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;

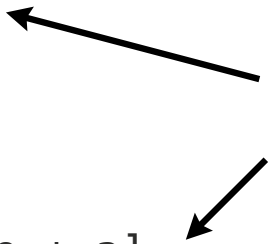
fun2:
      x <- load[base + a];
      y <- load[x];
      return;
```

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;
```

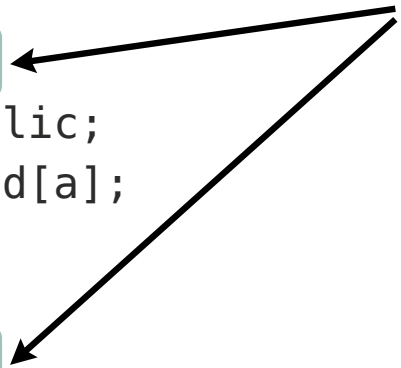
Hardware extension present in **recent Intel processors** as part of **CET**

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**

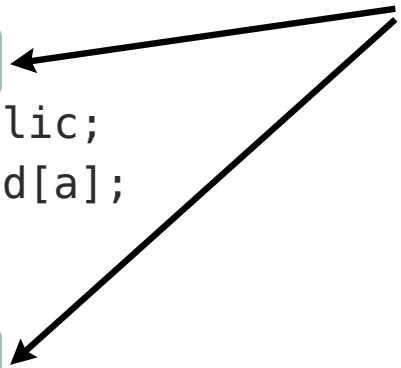
- special `ctarget` instruction marks **valid callable locations**

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**

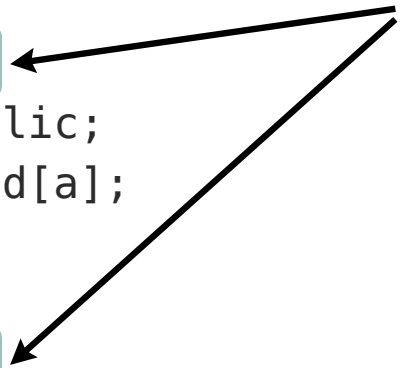
- special `ctarget` instruction marks **valid callable locations**
- hardware enforces that an indirect call must be followed by a `ctarget`

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**

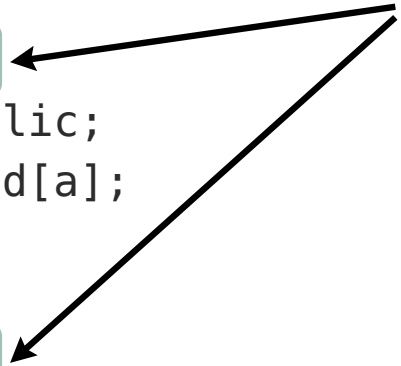
- special `ctarget` instruction marks **valid callable locations**
- hardware enforces that an indirect call must be followed by a `ctarget`
  - otherwise stops speculative execution

Hardware Defense: Indirect Branch Tracking (IBT)


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

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

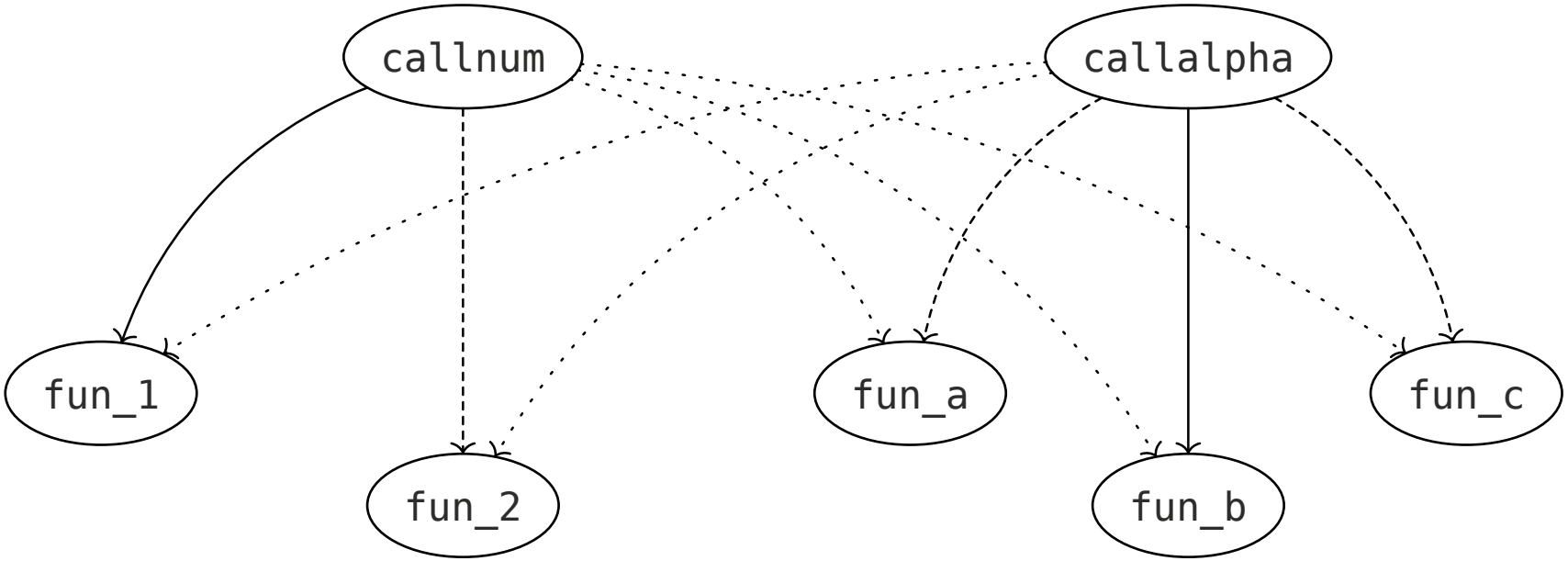
fun2: ctarget;
      x <- load[base + a];
      y <- load[x];
      return;
```



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

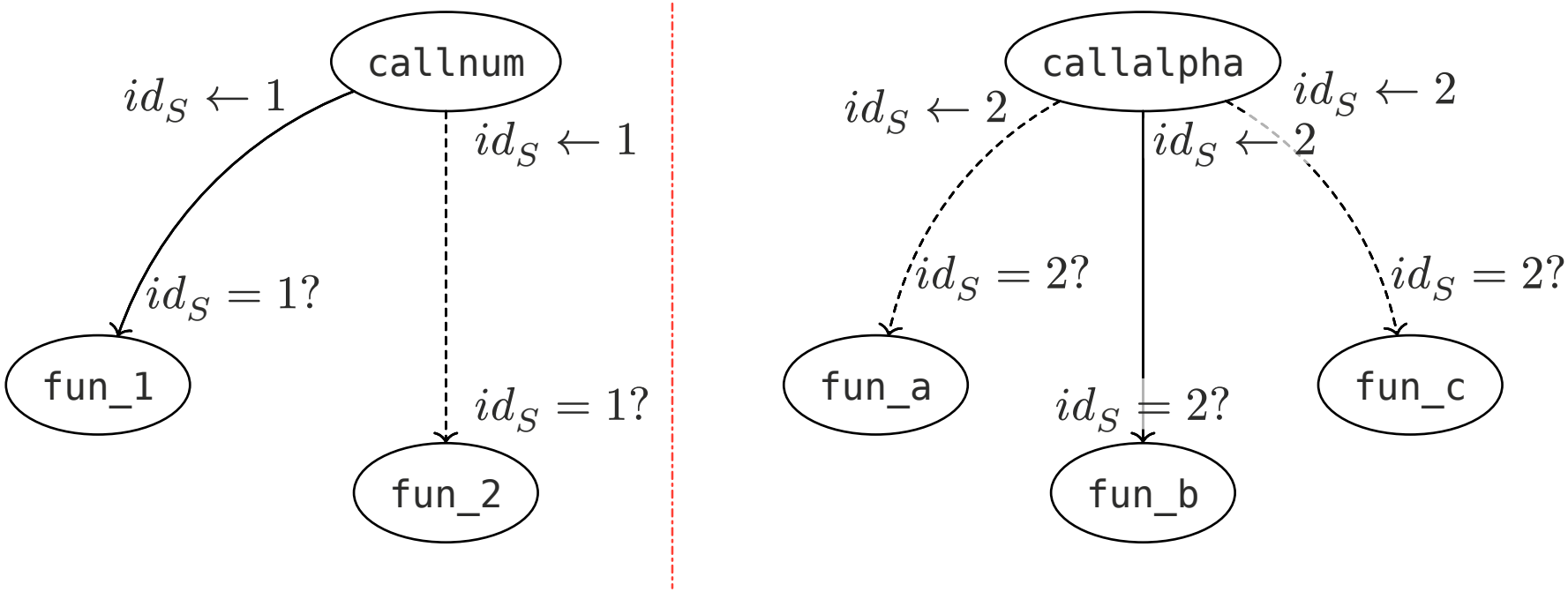
- special `ctarget` instruction marks **valid callable locations**
- hardware enforces that an indirect call must be followed by a `ctarget`
  - otherwise stops speculative execution
- restricts attack surface, but **no complete mitigation**!

IBT Granularity



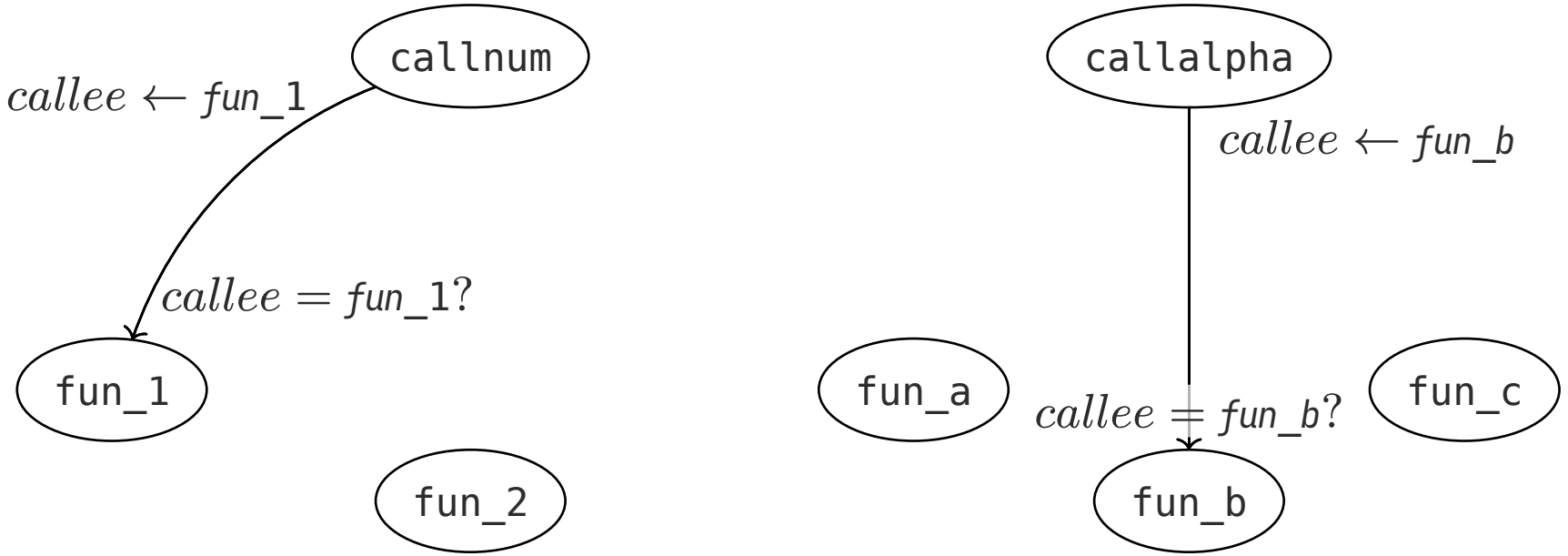
Hardware IBT: all calls can target all functions

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!

Again, possible target locations must be restricted

Securing Returns

Return instructions also allow for complex control flow

$\Longrightarrow$ and similar misspeculation!

Again, possible target locations must be restricted

$\Longrightarrow$ IBT **does not** apply to returns

Securing Returns

Return instructions also allow for complex control flow
⟹ and similar misspeculation!

Again, possible target locations must be restricted
⟹ IBT **does not** apply to returns

Target hardware configured s.t. return predictions
only depend on the **current process**

Securing Returns

Return instructions also allow for complex control flow
⟹ and similar misspeculation!

Again, possible target locations must be restricted
⟹ IBT **does not** apply to returns

Target hardware configured s.t. return predictions
only depend on the **current process**
⟹ Predicted locations **must** correspond to a **valid call site** in the program

Securing Returns

Return instructions also allow for complex control flow
⟹ and similar misspeculation!

Again, possible target locations must be restricted
⟹ IBT **does not** apply to returns

Target hardware configured s.t. return predictions
only depend on the **current process**
⟹ Predicted locations **must** correspond to a **valid call site** in the program
⟹ Sufficient to instrument after every `call` instruction

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;
      ...

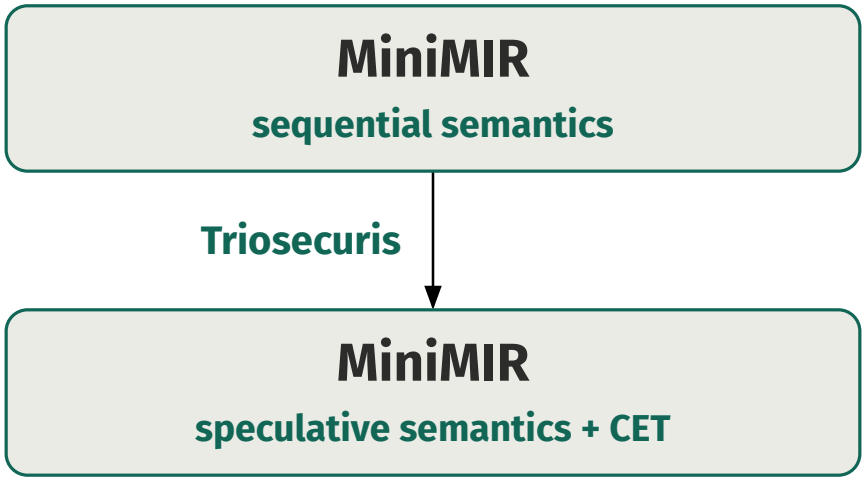
fun1: ctarget;
      msf <- callee = &fun1 ? msf : 1;
      a <- public;
      b <- load[msf ? 0 : a];
      callee <- peek;
      return;
```

Securing Returns


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

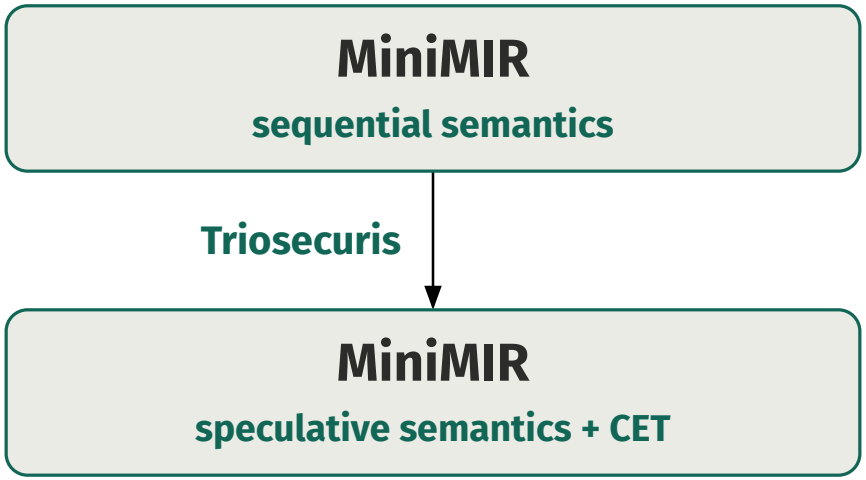
fun1: ctarget;
      msf <- callee = &fun1 ? msf : 1;
      a <- public;
      b <- load[msf ? 0 : a];
      callee <- peek;
      return;
```

Formal Model



Triosecuris is a **compilation pass**

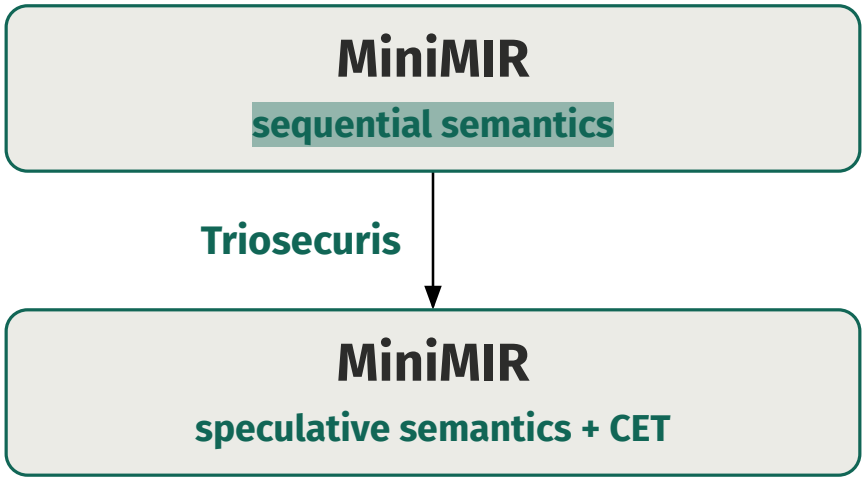
Formal Model



Triosecuris is a **compilation pass**

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

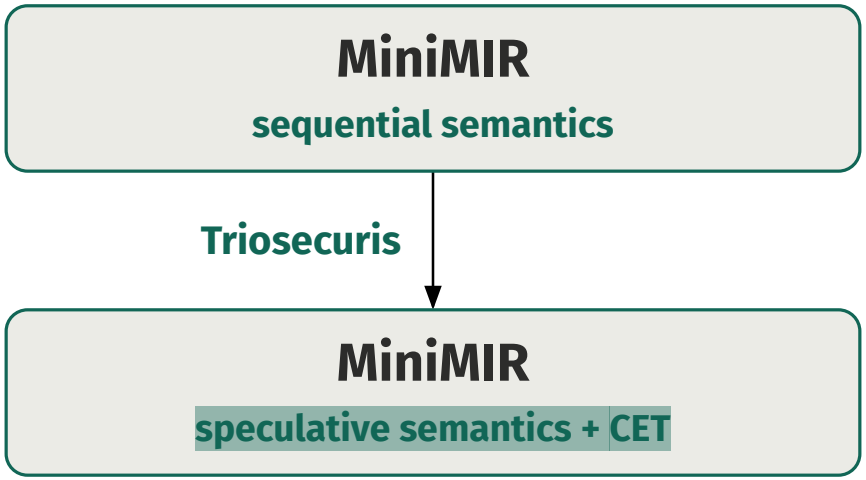
Formal Model



Triosecuris is a **compilation pass**

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

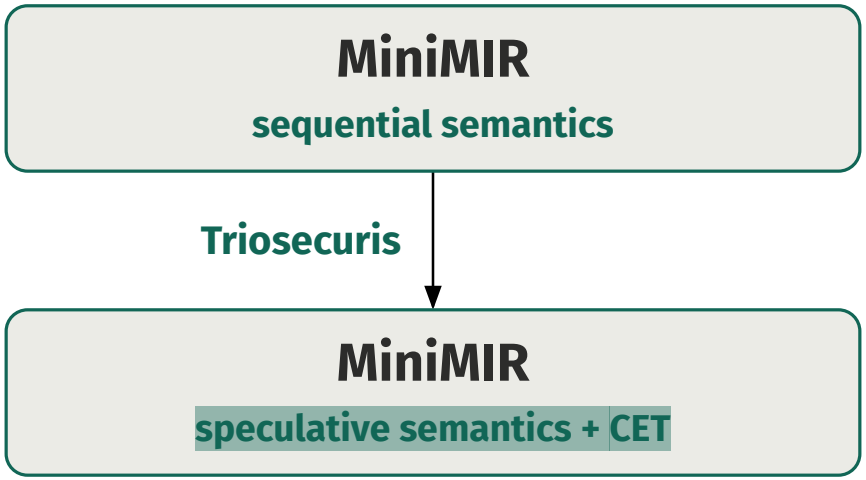
Formal Model



Triosecuris is a **compilation pass**

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

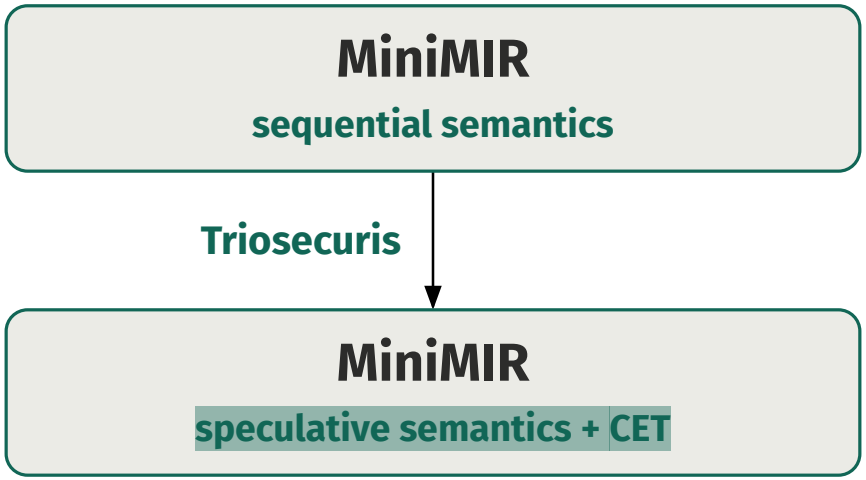
Formal Model



Triosecuris is a **compilation pass**

- MiniMIR: an **intermediate language** with **basic blocks** (like LLVM MIR) and abstract **code pointers**
- speculation is **over-approximated** and **unlimited**:

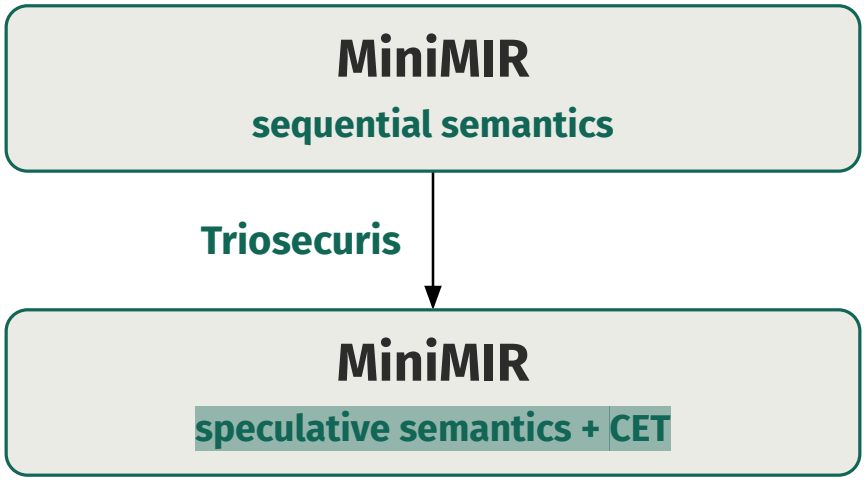
Formal Model



Triosecuris is a **compilation pass**

- MiniMIR: an **intermediate language** with **basic blocks** (like LLVM MIR) and abstract **code pointers**
- speculation is **over-approximated** and **unlimited**:
  - attacker controls outcomes of branches, calls and Returns

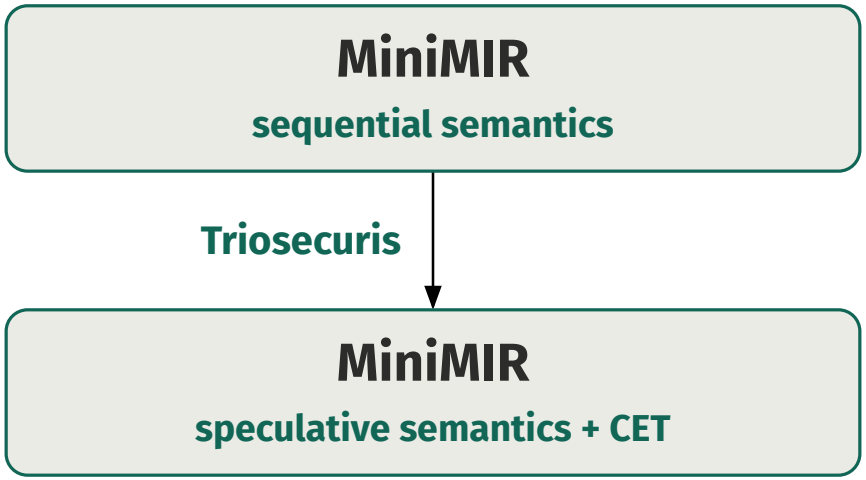
Formal Model



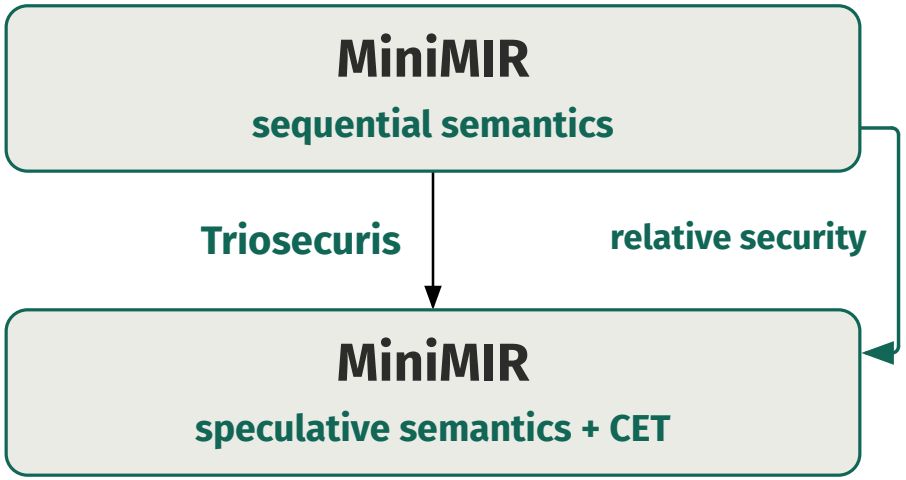
Triosecuris is a **compilation pass**

- MiniMIR: an **intermediate language** with **basic blocks** (like LLVM MIR) and abstract **code pointers**
- speculation is **over-approximated** and **unlimited**:
  - attacker controls outcomes of branches, calls and Returns
  - no fixed speculation window

Formal Guarantees

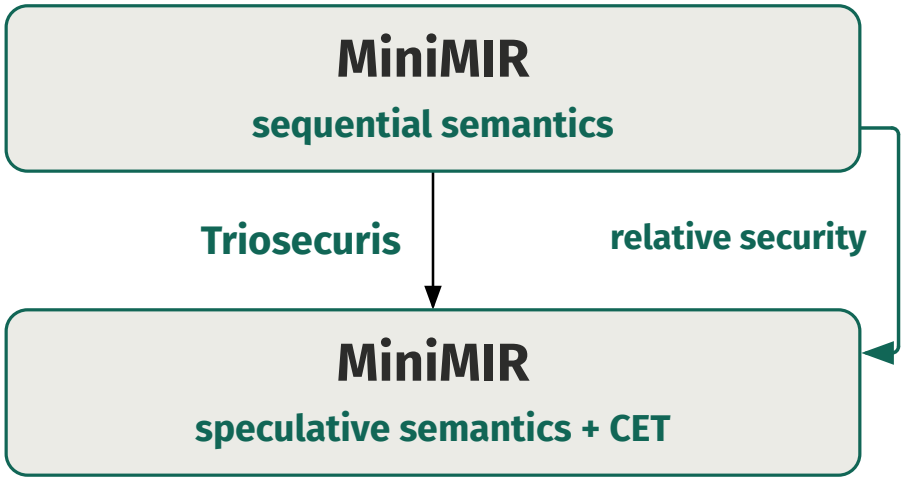


Formal Guarantees



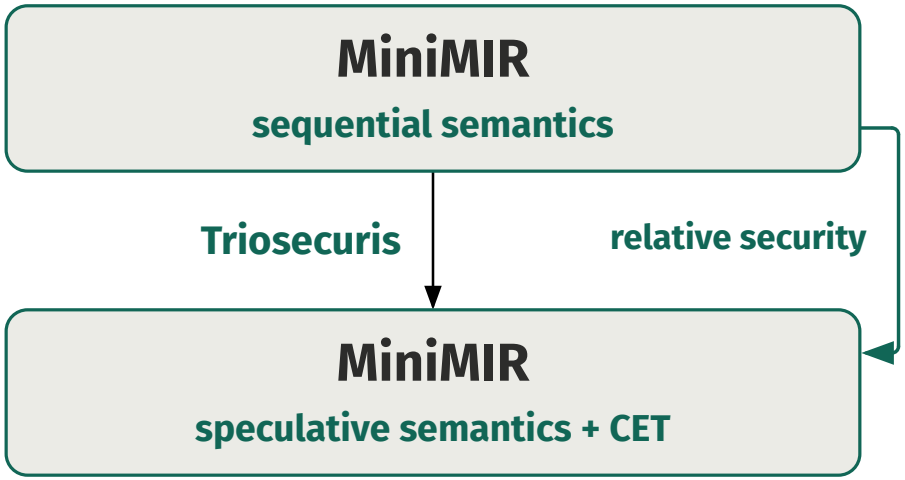
We prove **relative security**:

Formal Guarantees



We prove **relative security**:
The hardened program under **speculation** **leaks no more** than the **source program** does **sequentially**

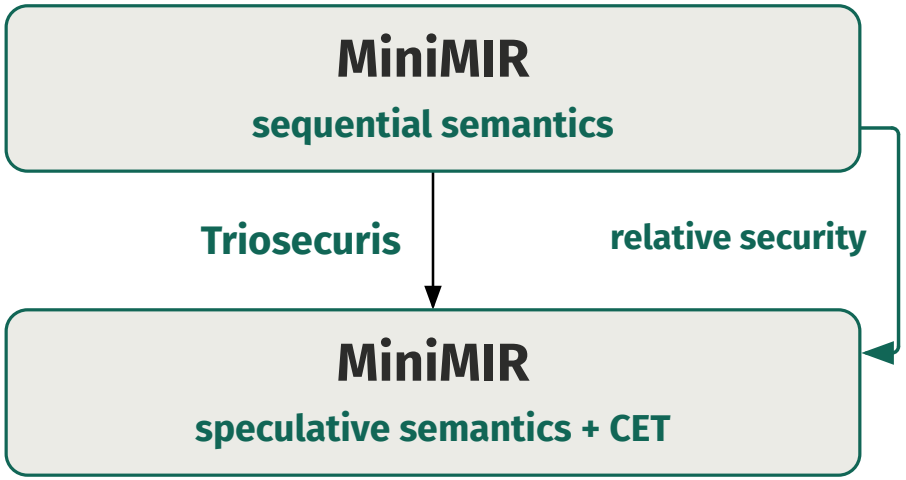
Formal Guarantees



We prove **relative security**:
The hardened program under **speculation** **leaks no more** than the **source program** does **sequentially**

- **sequential** indistinguishability of **source** $\Rightarrow$ **speculative** indistinguishability of **target**

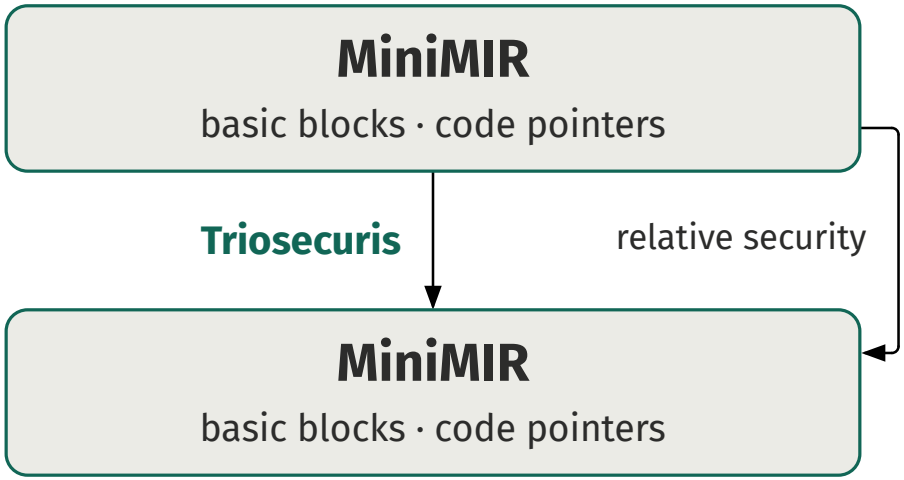
Formal Guarantees



We prove **relative security**:
The hardened program under **speculation leaks no more** than the **source program** does **sequentially**

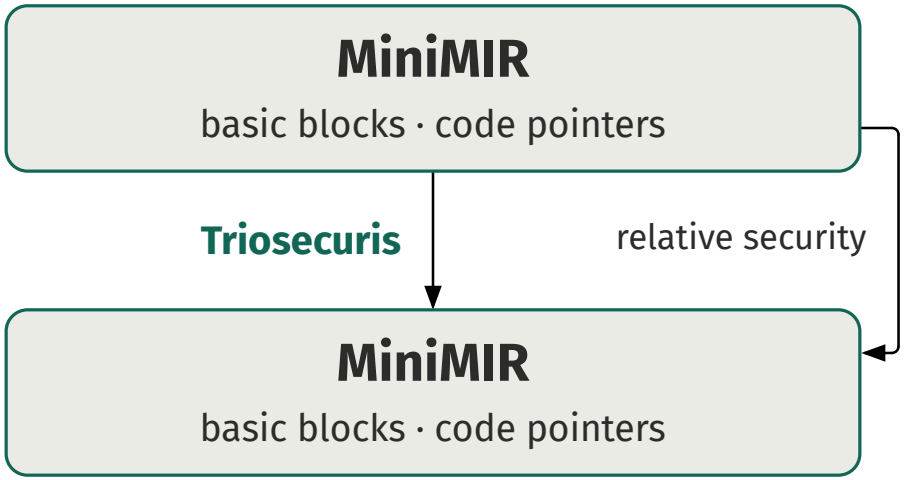
- **sequential** indistinguishability of **source** $\Rightarrow$ **speculative** indistinguishability of **target**
- holds for **arbitrary programs** — even **sequentially insecure** ones

Toward a Lower Level



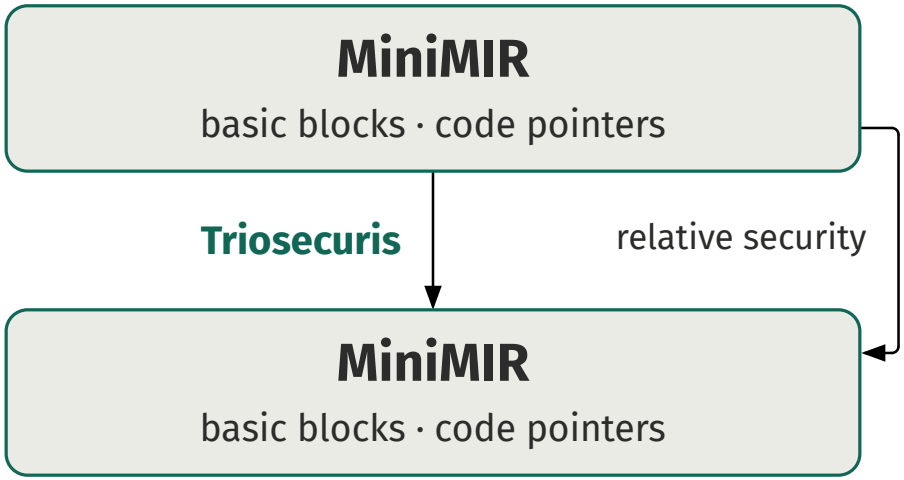
- MiniMIR is an intermediate language, not machine code

Toward a Lower Level



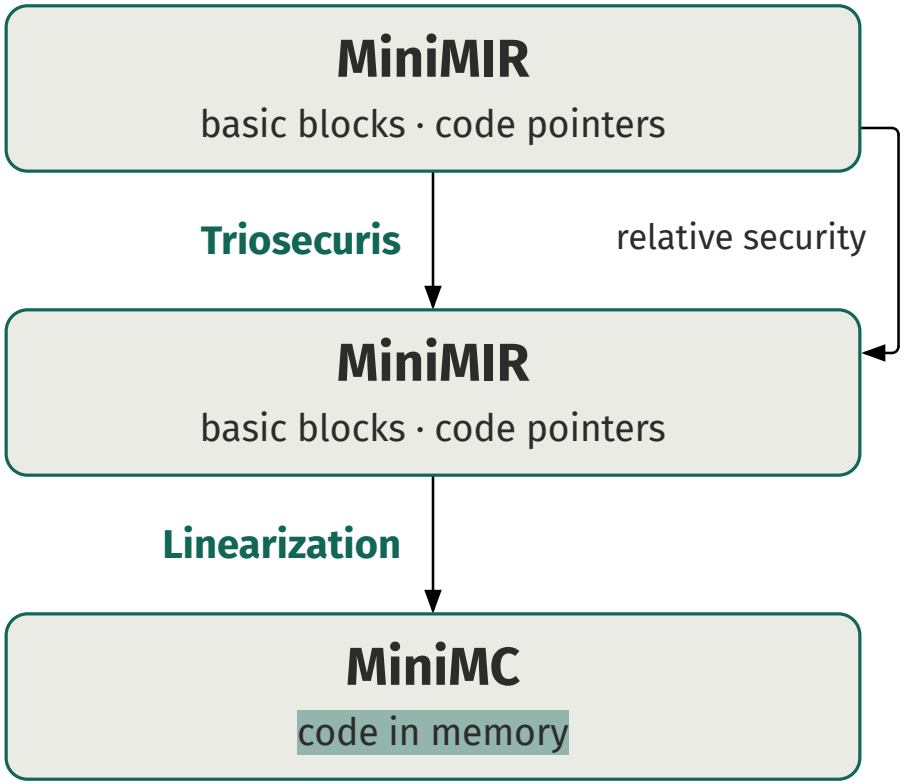
- MiniMIR is an intermediate language, not machine code
  - Security guarantees must be preserved by further compilation

Toward a Lower Level



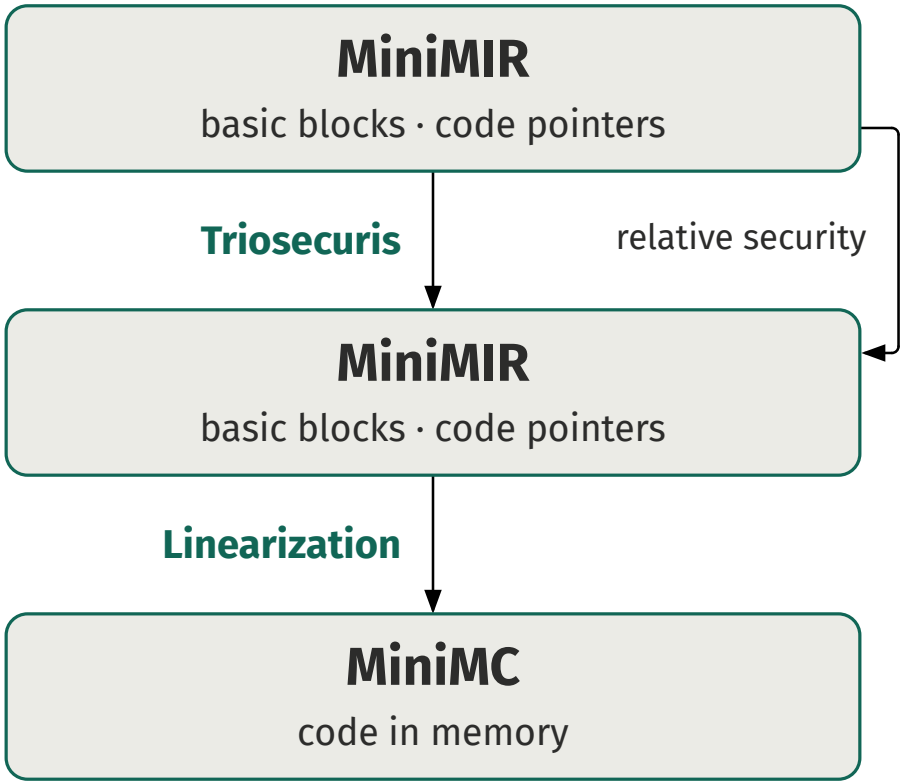
- MiniMIR is an intermediate language, not machine code
  - Security guarantees must be preserved by further compilation
  - MiniMir has **undefined behaviour**!

Toward a Lower Level



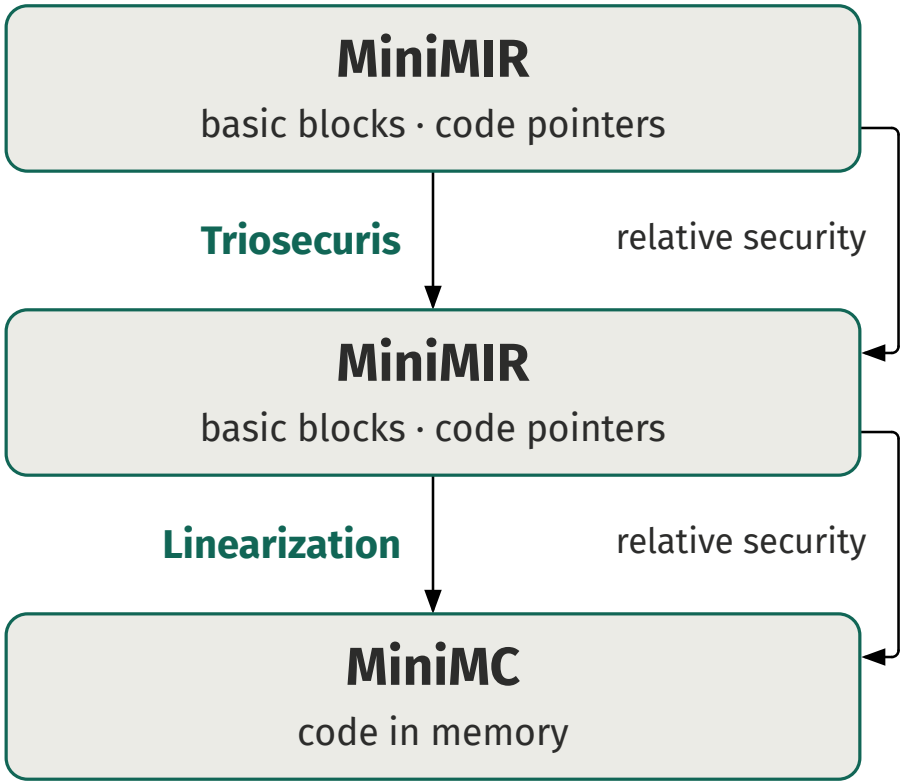
- MiniMIR is an intermediate language, not machine code
  - Security guarantees must be preserved by further compilation
  - MiniMir has **undefined behaviour**!
- We demonstrate this with **linearization**: code in memory, code pointers become **integers**

Toward a Lower Level



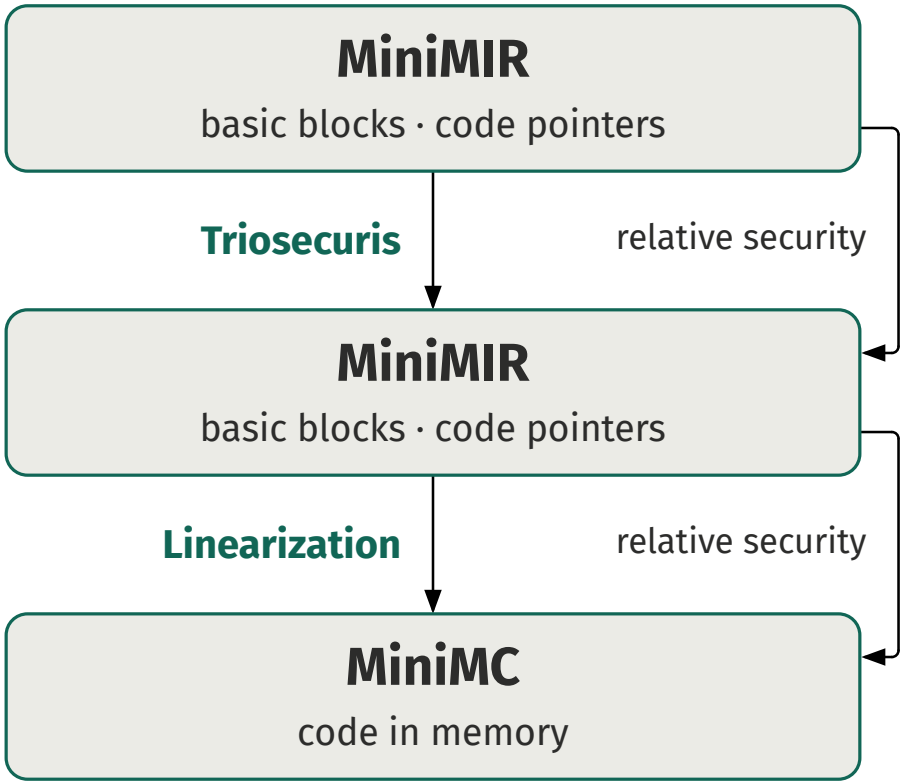
- MiniMIR is an intermediate language, not machine code
  - Security guarantees must be preserved by further compilation
  - MiniMir has **undefined behaviour**!
- We demonstrate this with **linearization**: code in memory, code pointers become **integers**
  - $\Longrightarrow$ **refines undefined behavior (UB)**

Toward a Lower Level



- MiniMIR is an intermediate language, not machine code
  - Security guarantees must be preserved by further compilation
  - MiniMir has **undefined behaviour**!
- We demonstrate this with **linearization**: code in memory, code pointers become **integers**
  - $\Longrightarrow$ **refines undefined behavior (UB)**
  - preserves security if its input is **safe** (free of UB)

Toward a Lower Level



- MiniMIR is an intermediate language, not machine code
  - Security guarantees must be preserved by further compilation
  - MiniMir has **undefined behaviour**!
- We demonstrate this with **linearization**: code in memory, code pointers become **integers**
  - $\Longrightarrow$ **refines undefined behavior (UB)**
  - preserves security if its input is **safe** (free of UB)
    - including during speculative execution!

Safety Preservation

Linearization needs the **hardened program** to be **safe, including during speculation**:

Safety Preservation

Linearization needs the **hardened program** to be **safe, including during speculation**:

- but **masking** introduces **ill-typed operations**!

Safety Preservation

Linearization needs the **hardened program** to be **safe, including during speculation**:

- but **masking** introduces **ill-typed operations**!
  turning `42 <= load[a]` into `42 <= load[0]`

Safety Preservation

Linearization needs the **hardened program** to be **safe, including during speculation**:

- but **masking** introduces **ill-typed operations**!
  turning `42 <= load[a]` into `42 <= load[0]`
  `42 <= 100` `42 <= &fun1`

Safety Preservation

Linearization needs the **hardened program** to be **safe, including during speculation**:

- but **masking** introduces **ill-typed operations**!
  turning `42 <= load[a]` into `42 <= load[0]`
  `42 <= 100` `42 <= &fun1`
- **postpone** with **undefined values** — Undefined Behavior only **if used**

Safety Preservation

Linearization needs the **hardened program** to be **safe, including during speculation**:

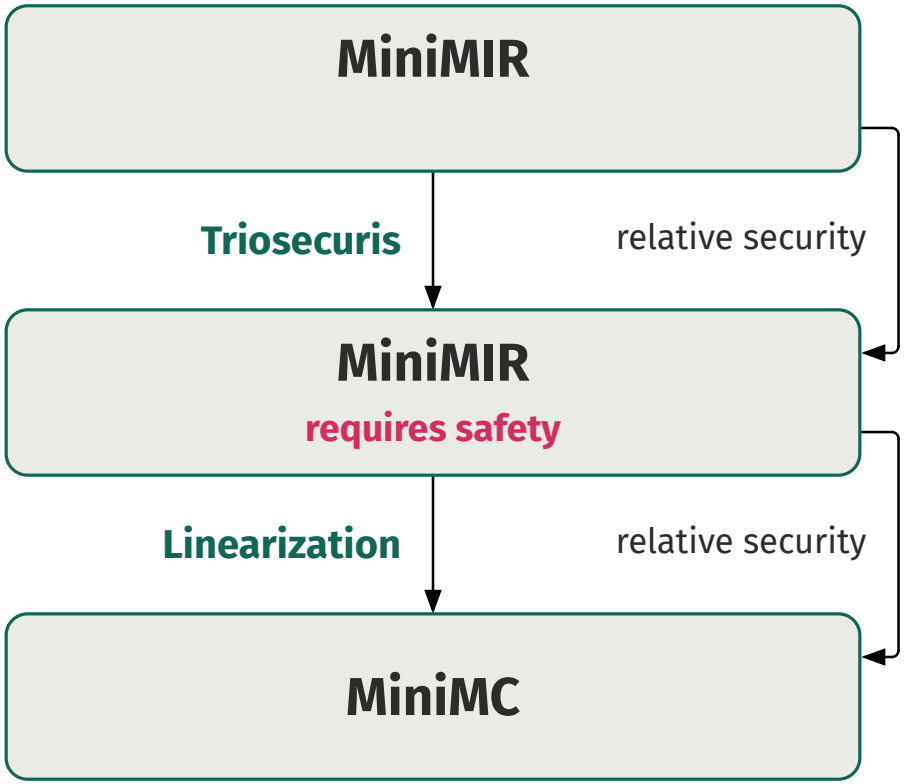
- but **masking** introduces **ill-typed operations**!
  turning `42 <= load[a]` into `42 <= load[0]`
  `42 <= 100` `42 <= &fun1`
- **postpone** with **undefined values** — Undefined Behavior only **if used**
- its use is **prevented** by **masking** — the error **never happens**

Safety Preservation

Linearization needs the **hardened program** to be **safe, including during speculation**:

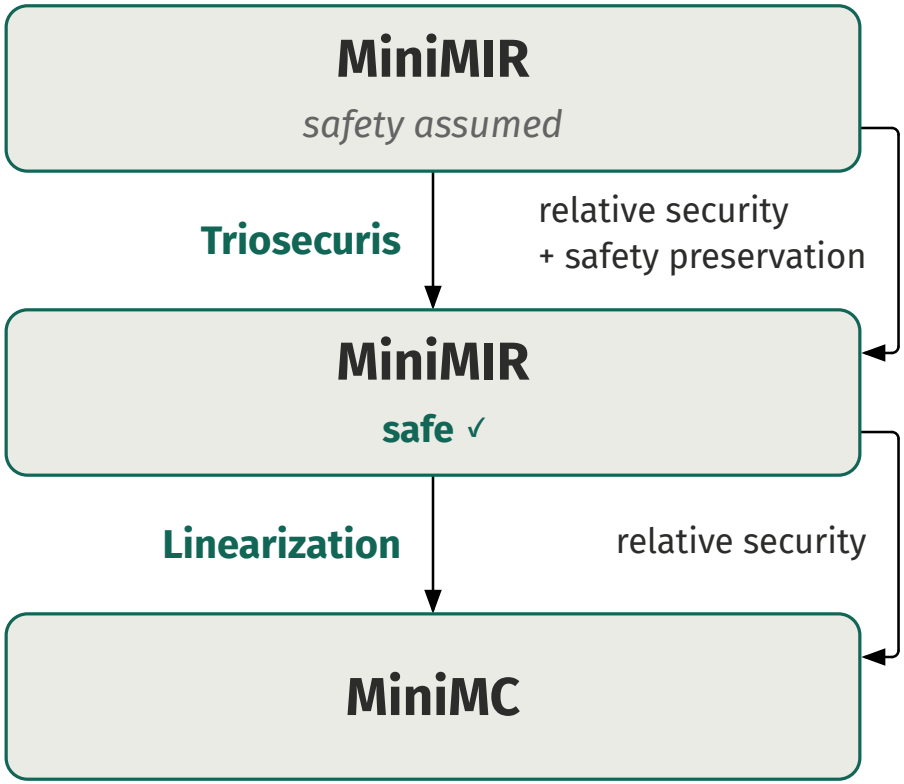
- but **masking** introduces **ill-typed operations**!
  turning `42 <= load[a]` into `42 <= load[0]`
  `42 <= 100` `42 <= &fun1`
- **postpone** with **undefined values** — Undefined Behavior only **if used**
- its use is **prevented** by **masking** — the error **never happens**
- ⟹ **safety preservation**: if the source is **sequentially safe**, the hardened program is **safe under speculation**

End-to-End Guarantees



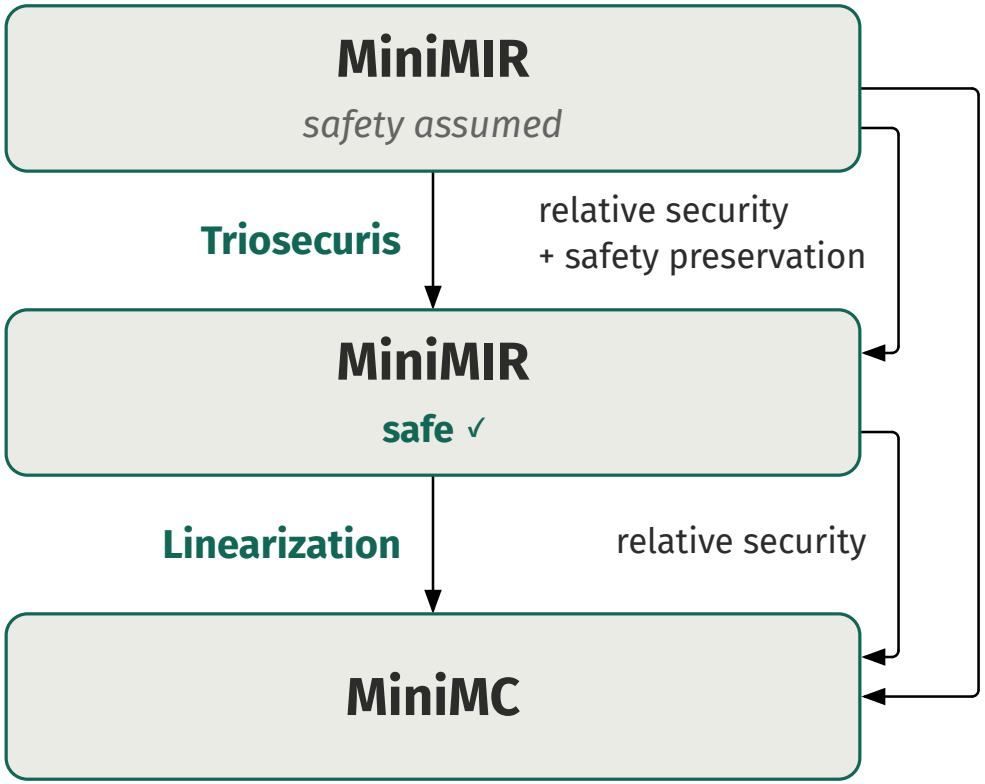
- We have **relative security** of **Triosecuris** and **Linearization**

End-to-End Guarantees



- We have **relative security** of **Triosecuris** and **Linearization**
- We have (speculative) **safety preservation** of **Triosecuris**

End-to-End Guarantees



- We have **relative security** of **Triosecuris** and **Linearization**
- We have (speculative) **safety preservation** of **Triosecuris**
- ⟹ Compose for **end-to-end relative security**

Prototype Implementation in **LLVM**

Triosecuris in LLVM

Prototype Implementation in **LLVM**

- built on Zhang et al.‘s Ultimate SLH implementation

Triosecuris in LLVM

Prototype Implementation in **LLVM**

- built on Zhang et al.‘s Ultimate SLH implementation
- currently waiting for benchmarking

Triosecuris in LLVM

Prototype Implementation in **LLVM**

- built on Zhang et al.‘s Ultimate SLH implementation
- currently waiting for benchmarking
- based on preliminary testing:

Prototype Implementation in **LLVM**

- built on Zhang et al.'s Ultimate SLH implementation
- currently waiting for benchmarking
- based on preliminary testing:
  - should **outperform USLH + retpoline**

Prototype Implementation in **LLVM**

- built on Zhang et al.'s Ultimate SLH implementation
- currently waiting for benchmarking
- based on preliminary testing:
  - should **outperform USLH + retpoline**
    - (retpoline is enabled automatically with SLH)

Prototype Implementation in **LLVM**

- built on Zhang et al.'s Ultimate SLH implementation
- currently waiting for benchmarking
- based on preliminary testing:
  - should **outperform USLH + retpoline**
    - (retpoline is enabled automatically with SLH)
  - **may** outperform **plain SLH + retpoline**

Closing the Distance

Several **large differences** between our proof and LLVM implementation:

Closing the Distance

Several **large differences** between our proof and LLVM implementation:

- Is security preserved by **register allocation**?

Closing the Distance

Several **large differences** between our proof and LLVM implementation:

- Is security preserved by **register allocation**?
  - **not true** in the general case! (van der Wall and Meyer 2025)

Closing the Distance

Several **large differences** between our proof and LLVM implementation:

- Is security preserved by **register allocation**?
  - **not true** in the general case! (van der Wall and Meyer 2025)
- Data stored on **stack**

Closing the Distance

Several **large differences** between our proof and LLVM implementation:

- Is security preserved by **register allocation**?
  - **not true** in the general case! (van der Wall and Meyer 2025)
- Data stored on **stack**
- **Naive, unoptimized implementation**

Closing the Distance

Several **large differences** between our proof and LLVM implementation:

- Is security preserved by **register allocation**?
  - **not true** in the general case! (van der Wall and Meyer 2025)
- Data stored on **stack**
- **Naive, unoptimized implementation**
  - (**always masking** vs. **only masking values once**)

Closing the Distance

Several **large differences** between our proof and LLVM implementation:

- Is security preserved by **register allocation**?
  - **not true** in the general case! (van der Wall and Meyer 2025)
- Data stored on **stack**
- **Naive, unoptimized implementation**
  - (**always masking** vs. **only masking values once**)

$\Longrightarrow$ We are working to address all of these

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

Conclusion

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

Bibliography

Wall, Sören van der, and Roland Meyer. 2025. “SNIP: Speculative Execution and Non-Interference Preservation for Compiler Transformations.” *Proc. ACM Program. Lang.* 9 (POPL): 1506–35. https://doi.org/10.1145/3704887.

Zhang, Zhiyuan, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. 2023. “Ultimate SLH: Taking Speculative Load Hardening to the Next Level.” In *32nd USENIX Security Symposium*, edited by Joseph A. Calandrino and Carmela Troncoso, *32nd USENIX Security Symposium*. USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-slh.