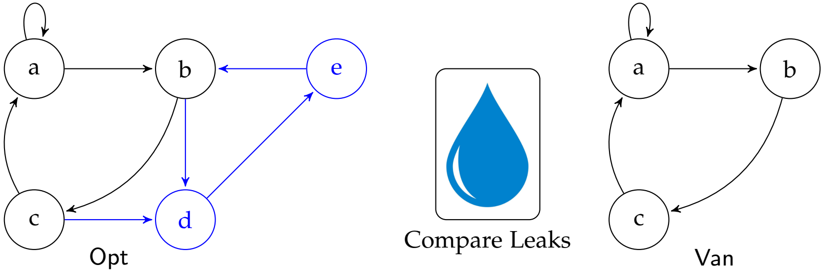


Van

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 π_1 and π_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 π_1 and π_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 π_1 and π_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 π_1 and π_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 π_1 and π_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 π_1 and π_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 π_1 and π_2, such that an attacker can:

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

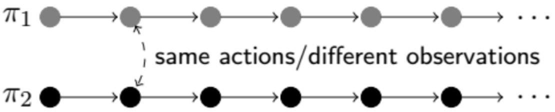
$$\mathcal{S}(\pi_1) = (s := 5)$$
$$\mathcal{S}(\pi_2) = (s := 4)$$
$$\mathcal{A}(\pi_1) = \mathcal{A}(\pi_2) = (x := 0)$$

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

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

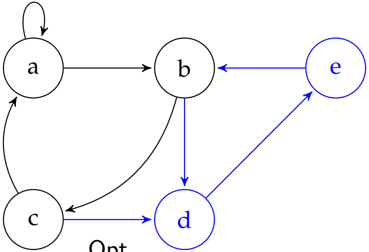
Traces π_1 and π_2 produce the leak (sl_1, sl_2) iff...

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

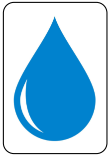


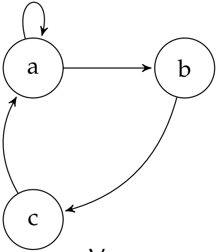
Relative Security

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



Opt


Compare Leaks



Van

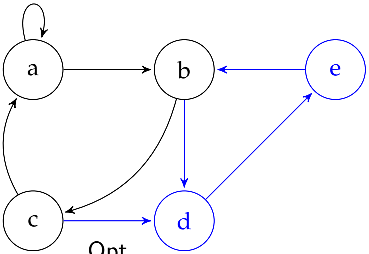
Relative Security

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

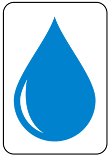
Definition (Relative Security)

For all possible leaks the semantics with Opt can produce

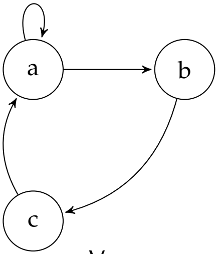
There exist traces without Opt which produces the same leak



Opt



Compare Leaks



Van

Relative Security

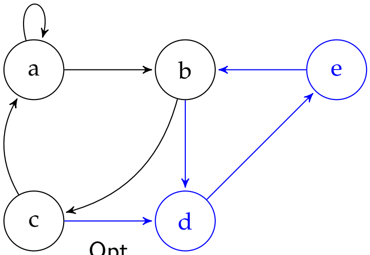
- Let $\mathsf{Trace}_{\mathbf{van}}, \mathsf{Trace}_{\mathbf{opt}}$ be the set of traces of the vanilla and *optimized* system
- Recall: $\mathcal{S}(\pi_1) = sl_1 \;\wedge\; \mathcal{S}(\pi_2) = sl_2 \;\wedge\; \mathcal{A}(\pi_1) = \mathcal{A}(\pi_2) \;\wedge\; \mathcal{O}(\pi_1) \neq \mathcal{O}(\pi_2)$

Definition (Relative Security)

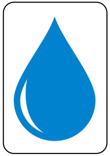
$\forall (sl_1, sl_2) \in \mathsf{Leak}.$

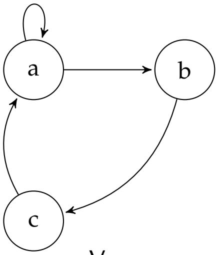
$$\forall o_1, o_2 \in \mathsf{Trace}_{\mathbf{opt}}.\; \mathcal{S}(o_1) = sl_1 \;\wedge\; \mathcal{S}(o_2) = sl_2 \;\wedge\; \mathcal{A}(o_1) = \mathcal{A}(o_2) \wedge \mathcal{O}(o_1) \neq \mathcal{O}(o_2)$$

There exist traces without Opt which produces the same leak



Opt


Compare Leaks



Van

Relative Security

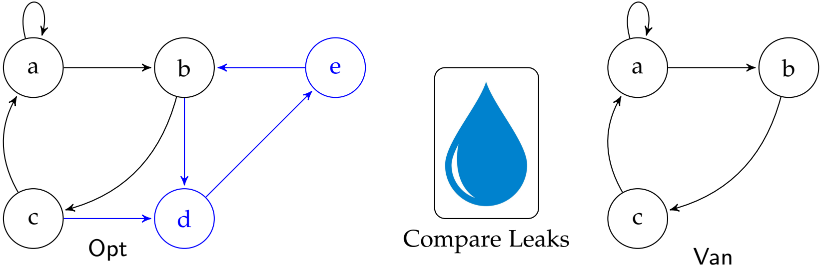
- Let $\textsf{Trace}_{\textbf{van}}, \textsf{Trace}_{\textbf{opt}}$ be the set of traces of the vanilla and *optimized* system
- Recall: $\mathcal{S}(\pi_1) = sl_1 \ \wedge \ \mathcal{S}(\pi_2) = sl_2 \ \ \wedge \ \ \mathcal{A}(\pi_1) = \mathcal{A}(\pi_2) \ \wedge \ \mathcal{O}(\pi_1) \neq \mathcal{O}(\pi_2)$

Definition (Relative Security)

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

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

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



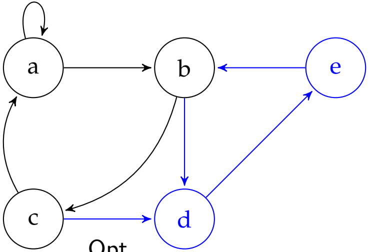
Relative Security

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

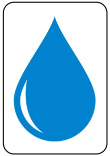
Definition (Relative Security)

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

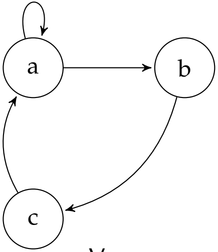
$\exists v_1, v_2 \in \mathsf{Trace}_{\mathbf{van}}.\ \mathcal{S}(v_1) = \mathcal{S}(o_1) \ \wedge \ \mathcal{S}(v_2) = \mathcal{S}(o_2) \wedge \ \mathcal{A}(v_1) = \mathcal{A}(v_2) \wedge \ \mathcal{O}(v_1) \neq \mathcal{O}(v_2)$



Opt



Compare Leaks



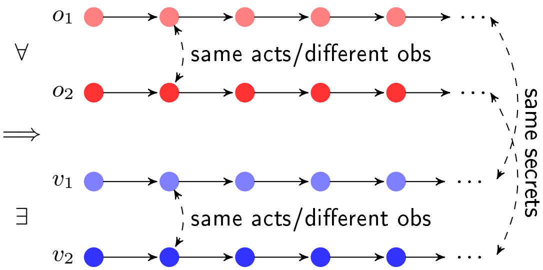
Van

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

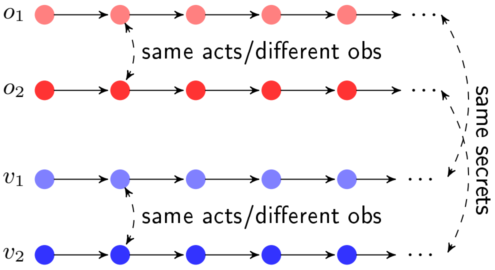
Use an *unwinding relation* over the states of the vanilla and optimised systems
... A two player game!

- 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!

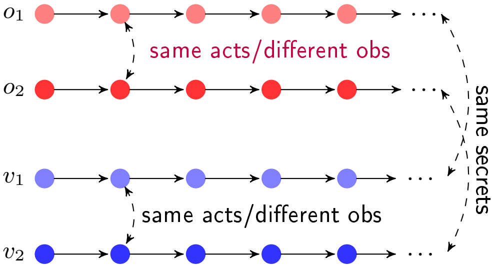
- Adversary controls optimized traces o_1 and o_2
- Protagonist build vanilla traces v_1 and v_2



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

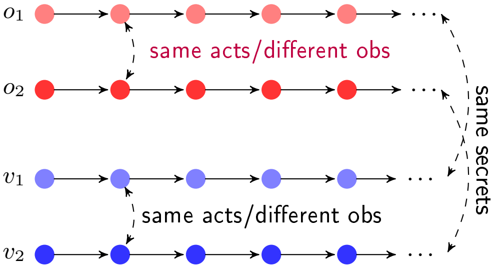


Assume: o_1 and o_2 – same acts/different obs

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



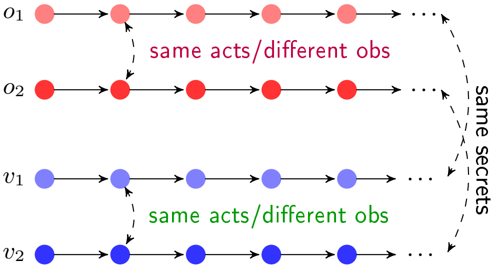
Assume: o_1 and o_2 – same acts/different obs

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

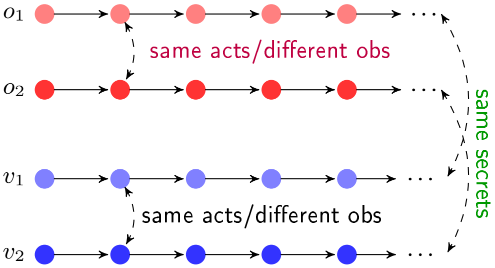
Contracts:

– *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

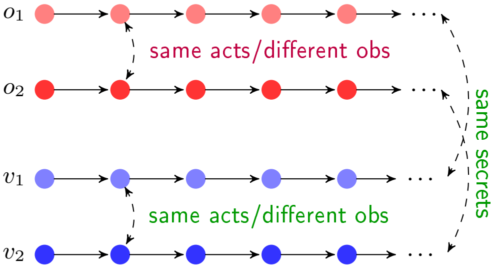
Contracts:

- *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

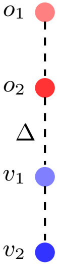
– *Secrecy*: same secrets for o_1, v_1 and o_2, v_2

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

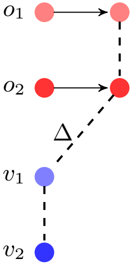


$\Delta \implies$ **Assumption + Contracts**

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**

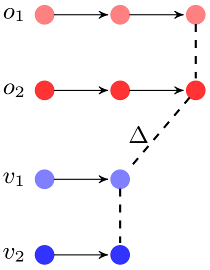


$\Delta \implies$ **Assumption + Contracts**

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**

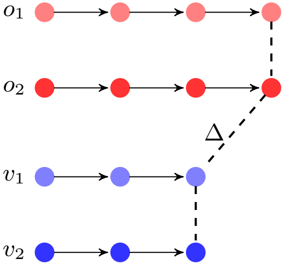


$\Delta \implies$ **Assumption + Contracts**

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**

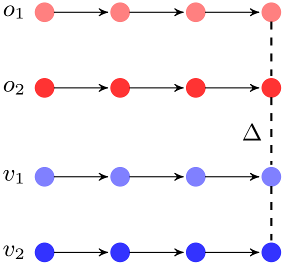


$\Delta \implies$ **Assumption + Contracts**

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**

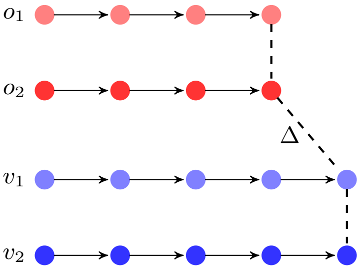


$\Delta \implies$ **Assumption + Contracts**

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**

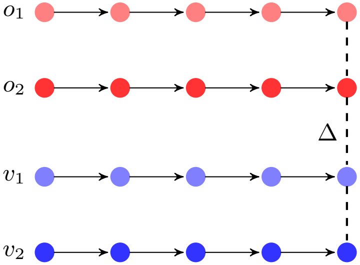


$\Delta \implies$ **Assumption + Contracts**

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**



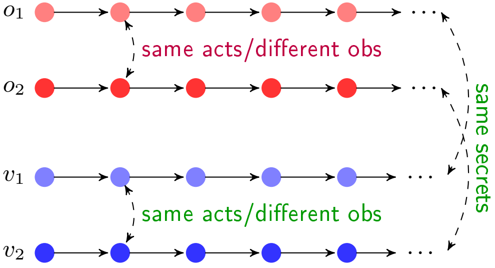
$\Delta \implies$ **Assumption + Contracts**

....

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**

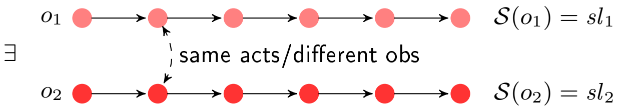
....

Assumption + Contracts $\implies$ *Relative Security*

Disproving Relative Security

Disproof

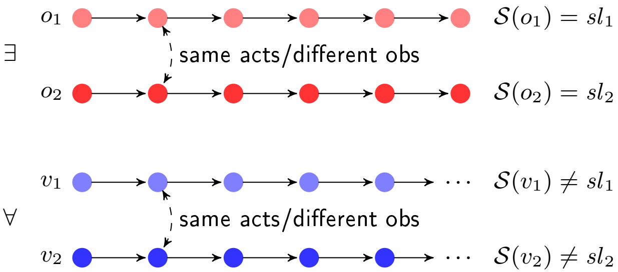
1) Provide traces (o_1, o_2) producing a concrete leak (sl_1, sl_2)



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

```
uint8_t function_v01(unsigned i) {
   if (i < N) {
     uint8_t v = a1[i];
     return a2[v];
   }
   return 0;}
```

Relatively Secure? (??)

Relative Security for Speculative Execution

```
uint8_t function_v01(unsigned i) {
   if (i < N) {
     uint8_t v = a1[i];
     return a2[v];
   }
   return 0;}
```

Not Relatively Secure (👎)

Relative Security for Speculative Execution

```c
uint8_t function_v01(unsigned i) {
   if (i < N) {
     uint8_t v = a1[i];
     return a2[v];
   }
   return 0;}
```

Not Relatively Secure (👎)

```c
uint8_t v01_secure(unsigned i) {
   if (i < N) {
      _mm_lfence(); //resolve spec
      uint8_t v = a1[i];
      return a2[v];
   }
   return 0;}
```

Relatively Secure? (??)

Relative Security for Speculative Execution

```
uint8_t function_v01(unsigned i) {
   if (i < N) {
     uint8_t v = a1[i];
     return a2[v];
   }
   return 0;}
```

Not Relatively Secure

```
uint8_t v01_secure(unsigned i) {
   if (i < N) {
      _mm_lfence(); //resolve spec
      uint8_t v = a1[i];
      return a2[v];
   }
   return 0;}
```

Relatively Secure!

Relative Security for Speculative Execution

```
uint8_t function_v01(unsigned i) {
  if (i < N) {
    uint8_t v = a1[i];
    return a2[v];
  }
  return 0;}
```

Not Relatively Secure ()

```
uint8_t v01_secure(unsigned i) {
  if (i < N) {
    _mm_lfence(); //resolve spec
    uint8_t v = a1[i];
    return a2[v];
  }
  return 0;}
```

Relatively Secure! ()

```
uint8_t v01_secure_2(unsigned i) {
  if (i < N) {
    uint8_t v = a1[i];
    _mm_lfence();//resolve spec
    return a2[v];
  }
  return 0;}
```

Relatively secure? (??)

Relative Security for Speculative Execution

```c
uint8_t function_v01(unsigned i) {
   if (i < N) {
     uint8_t v = a1[i];
     return a2[v];
   }
   return 0;}
```

Not Relatively Secure ()

```c
uint8_t v01_secure(unsigned i) {
   if (i < N) {
      _mm_lfence(); //resolve spec
      uint8_t v = a1[i];
      return a2[v];
   }
   return 0;}
```

Relatively Secure! ()

```c
uint8_t v01_secure_2(unsigned i) {
   if (i < N) {
      uint8_t v = a1[i];
      _mm_lfence();//resolve spec
      return a2[v];
   }
   return 0;}
```

Relatively Secure! ()

Relative Security vs TPOD

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

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

Relative Security vs TPOD

```c
uint8_t cond_secure(unsigned i) {
  if (i < N) {//Assuming N > 0...
    uint8_t v = a1[0]; //ind. of i
    return a2[v];
  }
  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

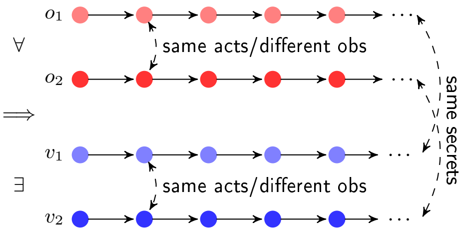
- Instantiation to a C-like language with speculative semantics
- Case studies from the Spectre benchmark verified
- An Isabelle/HOL mechanization of the general framework and the case studies

Contact me: jwright8@sheffield.ac.uk

Appendix

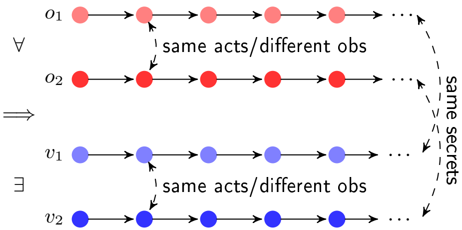
Relative Security vs TPOD

Relative Security

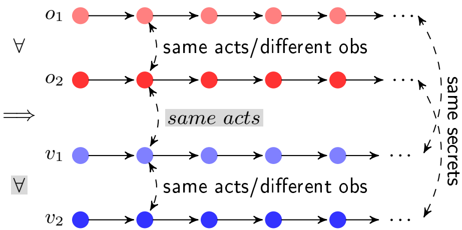


Relative Security vs TPOD

Relative Security



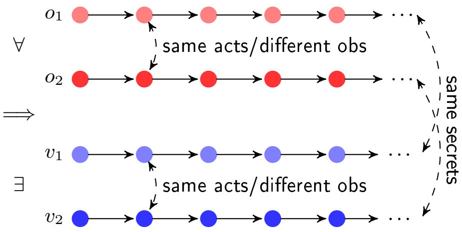
Trace-property dependent observational nondeterminism (TPOD)



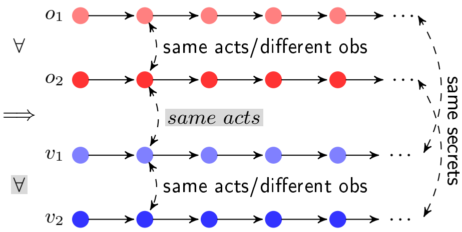
K. Cheang, C. Rasmussen, S. A. Seshia, and P. Subramanyan, "A formal approach to secure speculation," in CSF. IEEE, 2019

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

```c
uint8_t cond_secure(unsigned i) {
  if (i < N) {//Assuming N > 0...
    uint8_t v = a1[0]; //ind. of i
    return a2[v];
  }
  return 0;}
```

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

Relative Security vs. TPOD

```c
uint8_t cond_secure(unsigned i) {
  if (i < N) {//Assuming N > 0...
    uint8_t v = a1[0]; //ind. of i
    return a2[v];
  }
  return 0;}
```

Relatively Secure! () / TPOD ()

Example proof of security

```c
uint8_t v01_secure(unsigned i) {
  if (i < N) {
    _mm_lfence();//resolve spec
    uint8_t v = a1[i];
    return a2[v];
  }
  return 0;}
```

Δ_0

	Spec.	$v_1 = v_2 = o_1^0 = o_2^0$	$o_1^1 = o_2^1$	Memory invariants	Read locs. invariants
Δ_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

```c
uint8_t v01_secure(unsigned i) {
  if (i < N) {
    _mm_lfence();//resolve spec
    uint8_t v = a1[i];
    return a2[v];
  }
  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 ($\mathcal{O}$)
Δ_0	No	1	–	$\mathcal{S}(v_1) = \mathcal{S}(o_1) \wedge$ $\mathcal{S}(v_2) = \mathcal{S}(o_2)$	$v_1 = o_1 \wedge$ $v_2 = o_2$
Δ_1	No	2–7	–	$\cdots \wedge$ $v_1 =_i o_1$	$\cdots$

Example proof of security

```c
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 ($\mathcal{O}$)
Δ_0	No	1	–	$\mathcal{S}(v_1) = \mathcal{S}(o_1) \wedge$ $\mathcal{S}(v_2) = \mathcal{S}(o_2)$	$v_1 = o_1 \wedge$ $v_2 = o_2$
Δ_1	No	2–7	–	$\cdots \wedge$ $v_1 =_i o_1$	$\cdots$
Δ_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;}
```

$$\begin{tikzcd} \Delta_0 \arrow[r] & \Delta_1 \arrow[r] \arrow[dr] & \Delta_e \\ & & \color{purple}{\Delta_2} \end{tikzcd}$$

	Spec.	$v_1 = v_2 =$ $o_1^0 = o_2^0$	$o_1^1 = o_2^1$	Memory invariants	Read locs. invars ($\mathcal{O}$)
Δ_0	No	1	–	$\mathcal{S}(v_1) = \mathcal{S}(o_1) \wedge$ $\mathcal{S}(v_2) = \mathcal{S}(o_2)$	$v_1 = o_1 \wedge$ $v_2 = o_2$
Δ_1	No	2–7	–	$\cdots \wedge$ $v_1 =_i o_1$	$\cdots$
Δ_2	Yes	3	7	$\cdots$	$\cdots$
Δ_e	No	return	–	$\cdots$	$\cdots$

Example proof of security

```c
uint8_t v01_secure(unsigned i) {
  if (i < N) {
    _mm_lfence();//resolve spec
    uint8_t v = a1[i];
    return a2[v];
  }
  return 0;}
```

$$\begin{tikzcd} \Delta_0 \arrow[r] & \Delta_1 \arrow[r] \arrow[dr] & \Delta_e \\ & & \Delta_2 \arrow[ul, bend left] \end{tikzcd}$$

	Spec.	$v_1 = v_2 = o_1^0 = o_2^0$	$o_1^1 = o_2^1$	Memory invariants	Read locs. invars ($\mathcal{O}$)
Δ_0	No	1	–	$\mathcal{S}(v_1) = \mathcal{S}(o_1) \wedge \mathcal{S}(v_2) = \mathcal{S}(o_2)$	$v_1 = o_1 \wedge v_2 = o_2$
Δ_1	No	2–7	–	$\cdots \wedge v_1 =_i o_1$	$\cdots$
Δ_2	Yes	3	7	$\cdots$	$\cdots$
Δ_e	No	return	–	$\cdots$	$\cdots$

Example proof of security

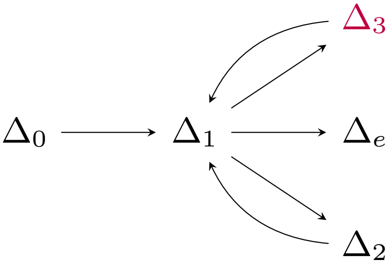
```
1  uint8_t v01_secure(unsigned i) {
2    if (i < N) {
3      _mm_lfence();//resolve spec
4      uint8_t v = a1[i];
5      return a2[v];
6    }
7    return 0;}
```

$$\begin{tikzcd}
 & & \Delta_3 \\
\Delta_0 \arrow[r] & \Delta_1 \arrow[ur] \arrow[r] \arrow[dr] & \Delta_e \\
 & & \Delta_2 \arrow[ul, bend left]
\end{tikzcd}$$

	Spec.	$v_1 = v_2 = o_1^0 = o_2^0$	$o_1^1 = o_2^1$	Memory invariants	Read locs. invars ($\mathcal{O}$)
Δ_0	No	1	–	$\mathcal{S}(v_1) = \mathcal{S}(o_1) \wedge$ $\mathcal{S}(v_2) = \mathcal{S}(o_2)$	$v_1 = o_1 \wedge$ $v_2 = o_2$
Δ_1	No	2–7	–	$\cdots \wedge$ $v_1 =_i o_1$	$\cdots$
Δ_2	Yes	3	7	$\cdots$	$\cdots$
Δ_3	Yes	7	3	$\cdots$	$\cdots$
Δ_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 ($\mathcal{O}$)
Δ_0	No	1	–	$\mathcal{S}(v_1) = \mathcal{S}(o_1) \wedge$ $\mathcal{S}(v_2) = \mathcal{S}(o_2)$	$v_1 = o_1 \wedge$ $v_2 = o_2$
Δ_1	No	2–7	–	$\cdots \wedge$ $v_1 =_i o_1$	$\cdots$
Δ_2	Yes	3	7	$\cdots$	$\cdots$
Δ_3	Yes	7	3	$\cdots$	$\cdots$
Δ_e	No	return	–	$\cdots$	$\cdots$

A simple language with speculative semantics - Syntax

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

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

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

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

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

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

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

Mispred

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

Resolve

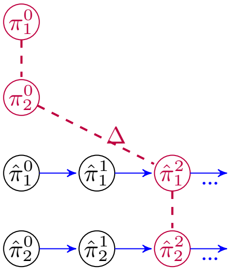
$$\frac{k > 0 \qquad \mathsf{resolve}(ps, pcs) \qquad C' = cfg_0 \cdot \ldots \cdot cfg_{k-1}}{(ps, cfg_0 \cdot \ldots \cdot cfg_k, inp, L) \Rightarrow_S (\mathsf{update}(ps, pcs), C', inp, L)}$$

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

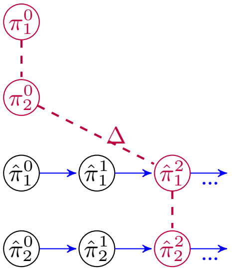
What if the player makes infinite independent steps?



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 ∞ when reacting)

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