Triosecuris: Formally Verified Protection Against Speculative Control-Flow Hijacking





Jonathan Baumann, Yonghyun Kim, Yan Farba, Cătălin Hriţcu, Julay Leatherman-Brooks

Max Planck Institute for Security and Privacy (MPI-SP)

Contributions

Triosecuris

Contributions

Triosecuris

- Protection against three kinds of **Spectre attacks**:

Contributions

Triosecuris

- Protection against three kinds of **Spectre attacks**:
  Spectre-BTB indirect calls

Contributions

Triosecuris

- Protection against three kinds of **Spectre attacks**:
  Spectre-BTB indirect calls
  Spectre-RSB returns

Contributions

Triosecuris

- Protection against three kinds of **Spectre attacks**:
  Spectre-BTB indirect calls
  Spectre-RSB returns
  Spectre-PHT conditional branches

Contributions

Triosecuris

- Protection against three kinds of **Spectre attacks**:
  Spectre-BTB indirect calls
  Spectre-RSB returns
  Spectre-PHT conditional branches
- combination of **hardware** and **software defenses**

Contributions

Triosecuris

- Protection against three kinds of **Spectre attacks**:
  Spectre-BTB indirect calls
  Spectre-RSB returns
  Spectre-PHT conditional branches
- combination of **hardware** and **software defenses**
- Protection for **arbitrary programs**

Contributions

Triosecuris

- Protection against three kinds of **Spectre attacks**:
  Spectre-BTB indirect calls
  Spectre-RSB returns
  Spectre-PHT conditional branches
- combination of **hardware** and **software defenses**
- Protection for **arbitrary programs**
- Verified in **Rocq**

Speculative Control-Flow Hijacking


```
l:    a <- secret;
      ... // f <- &fun1
      call f;
      ...

fun1:
      a <- public;
      b <- load[a];
      return;

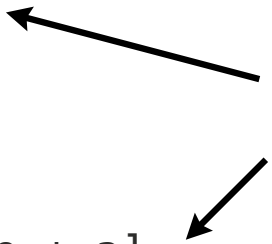
fun2:
      x <- load[base + a];
      y <- load[x];
      return;
```

Speculative Control-Flow Hijacking


```
l:    a <- secret;
      ... // f <- &fun1
      call f;
      ...

fun1:
      a <- public;
      b <- load[a];
      return;

fun2:
      x <- load[base + a];
      y <- load[x];
      return;
```



Memory accesses leak addresses via cache-timing attacks

Speculative Control-Flow Hijacking


```
l:    a <- secret;
      ... // f <- &fun1
      call f;
      ...

fun1:
      a <- public;
      b <- load[a];
      return;

fun2:
      x <- load[base + a];
      y <- load[x];
      return;
```

Indirect call

Memory accesses leak addresses via cache-timing attacks

Speculative Control-Flow Hijacking


```
l:    a <- secret;
      ... // f <- &fun1
      call f;
      ...

fun1:
      a <- public;
      b <- load[a];
      return;

fun2:
      x <- load[base + a];
      y <- load[x];
      return;
```

Value of `f` chosen dynamically

Indirect call

Speculative Control-Flow Hijacking


```
l:    a <- secret;
      ... // f <- &fun1
      call f;
      ...

fun1:
      a <- public;
      b <- load[a];
      return;

fun2:
      x <- load[base + a];
      y <- load[x];
      return;
```

Stalling until `f` resolves is **slow**

Speculative Control-Flow Hijacking


```
l:    a <- secret;
      ... // f <- &fun1
      call f;
      ...

fun1:
      a <- public;
      b <- load[a];
      return;

fun2:
      x <- load[base + a];
      y <- load[x];
      return;
```

Stalling until `f` resolves is **slow**

Instead:
CPU uses **Branch Target Buffer** (BTB)
to predict target location

Speculative Control-Flow Hijacking


```
l:    a <- secret;
      ... // f <- &fun1
      call f;
      ...

fun1:
      a <- public;
      b <- load[a];
      return;

fun2:
      x <- load[base + a];
      y <- load[x];
      return;
```

Stalling until `f` resolves is **slow**

Instead:
CPU uses **Branch Target Buffer** (BTB)
to predict target location
$\Longrightarrow$ **attacker-controllable**

Speculative Control-Flow Hijacking


```
l:    a <- secret;
      ... // f <- &fun1
      call f;
      ...

fun1:
      a <- public;
      b <- load[a];
      return;

fun2:
      x <- load[base + a];
      y <- load[x];
      return;
```

Stalling until `f` resolves is **slow**

Instead:
CPU uses **Branch Target Buffer** (BTB)
to predict target location
$\Longrightarrow$ **attacker-controllable**
$\Longrightarrow$ can predict **arbitrary locations**

Speculative Control-Flow Hijacking


```
l:    a <- secret;
      ... // f <- &fun1
      call f;
      ...

fun1:
      a <- public;
      b <- load[a];
      return;

fun2:
      x <- load[base + a];
      y <- load[x];
      return;
```

Stalling until `f` resolves is **slow**

Instead:
CPU uses **Branch Target Buffer** (BTB)
to predict target location
⟹ **attacker-controllable**
⟹ can predict **arbitrary locations**

Speculative Control-Flow Hijacking


```
l:    a <- secret;
      ... // f <- &fun1
      call f;
      ...

fun1:
      a <- public;
      b <- load[a];
      return;

fun2:
      x <- load[base + a];
      y <- load[x];
      return;
```

Stalling until `f` resolves is **slow**

Instead:
CPU uses **Branch Target Buffer** (BTB)
to predict target location
⟹ **attacker-controllable**
⟹ can predict **arbitrary locations**

Hardware Defense: Indirect Branch Tracking (IBT)


```
l:    a <- secret;
      ... // f <- &fun1
      call f;
      ...

fun1: ctarget;
      a <- public;
      b <- load[a];
      return;

fun2: ctarget;
      x <- load[base + a];
      y <- load[x];
      return;
```

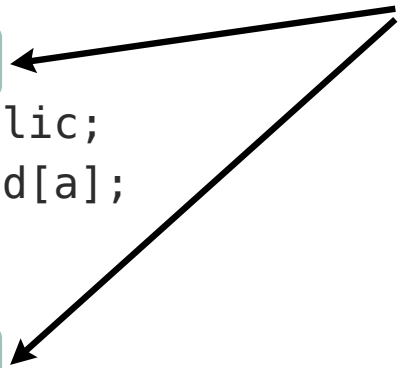
Hardware extension present in **recent Intel processors** as part of **CET**

Hardware Defense: Indirect Branch Tracking (IBT)


```
l:     a <- secret;
       ... // f <- &fun1
       call f;
       ...

fun1:  ctarget;
       a <- public;
       b <- load[a];
       return;

fun2:  ctarget;
       x <- load[base + a];
       y <- load[x];
       return;
```



Hardware extension present in **recent Intel processors** as part of **CET**

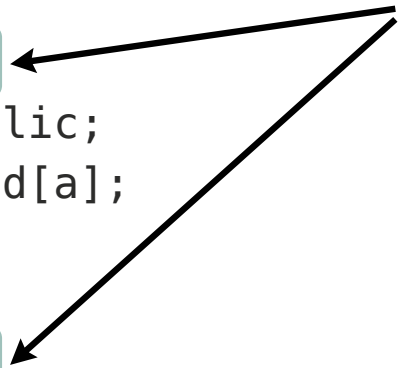
- special `ctarget` instruction marks **valid callable locations**

Hardware Defense: Indirect Branch Tracking (IBT)


```
l:    a <- secret;
      ... // f <- &fun1
      call f;
      ...

fun1: ctarget;
      a <- public;
      b <- load[a];
      return;

fun2: ctarget;
      x <- load[base + a];
      y <- load[x];
      return;
```



Hardware extension present in **recent Intel processors** as part of **CET**

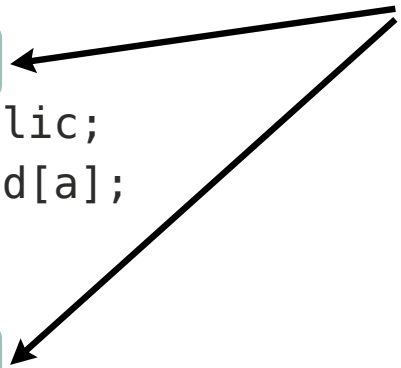
- special `ctarget` instruction marks **valid callable locations**
- hardware enforces that an indirect call must be followed by a `ctarget`

Hardware Defense: Indirect Branch Tracking (IBT)


```
l:    a <- secret;
      ... // f <- &fun1
      call f;
      ...

fun1: ctarget;
      a <- public;
      b <- load[a];
      return;

fun2: ctarget;
      x <- load[base + a];
      y <- load[x];
      return;
```



Hardware extension present in **recent Intel processors** as part of **CET**

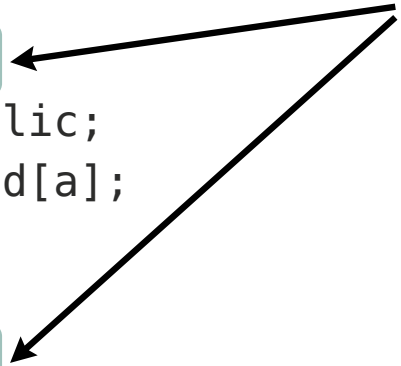
- special `ctarget` instruction marks **valid callable locations**
- hardware enforces that an indirect call must be followed by a `ctarget`
  - otherwise stops speculative execution

Hardware Defense: Indirect Branch Tracking (IBT)


```
l:    a <- secret;
      ... // f <- &fun1
      call f;
      ...

fun1: ctarget;
      a <- public;
      b <- load[a];
      return;

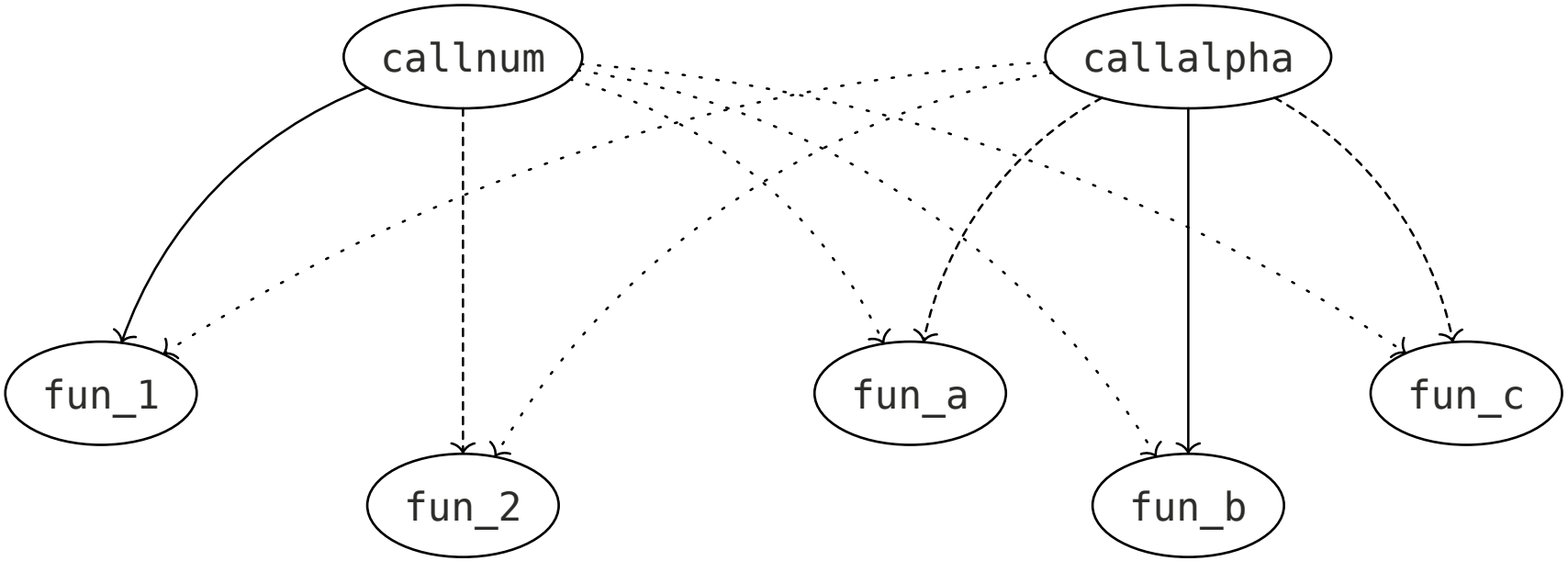
fun2: ctarget;
      x <- load[base + a];
      y <- load[x];
      return;
```



Hardware extension present in **recent Intel processors** as part of **CET**

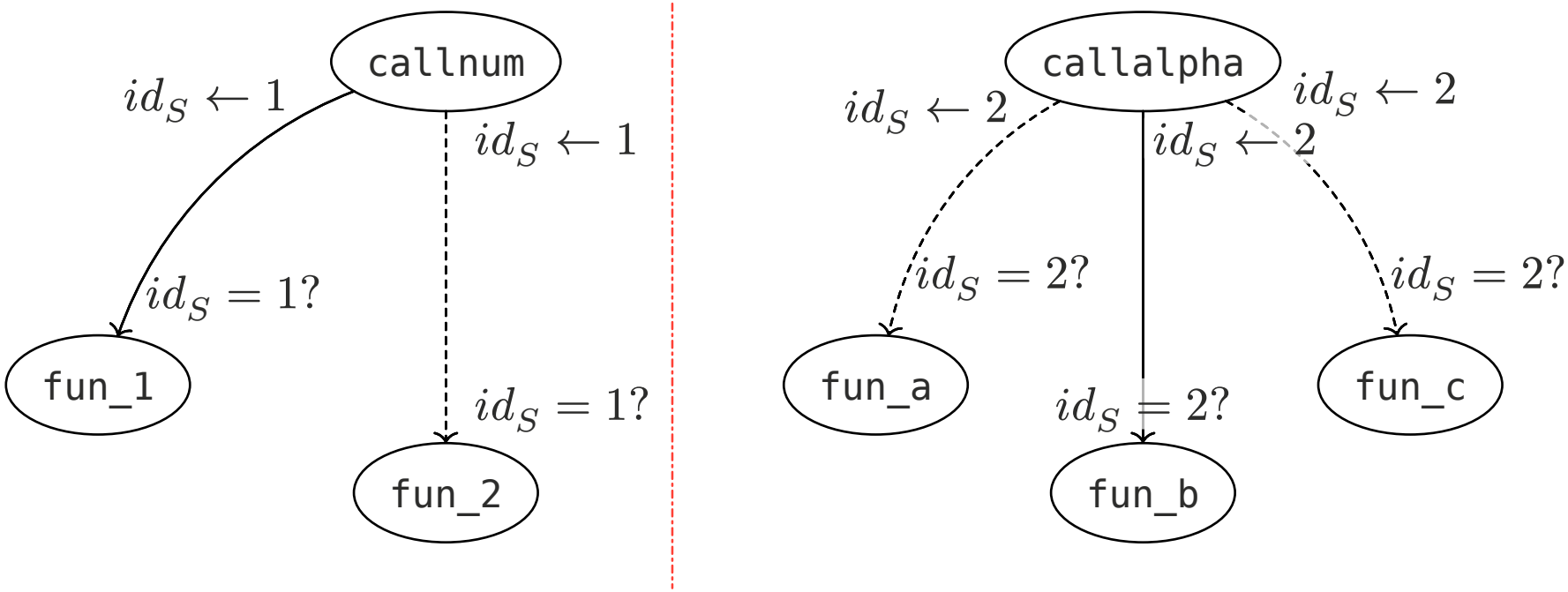
- special `ctarget` instruction marks **valid callable locations**
- hardware enforces that an indirect call must be followed by a `ctarget`
  - otherwise stops speculative execution
- restricts attack surface, but **no complete mitigation**!

IBT Granularity



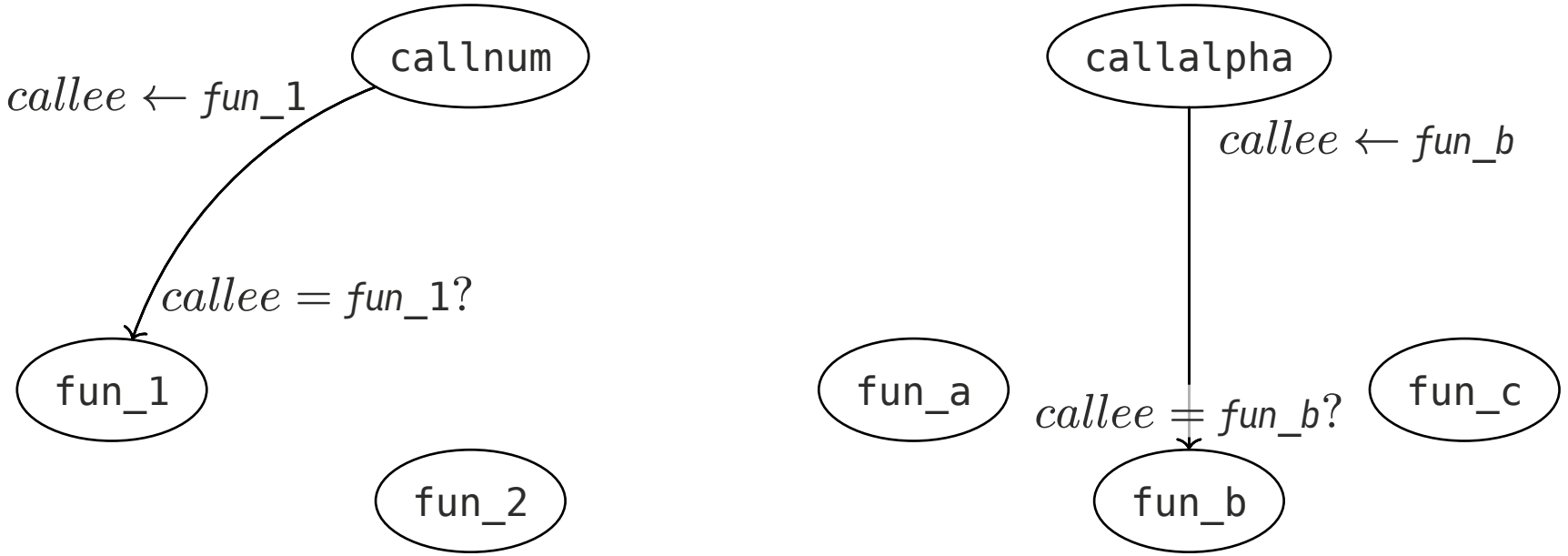
Hardware IBT: all calls can target all functions

IBT Granularity



FineIBT: functions partitioned into static sets

IBT Granularity



Triosecuris: precise detection of misprediction

From Detection To Protection

We leverage an existing mitigation against a related attack:
Ultimate SLH

From Detection To Protection

We leverage an existing mitigation against a related attack:

Ultimate SLH

- Detects **Spectre-PHT** (conditional branch misprediction) and tracks it with a **misspeculation flag**

From Detection To Protection

We leverage an existing mitigation against a related attack:

Ultimate SLH

- Detects **Spectre-PHT** (conditional branch misprediction) and tracks it with a **misspeculation flag**
- Prevents leakage by **masking the arguments** of vulnerable instructions with the misspeculation flag

From Detection To Protection

We leverage an existing mitigation against a related attack:

Ultimate SLH

- Detects **Spectre-PHT** (conditional branch misprediction) and tracks it with a **misspeculation flag**
- Prevents leakage by **masking the arguments** of vulnerable instructions with the misspeculation flag

```
b <- load[msf ? 0 : a]
```

  - Addresses of memory accesses

From Detection To Protection

We leverage an existing mitigation against a related attack:

Ultimate SLH

- Detects **Spectre-PHT** (conditional branch misprediction) and tracks it with a **misspeculation flag**
- Prevents leakage by **masking the arguments** of vulnerable instructions with the misspeculation flag

```
b <- load[msf ? 0 : a]

branch (msf ? 0 : e) to l
```

- Addresses of memory accesses
- Control-flow instructions

From Detection To Protection

We leverage an existing mitigation against a related attack:

Ultimate SLH

- Detects **Spectre-PHT** (conditional branch misprediction) and tracks it with a **misspeculation flag**
- Prevents leakage by **masking the arguments** of vulnerable instructions with the misspeculation flag

```
b <- load[msf ? 0 : a]

branch (msf ? 0 : e) to l

x <- div (msf ? 0 : p) (msf ? 0 : q)
```

  - Addresses of memory accesses
  - Control-flow instructions
  - Variable-timing instructions, e.g. division

From Detection To Protection

We leverage an existing mitigation against a related attack:
Ultimate SLH

- Detects **Spectre-PHT** (conditional branch misprediction) and tracks it with a **misspeculation flag**
- Prevents leakage by **masking the arguments** of vulnerable instructions with the misspeculation flag

```
b <- load[msf ? 0 : a]

branch (msf ? 0 : e) to l

x <- div (msf ? 0 : p) (msf ? 0 : q)
```

  - Addresses of memory accesses
  - Control-flow instructions
  - Variable-timing instructions, e.g. division

⟹ We can set the same **misspeculation flag** for **other speculation mechanisms**

Putting it Together


```
l:    a <- secret;
      ... // f <- &fun1

      call f;
      ...

fun1: ctarget;

      a <- public;
      b <- load[a];
      return;

fun2: ctarget;

      x <- load[base + a];
      y <- load[x];
      return;
```



marks callable locations

Putting it Together


```
l:    a <- secret;
      ... // f <- &fun1

      call f;
      ...

fun1: ctarget;

      a <- public;
      b <- load[a];
      return;

fun2: ctarget;

      x <- load[base + a];
      y <- load[x];
      return;
```



Putting it Together


```
l:    a <- secret;
      ... // f <- &fun1
      callee <- f;
      call f;
      ...

fun1: ctarget;

      a <- public;
      b <- load[msf ? 0 : a];
      return;

fun2: ctarget;

      x <- load[msf ? 0 : base + a];
      y <- load[msf ? 0 : x];
      return;
```

Putting it Together


```
l:    a <- secret;
      ... // f <- &fun1
      callee <- f;
      call f;
      ...

fun1: ctarget;
      msf <- callee = &fun1 ? msf : 1;
      a <- public;
      b <- load[msf ? 0 : a];
      return;

fun2: ctarget;
      msf <- callee = &fun2 ? msf : 1;
      x <- load[msf ? 0 : base + a];
      y <- load[msf ? 0 : x];
      return;
```

Putting it Together


```
l:    a <- secret;
      ... // f <- &fun1
      callee <- f;
      call f;
      ...

fun1: ctarget;
      msf <- callee = &fun1 ? msf : 1;
      a <- public;
      b <- load[msf ? 0 : a];
      return;

fun2: ctarget;
      msf <- callee = &fun2 ? msf : 1;
      x <- load[msf ? 0 : base + a];
      y <- load[msf ? 0 : x];
      return;
```

Putting it Together


```
l:    a <- secret;
      ... // f <- &fun1
      callee <- f;
      call f;
      ...

fun1: ctarget;
      msf <- callee = &fun1 ? msf : 1;
      a <- public;
      b <- load[msf ? 0 : a];
      return;

fun2: ctarget;
      msf <- callee = &fun2 ? msf : 1;
      x <- load[msf ? 0 : base + a];
      y <- load[msf ? 0 : x];
      return;
```

Securing Returns

Return instructions also allow for complex control flow

Securing Returns

Return instructions also allow for complex control flow
$\Longrightarrow$ and similar misspeculation!

Securing Returns

Return instructions also allow for complex control flow
$\Longrightarrow$ and similar misspeculation!

Securing Returns

Return instructions also allow for complex control flow
$\Longrightarrow$ and similar misspeculation!

Securing Returns

Return instructions also allow for complex control flow
$\Longrightarrow$ and similar misspeculation!

For our target hardware configurations:

Securing Returns

Return instructions also allow for complex control flow
$\Longrightarrow$ and similar misspeculation!

For our target hardware configurations:

Securing Returns

Return instructions also allow for complex control flow
$\Longrightarrow$ and similar misspeculation!

For our target hardware configurations:

Securing Returns

Return instructions also allow for complex control flow
$\Longrightarrow$ and similar misspeculation!

For our target hardware configurations:

- Predicted locations **must** correspond to a **valid call site**

Securing Returns

Return instructions also allow for complex control flow
$\Longrightarrow$ and similar misspeculation!

For our target hardware configurations:

- Predicted locations **must** correspond to a **valid call site**
  - add instrumentation after every call

Securing Returns


```
l:    a <- secret;
      ... // f <- &fun1
      callee <- f;
      call f;

      ...

fun1: ctarget;
      msf <- callee = &fun1 ? msf : 1;
      a <- public;
      b <- load[msf ? 0 : a];

      return;
```

Securing Returns


```
l:    a <- secret;
      ... // f <- &fun1
      callee <- f;
      call f;

      ...

fun1: ctarget;
      msf <- callee = &fun1 ? msf : 1;
      a <- public;
      b <- load[msf ? 0 : a];
      callee <- peek;
      return;
```

Securing Returns


```
l:    a <- secret;
      ... // f <- &fun1
      callee <- f;
      call f;

      ...

fun1: ctarget;
      msf <- callee = &fun1 ? msf : 1;
      a <- public;
      b <- load[msf ? 0 : a];
      callee <- peek;
      return;
```

reads return address from stack

Securing Returns


```
l:    a <- secret;
      ... // f <- &fun1
      callee <- f;
      call f;
      msf <- callee = (&l + 5) ? msf : 1;
      ...

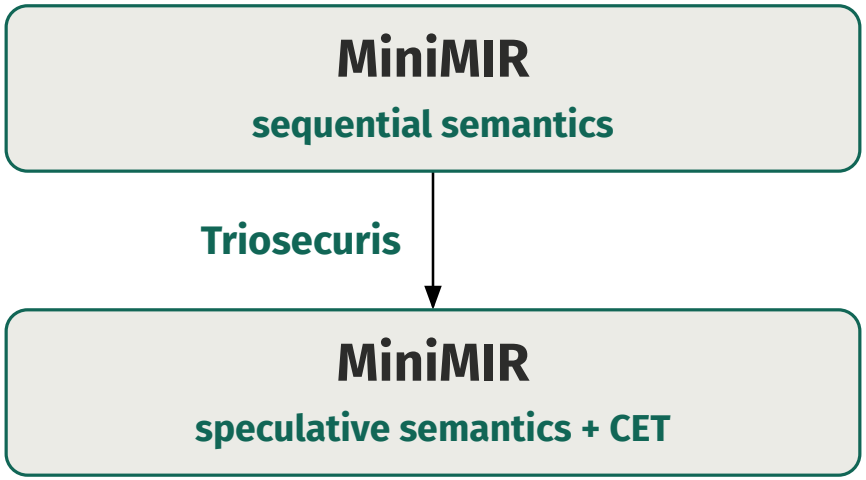
fun1: ctarget;
      msf <- callee = &fun1 ? msf : 1;
      a <- public;
      b <- load[msf ? 0 : a];
      callee <- peek;
      return;
```

Securing Returns


```
l:    a <- secret;
      ... // f <- &fun1
      callee <- f;
      call f;
      msf <- callee = (&l + 5) ? msf : 1;
      ...

fun1: ctarget;
      msf <- callee = &fun1 ? msf : 1;
      a <- public;
      b <- load[msf ? 0 : a];
      callee <- peek;
      return;
```

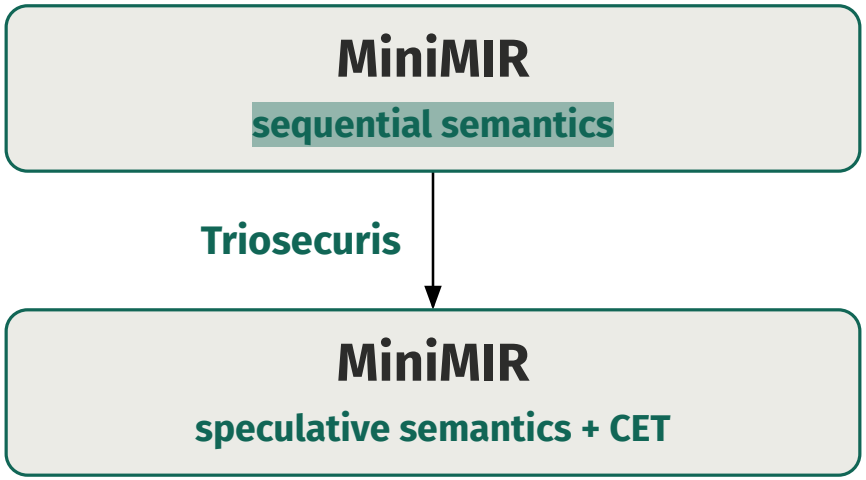
Formal Model



Triosecuris is a **compilation pass** — from MiniMIR to MiniMIR:

- MiniMIR: an **intermediate language** with **basic blocks** (like LLVM MIR) and abstract **code pointers**

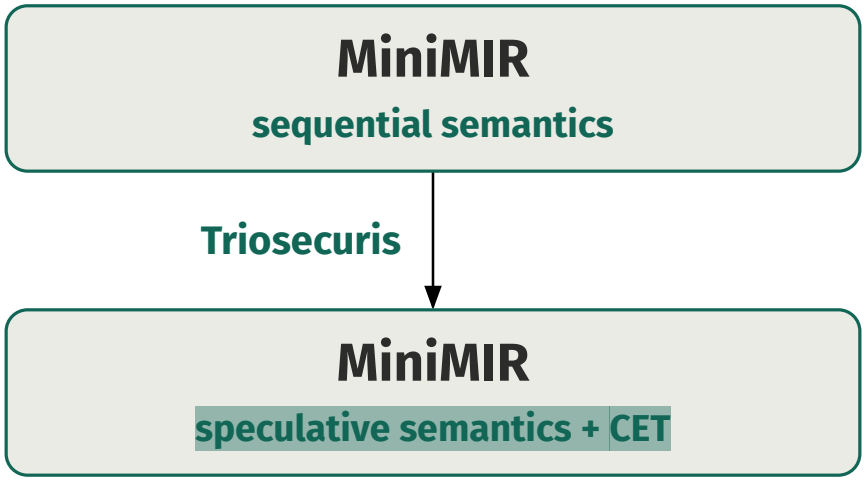
Formal Model



Triosecuris is a **compilation pass** — from MiniMIR to MiniMIR:

- MiniMIR: an **intermediate language** with **basic blocks** (like LLVM MIR) and abstract **code pointers**

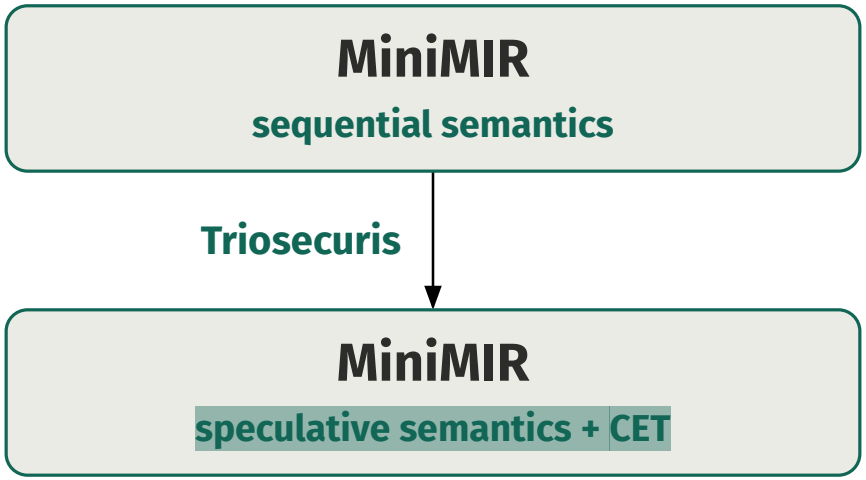
Formal Model



Triosecuris is a **compilation pass** — from MiniMIR to MiniMIR:

- MiniMIR: an **intermediate language** with **basic blocks** (like LLVM MIR) and abstract **code pointers**

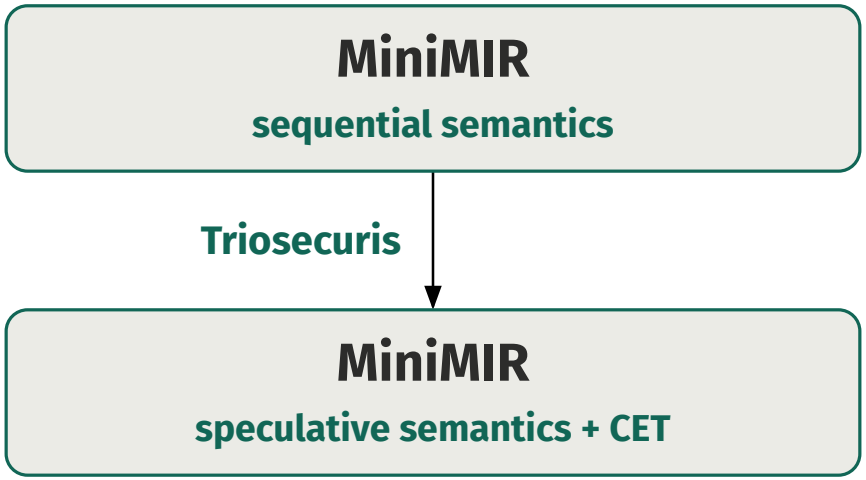
Formal Model



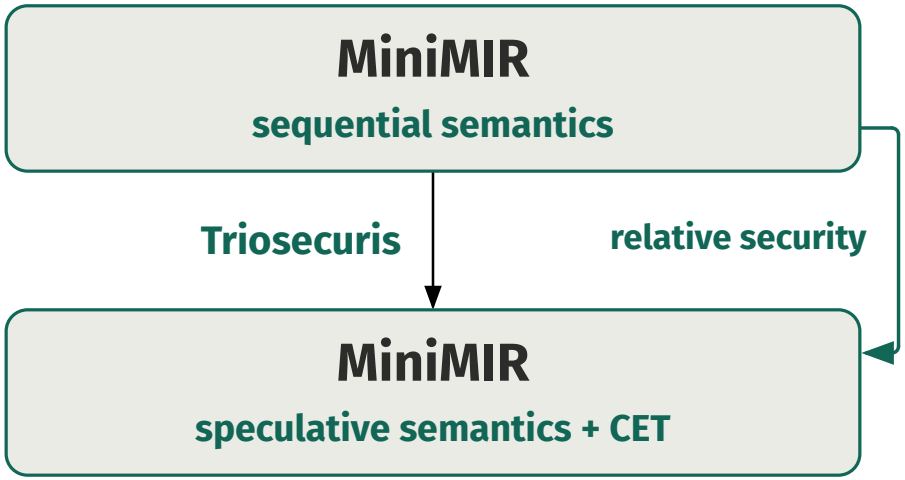
Triosecuris is a **compilation pass** — from MiniMIR to MiniMIR:

- MiniMIR: an **intermediate language** with **basic blocks** (like LLVM MIR) and abstract **code pointers**
- speculation is **over-approximated**: the attacker controls all predictions — **PHT**, **BTB**, **RSB** (Barthe et al. 2021)

Formal Guarantees



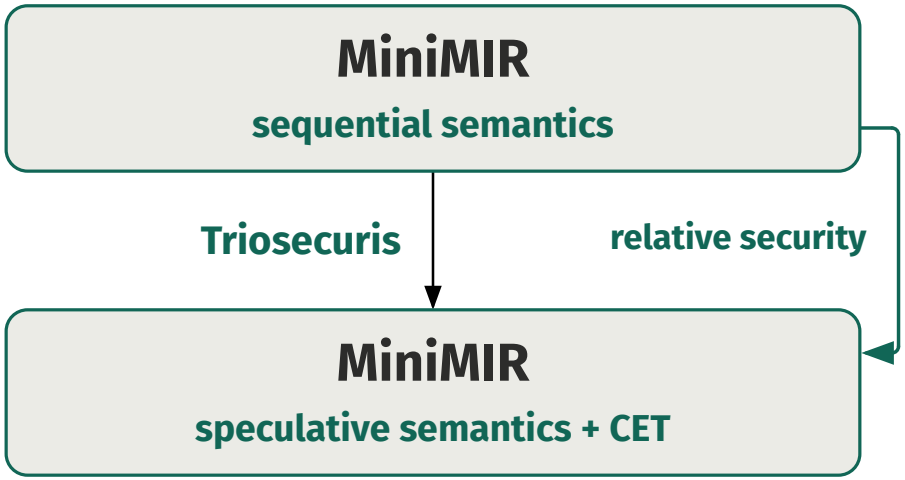
Formal Guarantees



We prove **relative security**:

- the hardened program under **speculation leaks no more** than the **source program** does **sequentially**

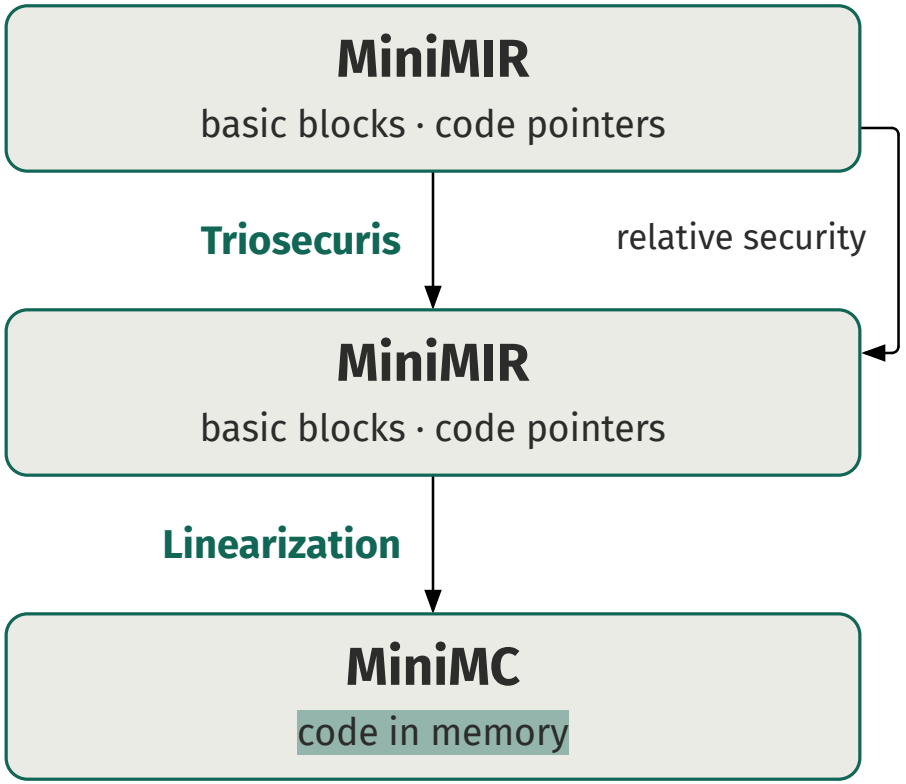
Formal Guarantees



We prove **relative security**:

- the hardened program under **speculation leaks no more** than the **source program** does **sequentially**
- holds for **arbitrary programs** — even **sequentially insecure** ones

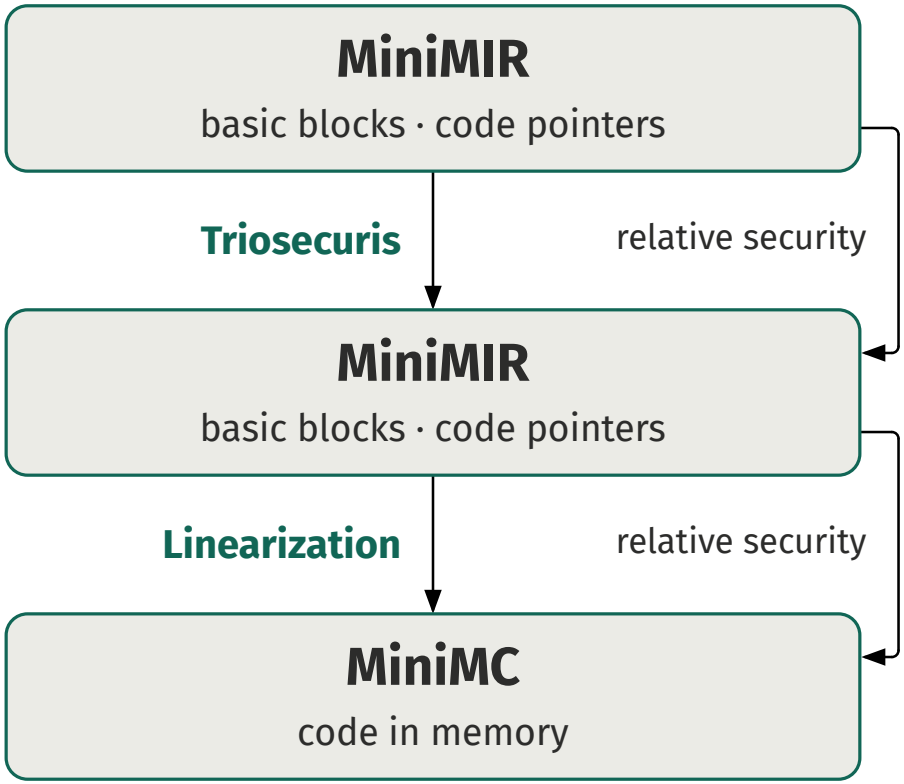
Toward a Lower Level



Guarantees carry down to a **lower level**:

- **linearization** — code in memory, code pointers become **integers**
  - ⟹ **refines undefined behavior (UB)**

Toward a Lower Level



Guarantees carry down to a **lower level**:

- **linearization** — code in memory, code pointers become **integers**
  - ⟹ **refines undefined behavior (UB)**
- **relative security** holds for **source executions without UB**

Safety Preservation

Linearization needs the **hardened program** to be **safe**:

Safety Preservation

Linearization needs the **hardened program** to be **safe**:

- but **masking** makes **ill-typed operations**:
  turning `42 <= load[a]` into `42 <= &fun1`

Safety Preservation

Linearization needs the **hardened program** to be **safe**:

- but **masking** makes **ill-typed operations**:
  turning `42 <= load[a]` into `42 <= &fun1`
- **postpone** with **undefined values** — from **creation** to **use**

Safety Preservation

Linearization needs the **hardened program** to be **safe**:

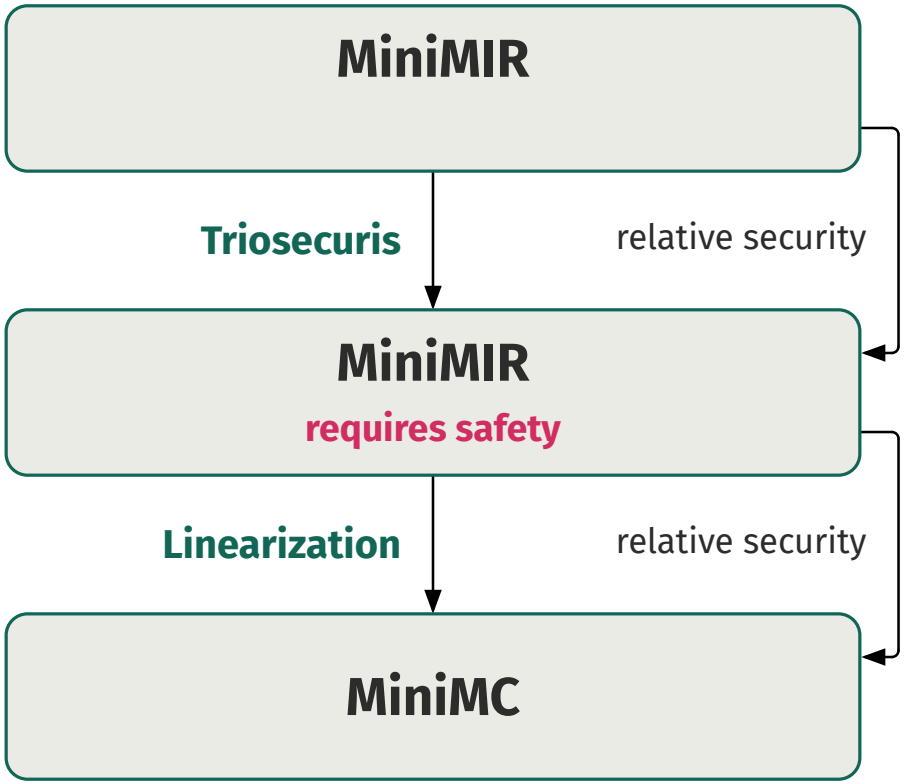
- but **masking** makes **ill-typed operations**:
  turning `42 <= load[a]` into `42 <= &fun1`
- **postpone** with **undefined values** — from **creation** to **use**
- its use is **prevented** by **masking** — the error **never happens**

Safety Preservation

Linearization needs the **hardened program** to be **safe**:

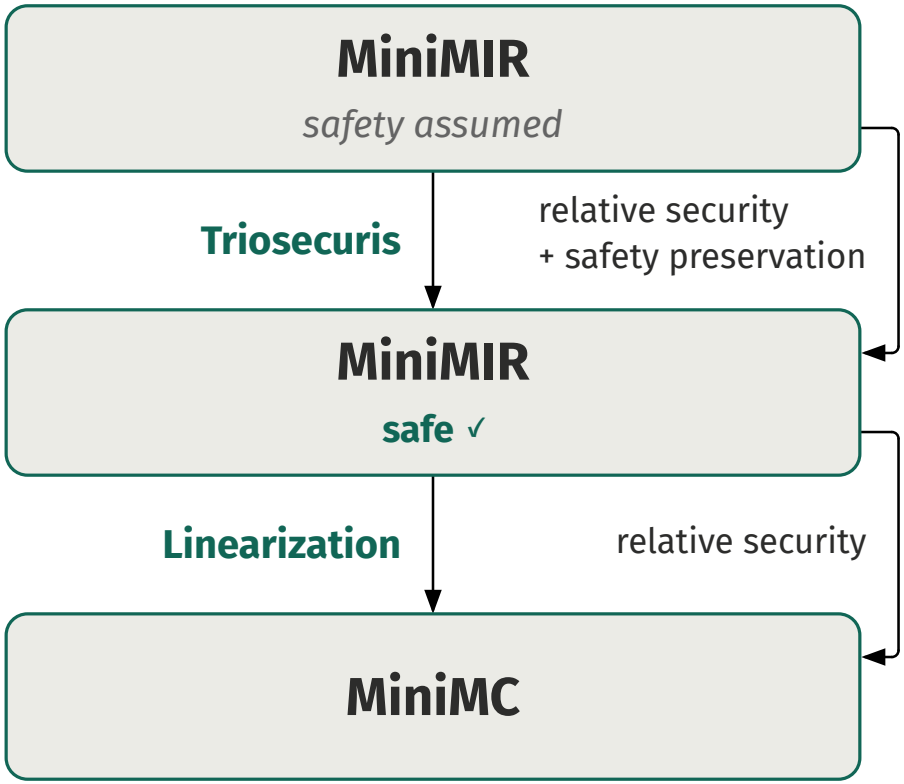
- but **masking** makes **ill-typed operations**:
  turning `42 <= load[a]` into `42 <= &fun1`
- **postpone** with **undefined values** — from **creation** to **use**
- its use is **prevented** by **masking** — the error **never happens**
- $\Longrightarrow$ **safety preservation**: if the source is safe, the hardened program is **safe**

End-to-End Guarantees



The **hardened program** must be **safe**:

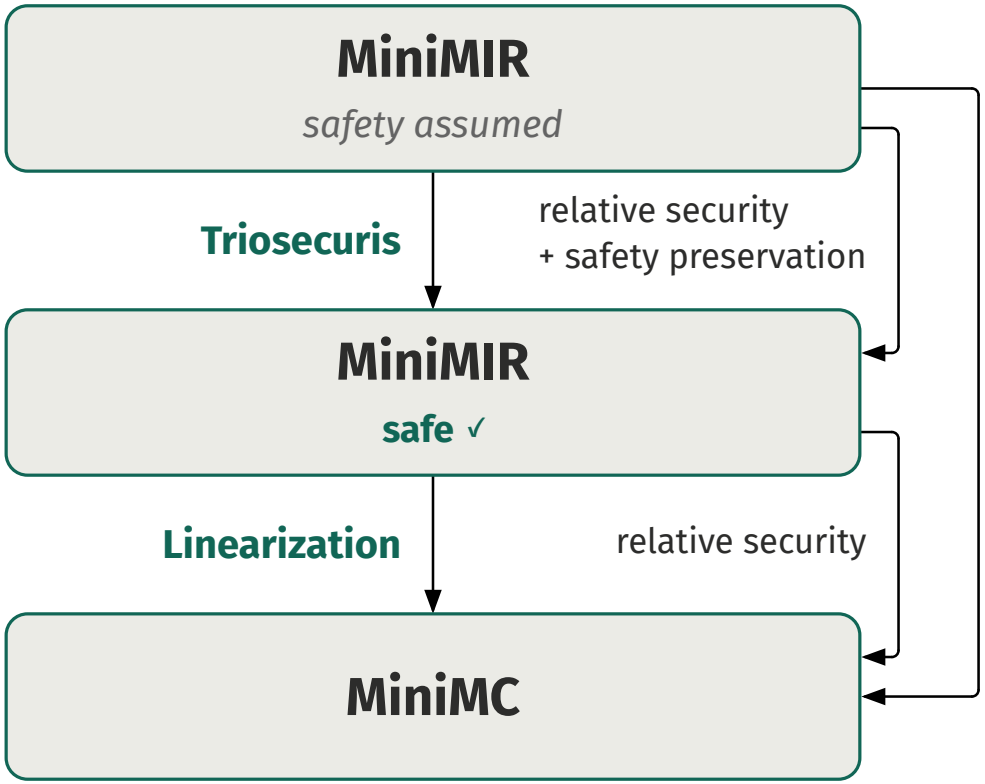
End-to-End Guarantees



The **hardened program** must be **safe**:

- **safety preservation** provides it

End-to-End Guarantees



The **hardened program** must be **safe**:

- **safety preservation** provides it
- $\Longrightarrow$ **end-to-end relative security**

Formally verified protection against speculative control-flow hijacking

- protection against three kinds of **Spectre attacks** — **PHT**, **BTB**, **RSB**

Formally verified protection against speculative control-flow hijacking

- protection against three kinds of **Spectre attacks** — **PHT**, **BTB**, **RSB**
- combination of **hardware** (**CET**) and **software** (**Ultimate SLH**) defenses

Formally verified protection against speculative control-flow hijacking

- protection against three kinds of **Spectre attacks** — **PHT**, **BTB**, **RSB**
- combination of **hardware** (**CET**) and **software** (**Ultimate SLH**) defenses
- protection for **arbitrary programs** — **relative security**

Formally verified protection against speculative control-flow hijacking

- protection against three kinds of **Spectre attacks** — **PHT**, **BTB**, **RSB**
- combination of **hardware** (**CET**) and **software** (**Ultimate SLH**) defenses
- protection for **arbitrary programs** — **relative security**
- verified in **Rocq** — fully machine-checked, ~**6,800** lines of proof
  - **property-based testing** before the proof — ~**2,500** lines

Future Work

- **implement** it in **LLVM** — extend the existing **Ultimate SLH** pass

Future Work

- **implement** it in **LLVM** — extend the existing **Ultimate SLH** pass
- **improve efficiency** — a **selective masking** algorithm (Baumann et al. 2025)
  (Ultimate SLH masks **everything**: **~150%** on SPEC)

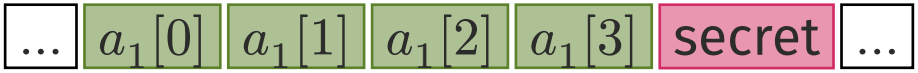
Future Work

- **implement** it in **LLVM** — extend the existing **Ultimate SLH** pass
- **improve efficiency** — a **selective masking** algorithm (Baumann et al. 2025)
  (Ultimate SLH masks **everything**: **~150%** on SPEC)
- **alternative mitigations** — formalize **SLH + retpoline + SafeRET**:
  protection **without CET**

Formally verified protection against speculative control-flow hijacking

- protection against three kinds of **Spectre attacks** — **PHT**, **BTB**, **RSB**
- combination of **hardware** (**CET**) and **software** (**Ultimate SLH**) defenses
- protection for **arbitrary programs** — **relative security**
- verified in **Rocq** — fully machine-checked, ~**6,800** lines of proof
  - **property-based testing** before the proof — ~**2,500** lines

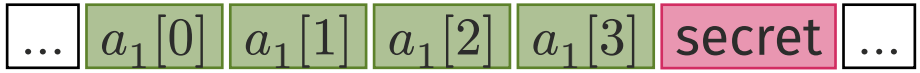
Spectre (PHT)




```
if i < size(a1) then
    j ← a1[i];
    x ← a2[j]

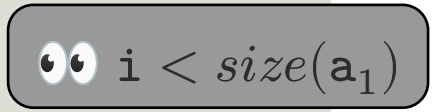
else
    ...
```

Spectre (PHT)

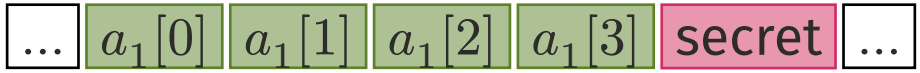



```
if i < size(a1) then
  j ← a1[i];
  x ← a2[j]

else
  ...
```

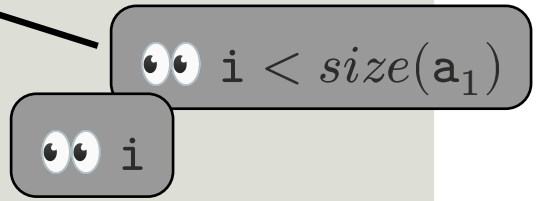


Spectre (PHT)

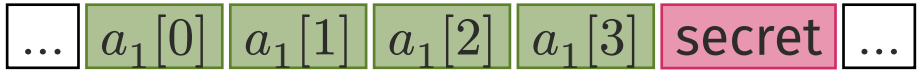



```
if i < size(a1) then
  j ← a1[i];
  x ← a2[j]

else
  ...
```



Spectre (PHT)




```
if i < size(a1) then
  j ← a1[i];
  x ← a2[j]

else
  ...
```

👀 $\mathtt{i} < size(\mathtt{a}_1)$

👀 $\mathtt{i}$

👀 $\mathtt{a}_1[\mathtt{i}]$

Spectre (PHT)

let $\mathtt{i} = 4$

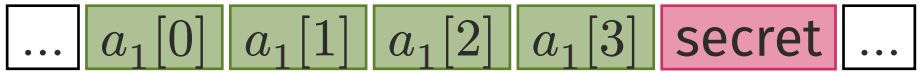



```
if i < size(a1) then
    j ← a1[i];
    x ← a2[j]

else
    ...
```

Spectre (PHT)

let $\mathtt{i} = 4$

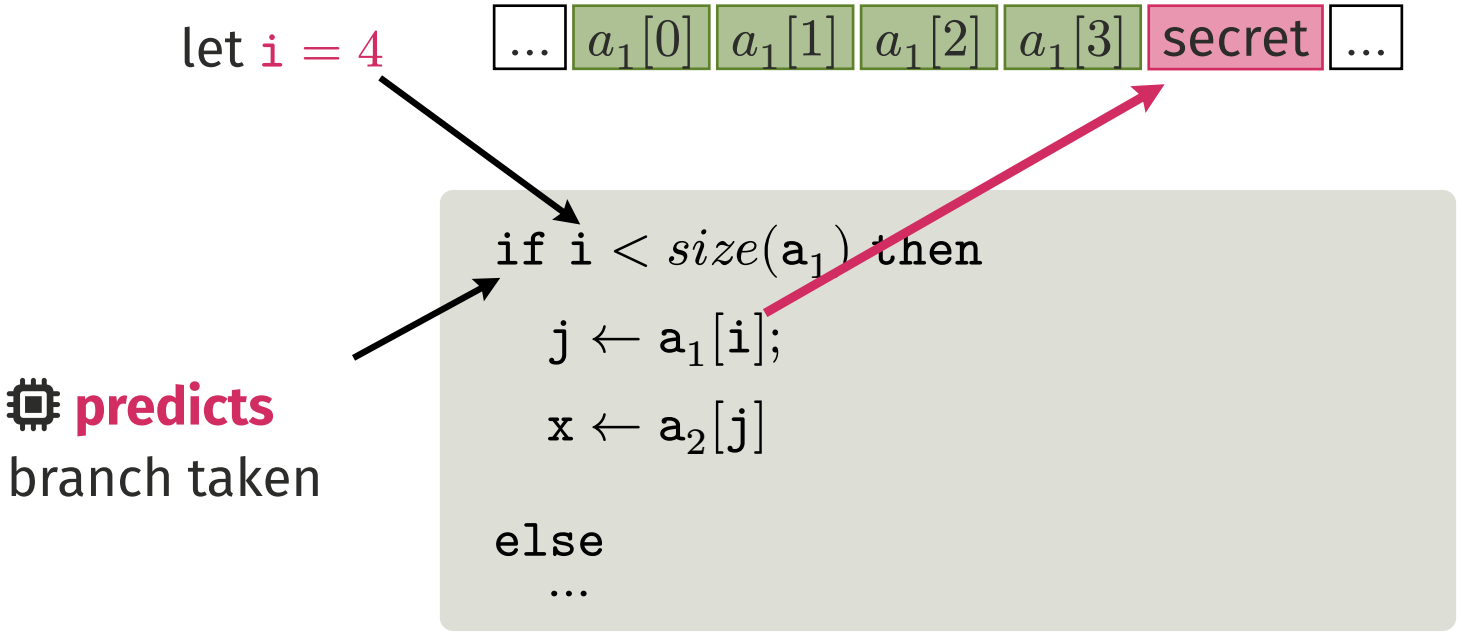


predicts
branch taken

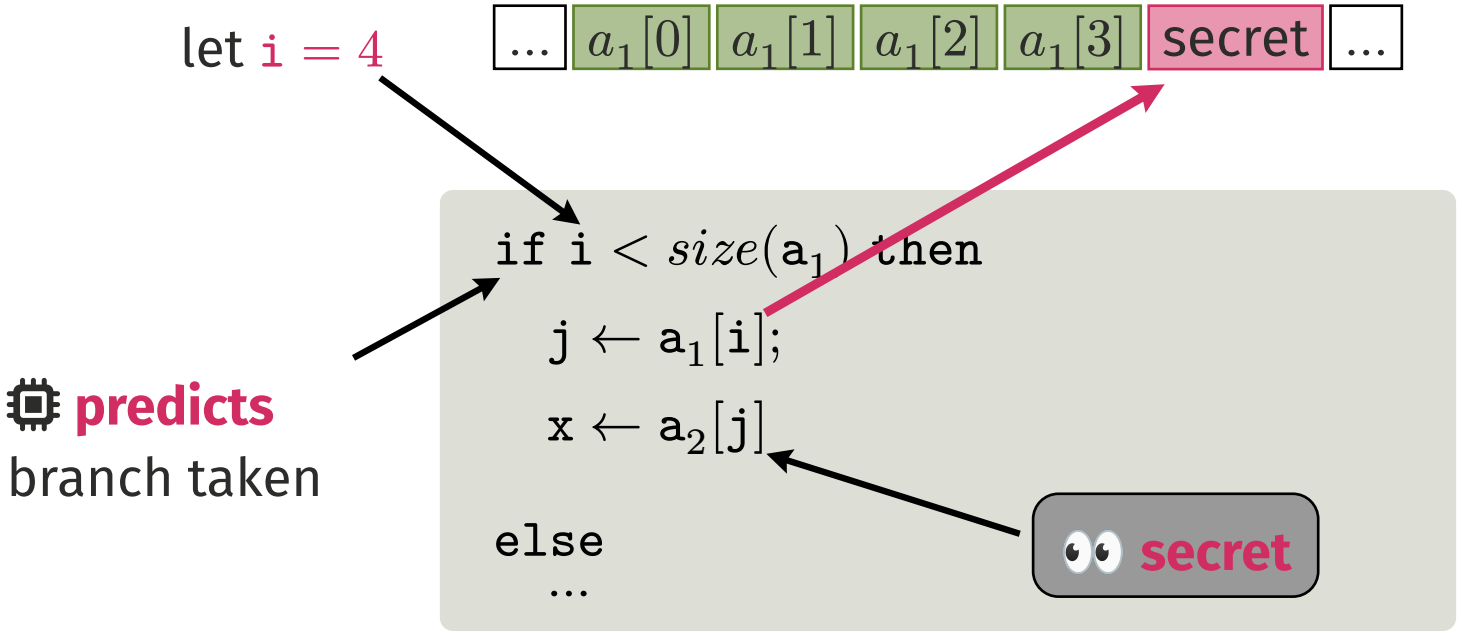
```
if i < size(a1) then
    j ← a1[i];
    x ← a2[j]

else
    ...
```

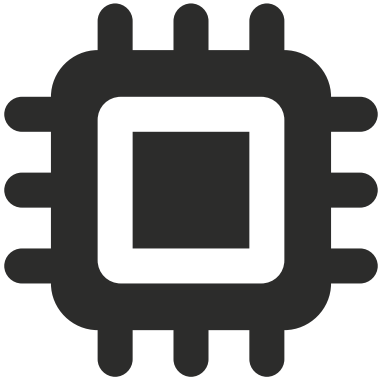
Spectre (PHT)


let i = 4
... $a_1[0]$ $a_1[1]$ $a_1[2]$ $a_1[3]$ secret ...
if i < $size(a_1)$ then
j ← a_1[i];
x ← a_2[j]
else
...
predicts
branch taken

Spectre (PHT)


let i = 4
... $a_1[0]$ $a_1[1]$ $a_1[2]$ $a_1[3]$ secret ...
if i < $size(a_1)$ then
j ← a_1[i];
x ← a_2[j]
else
...
predicts
branch taken
secret

Spectre Variants

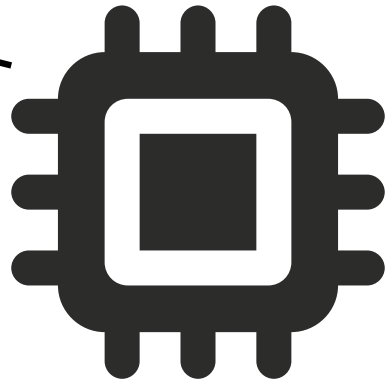


predicts

Spectre Variants

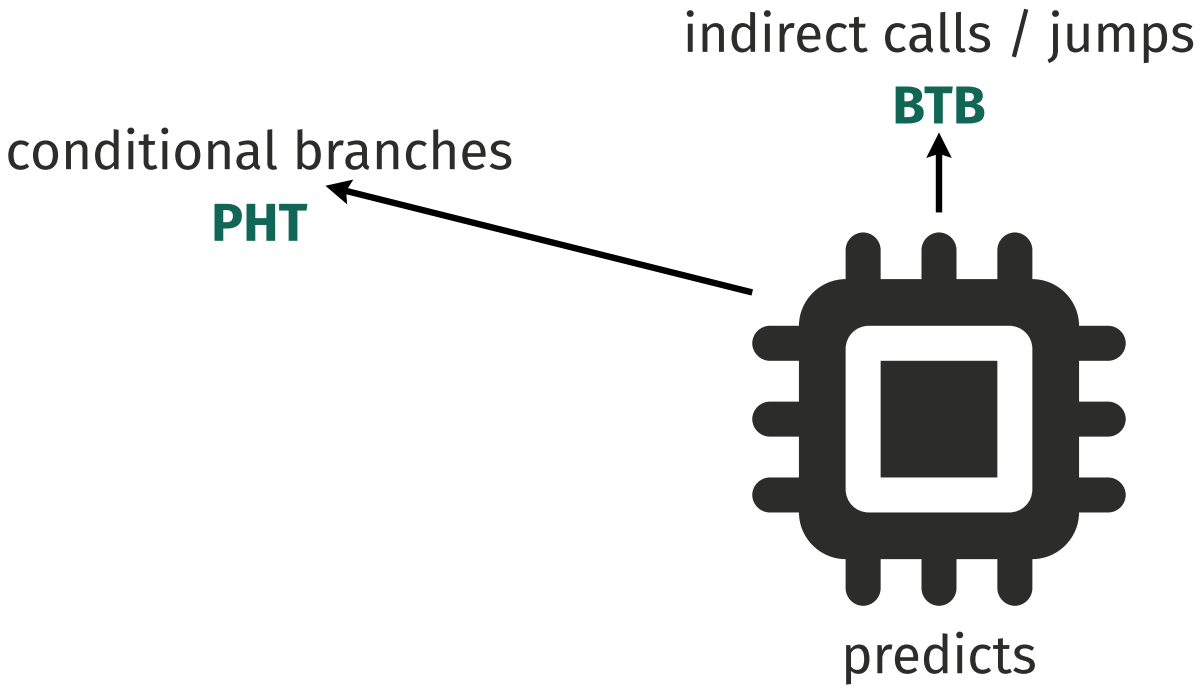
conditional branches

PHT

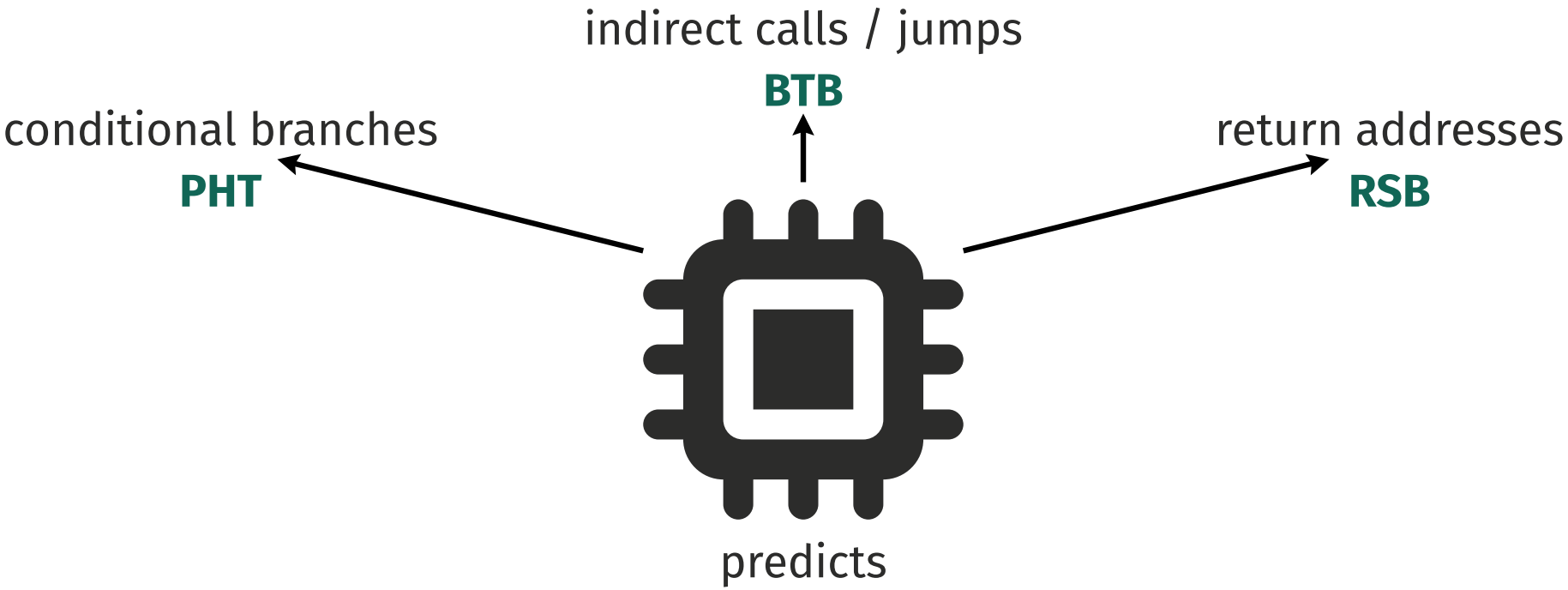


predicts

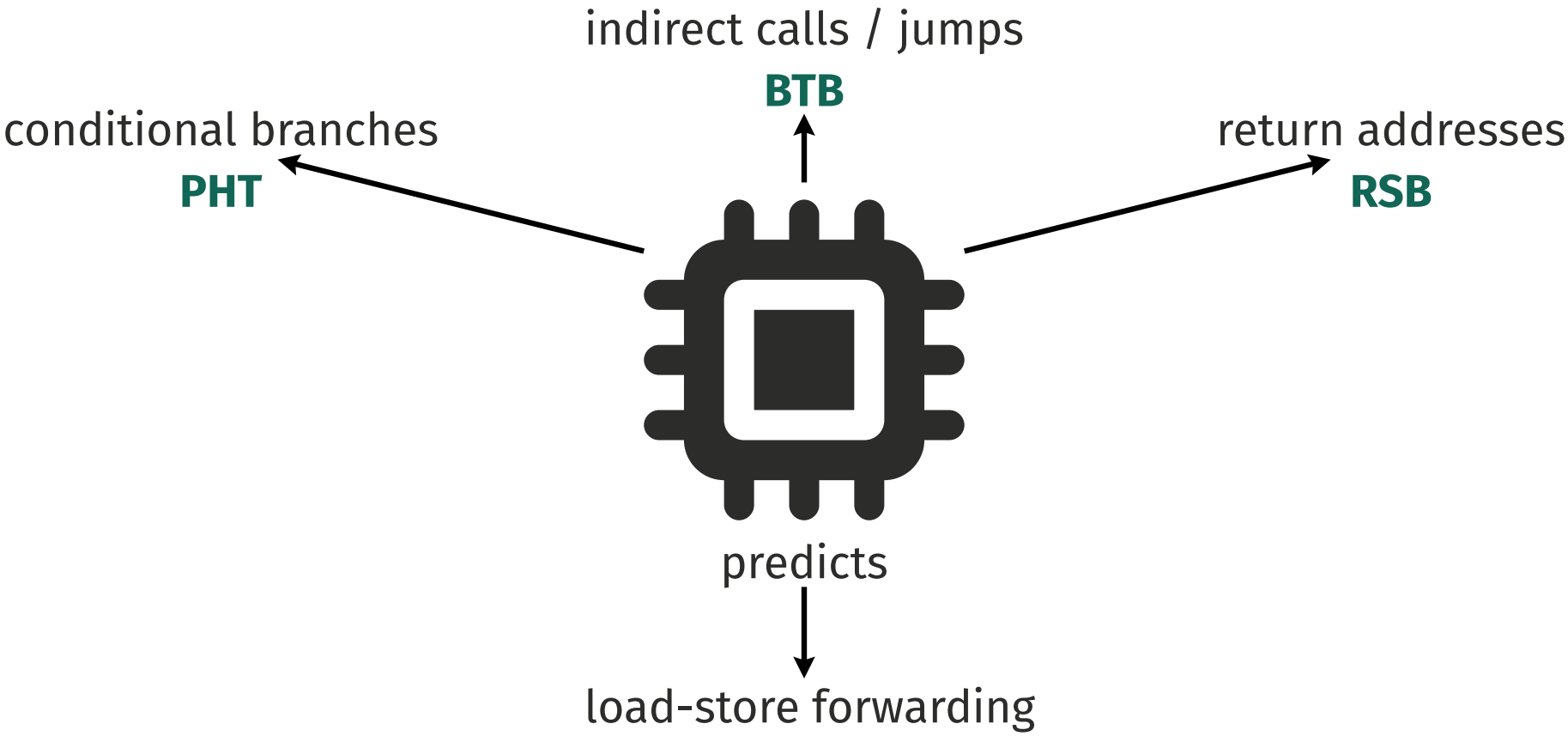
Spectre Variants



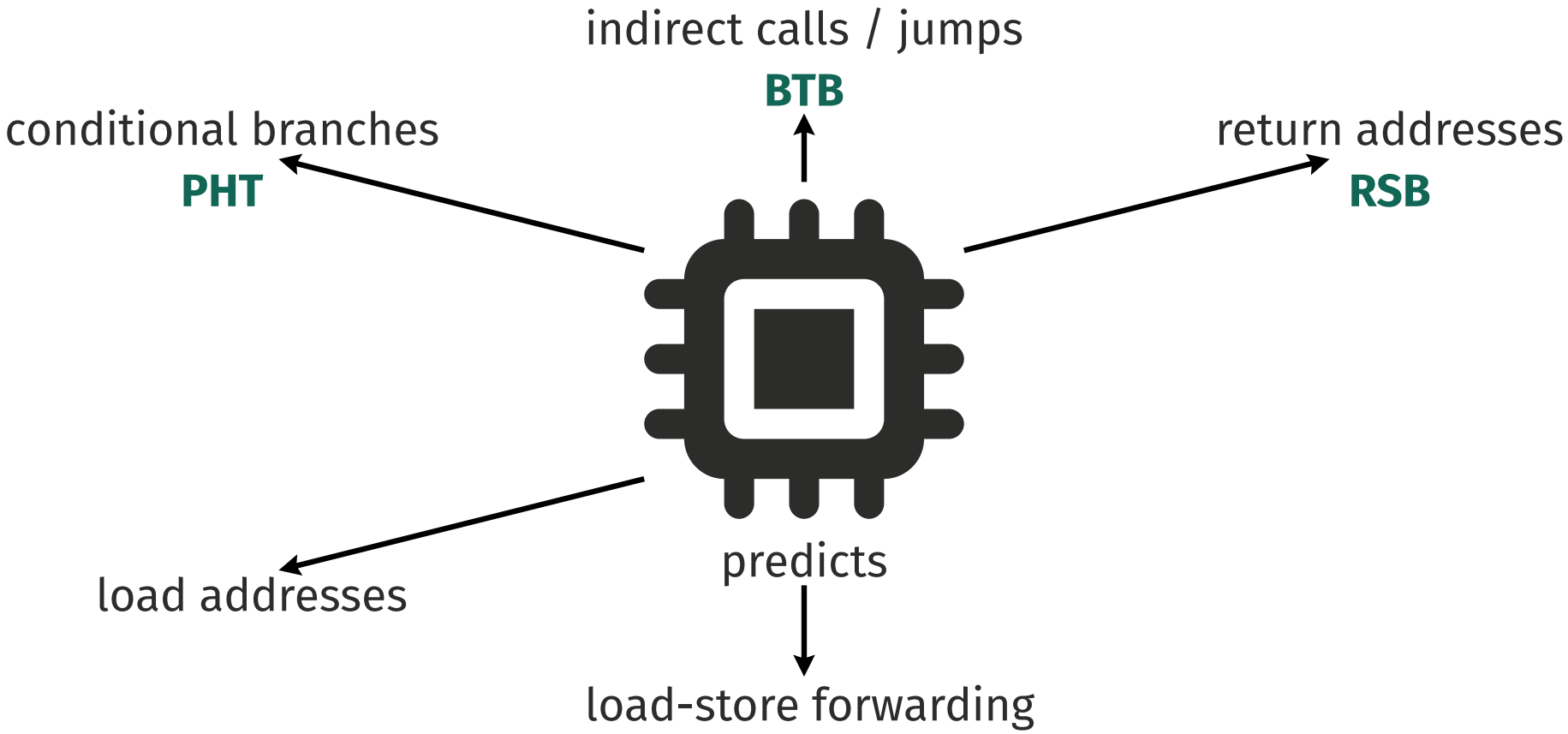
Spectre Variants



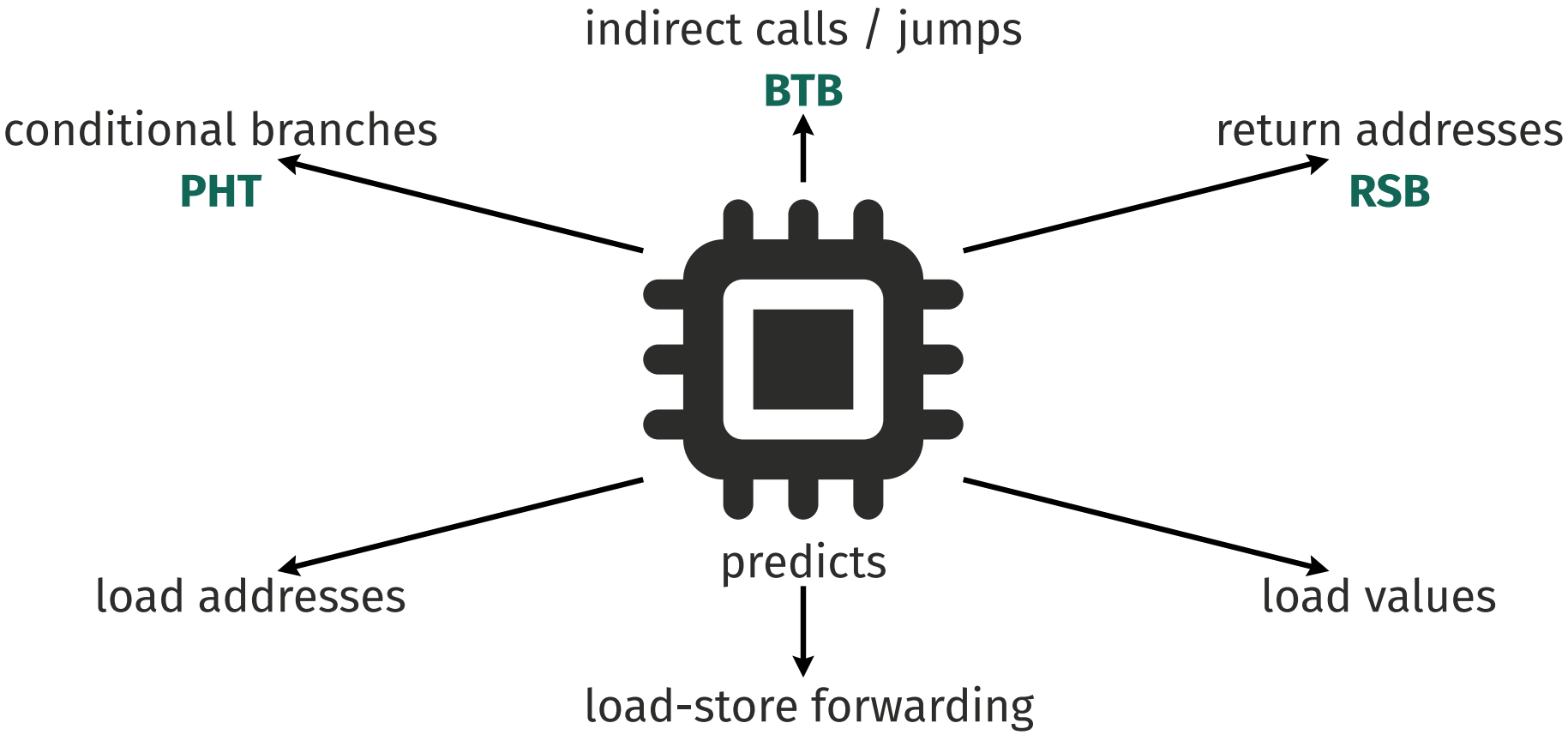
Spectre Variants



Spectre Variants



Spectre Variants



Ultimate SLH


```
if i < a1_size then

  j <- a1[i];
  x <- a2[j]
else
```

Software defense against Spectre PHT

Ultimate SLH

special register stores misspeculation flag

```
if i < a1_size then
  msf := (i < a1_size)? msf : 1;
  j <- a1[i];
  x <- a2[j]
else
  msf := (i < a1_size) ? 1 : msf;
```

Software defense against Spectre PHT

- keep track of misspeculation

Ultimate SLH

updated using **constant-time conditional**

```
if i < a1_size then
  msf := (i < a1_size) ? msf : 1;
  j <- a1[i];
  x <- a2[j]
else
  msf := (i < a1_size) ? 1 : msf;
```

Software defense against Spectre PHT

- keep track of misspeculation

Ultimate SLH


```
if i < a1_size then
  msf := (i < a1_size) ? msf : 1;
  j <- a1[i];
  x <- a2[j]
else
  msf := (i < a1_size) ? 1 : msf;
```

Software defense against Spectre PHT

- keep track of misspeculation
- prevent speculative leaks by **masking**:

Ultimate SLH


```
if i < a1_size then
  msf := (i < a1_size) ? msf : 1;
  j <- a1[msf == 1 ? 0 : i];
  x <- a2[msf == 1 ? 0 : j]
else
  msf := (i < a1_size) ? 1 : msf;
```

Software defense against Spectre PHT

- keep track of misspeculation
- prevent speculative leaks by **masking**:
  - addresses
  - branch conditions
  - call targets
  - operands of variable-time instructions

zero out when misspeculating

Formally Defined Attacker: Directives


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
→     call f;
      ...;

fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;

l2:   y <- load[x];
      return;
```

Formally Defined Attacker: Directives


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;

l2:   y <- load[x];
      return;
```

DCall (fun2, 1)

Formally Defined Attacker: Directives


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;

l2:   y <- load[x];
      return;
```

DCall (fun2, 1)

Formally Defined Attacker: Directives


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
→     branch true to l2;
      return;

l2:   y <- load[x];
      return;
```

DCall (fun2, 1)

DBranch false

Formally Defined Attacker: Directives


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;

→     l2:   y <- load[x];
      return;
```

DCall (fun2, 1)

DBranch false

Formally Defined Attacker: Directives


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;

l2:   y <- load[x];
      return;
```

DCall (fun2, 1)

DBranch false

Formally Defined Attacker: Directives


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;

l2:   y <- load[x];
      return;
```

DCall (fun2, 1)

DBranch false

DRet (fun1, 2)

Formally Defined Attacker: Directives


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

fun1: a <- public;
      call &fun2;
→     load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;

l2:   y <- load[x];
      return;
```

DCall (fun2, 1)

DBranch false

DRet (fun1, 2)

Formally Defined Attacker: Directives


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;
```

DCall (fun2, 1)

DBranch false

trace of directives: [DCall (fun2, 1)]

DRet (fun1, 2)

Formally Defined Attacker: Directives


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;
```

DCall (fun2, 1)

DBranch false

t (fun1, 2)

trace of directives: [DCall (fun2, 1); DBranch false]

Formally Defined Attacker: Directives


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;
```

DCall (fun2, 1)

DBranch false

trace of directives: [DCall (fun2, 1); DBranch false; DRet (fun1, 2)]

Formally Defined Attacker: Observation


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;

l2:   y <- load[x];
      return;
```

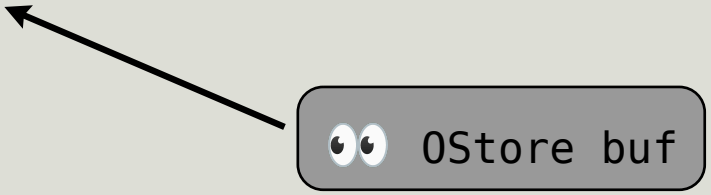
Formally Defined Attacker: Observation


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;

l2:   y <- load[x];
      return;
```



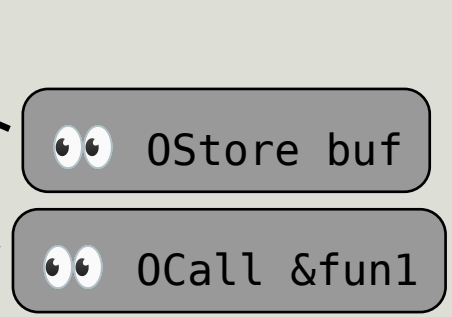
Formally Defined Attacker: Observation


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;

l2:   y <- load[x];
      return;
```



Formally Defined Attacker: Observation


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;

l2:   y <- load[x];
      return;
```



Formally Defined Attacker: Observation

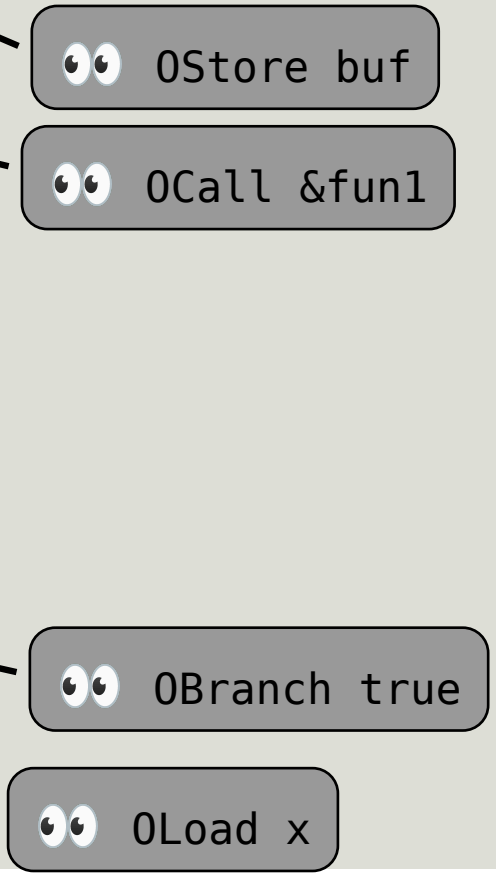
MAX PLANCK INSTITUTE FOR SECURITY AND PRIVACY

```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;

l2:   y <- load[x];
      return;
```



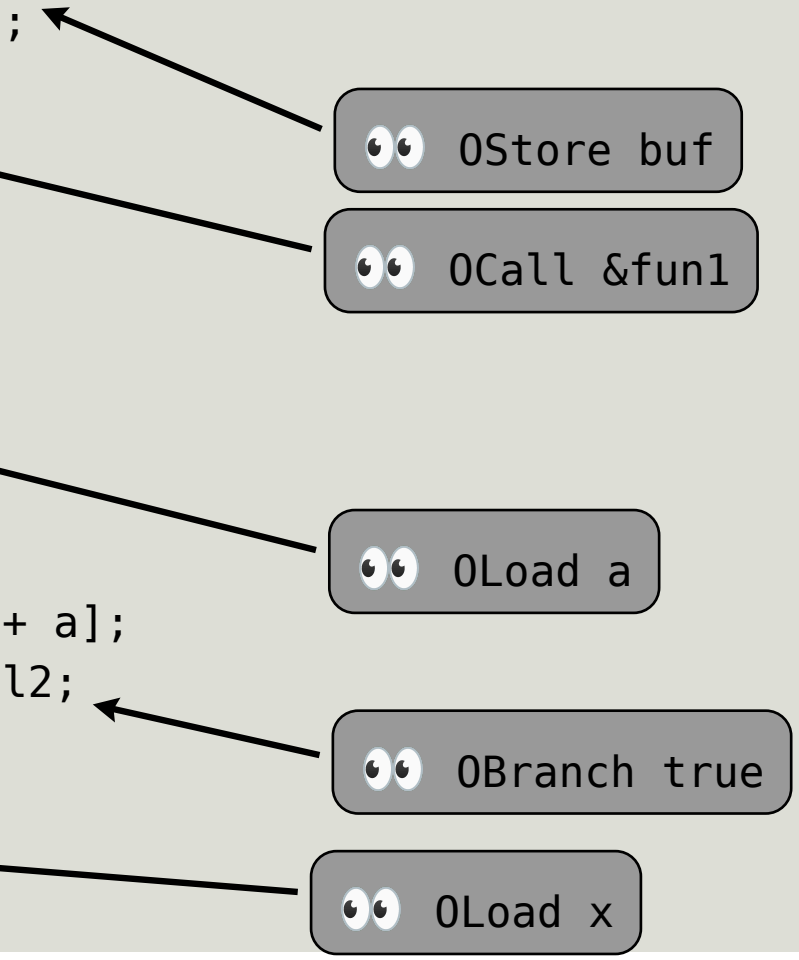
Formally Defined Attacker: Observation


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;

l2:   y <- load[x];
      return;
```



Formally Defined Attacker: Observation


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;
```

👀 OStore buf

👀 OCall &fun1

👀 OLoad a

👀 OBranch true

👀 OLoad x

trace of observations: `[OStore buf]`

Formally Defined Attacker: Observation


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

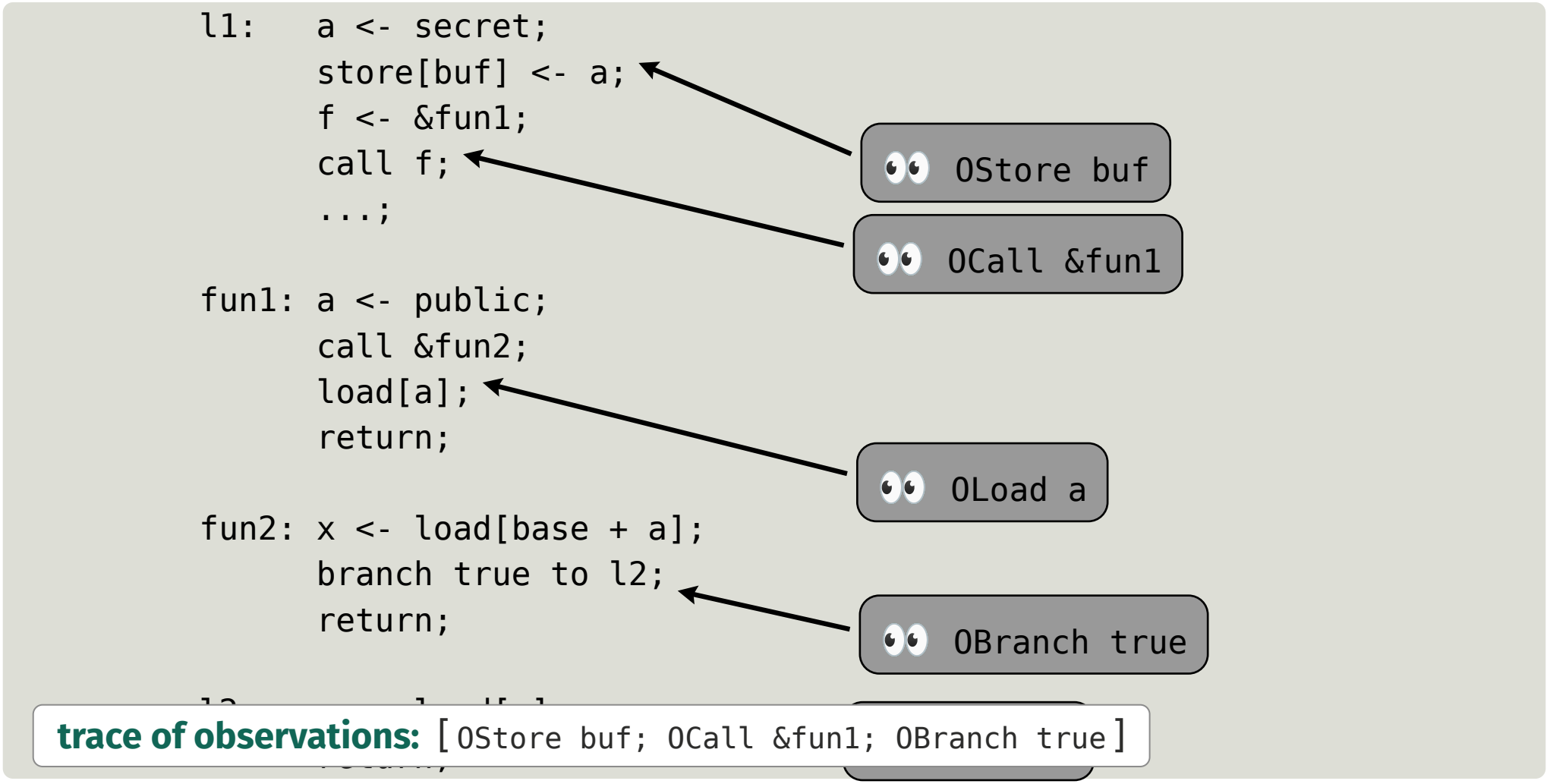
fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;
```

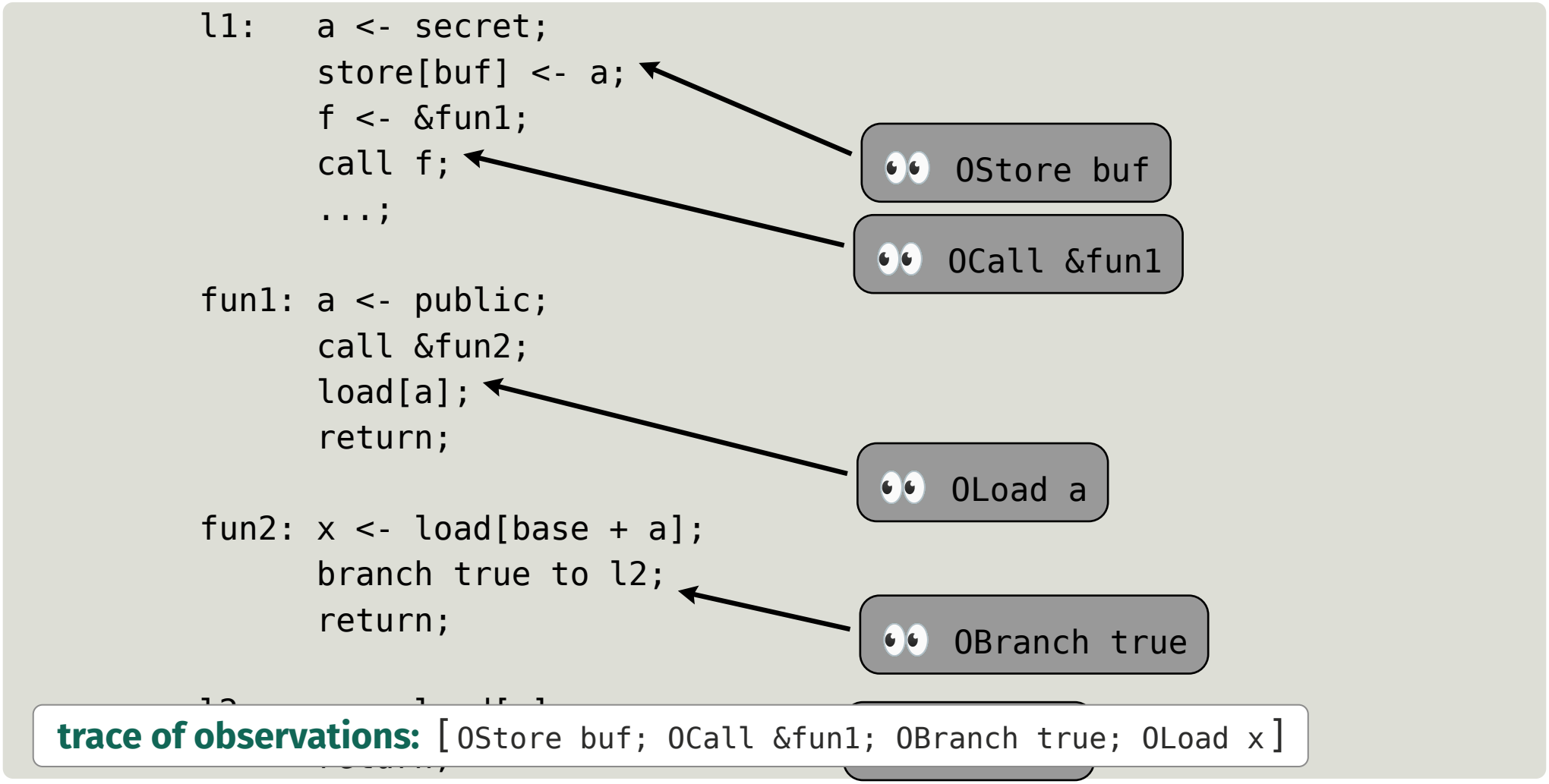


trace of observations: `[OStore buf; OCall &fun1]`

Formally Defined Attacker: Observation



Formally Defined Attacker: Observation



Formally Defined Attacker: Observation


```
l1:   a <- secret;
      store[buf] <- a;
      f <- &fun1;
      call f;
      ...;

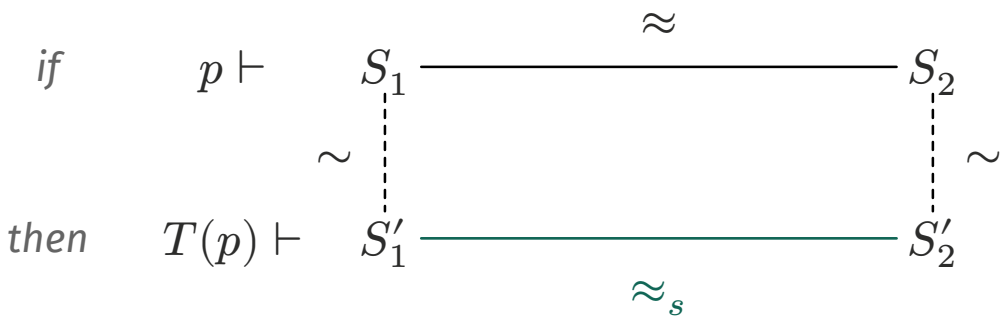
fun1: a <- public;
      call &fun2;
      load[a];
      return;

fun2: x <- load[base + a];
      branch true to l2;
      return;
```



trace of observations: `[OStore buf; OCall &fun1; OBranch true; OLoad x; OLoad a]`

Relative Security: Formal Statement



$\approx$: indistinguishable to a **sequential** attacker $\quad$ $\approx_s$: to a **speculative** attacker

Masking-Induced Undefined Behavior: Example

source

```
store[i] <- &g
n <- load[j]
b <- n <= 42
branch b to l
```

hardened

```
store[msf ? 0 : i] <- &g
n <- load[msf ? 0 : j]
b <- n <= 42
branch (msf ? 0 : b) to l'
```

- sequentially: `j` holds an integer — the comparison `n <= 42` is **defined**
- misspeculating (`msf = 1`): store and load both use **address 0** — `n` becomes `&g`, and `&g <= 42` has no defined result → **undefined value**
- `b` is **masked** at its use — the undefined value is **never used**

Bibliography

Barthe, Gilles, Sunjay Cauligi, Benjamin Grégoire, et al. 2021. “High-Assurance Cryptography in the Spectre Era.” In “42nd IEEE Symposium on Security and Privacy, SP.” Special issue, *42nd IEEE Symposium on Security and Privacy, SP*, 1884–901. https://doi.org/10.1109/SP40001.2021.00046.

Baumann, Jonathan, Roberto Blanco, Léon Ducruet, Sebastian Harwig, and Catalin Hritcu. 2025. “FSLH: Flexible Mechanized Speculative Load Hardening.” In “38th IEEE Computer Security Foundations Symposium, CSF 2025, Santa Cruz, CA, USA, June 16-20, 2025.” Special issue, *38th IEEE Computer Security Foundations Symposium, CSF 2025, Santa Cruz, CA, USA, June 16-20, 2025*, 569–84. https://doi.org/10.1109/CSF64896.2025.00023.