# Cyber in Saclay – Student Session

## Efficient Relational Symbolic Execution for Spectre with Haunted RelSE

February, 8th 2021

Lesly-Ann Daniel
CEA, LIST, Université Paris-Saclay
France

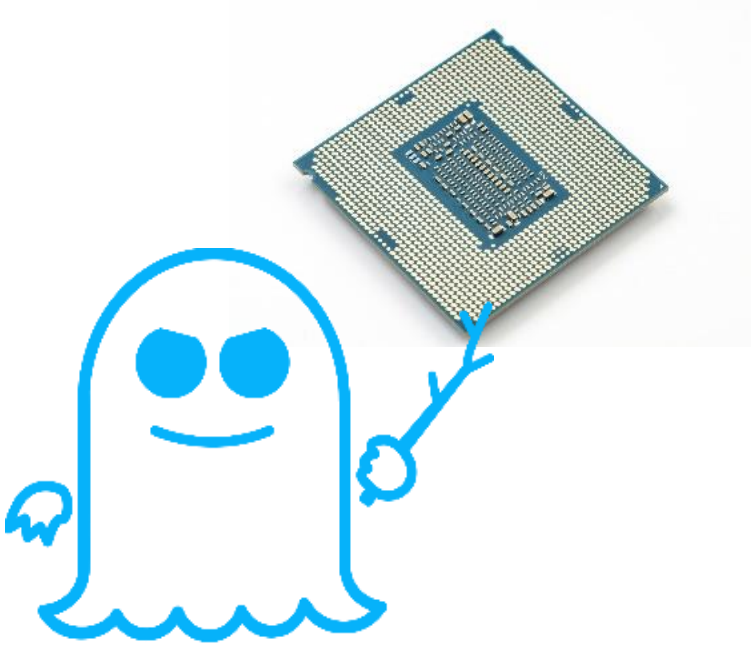
Sébastien Bardin
CEA, LIST, Université Paris-Saclay
France

Tamara Rezk
Inria
France

# Spectre haunting our code

## Spectre attacks (2018)

- Exploit speculative execution in processors
- Affect almost all processors
- Attackers can force mispeculations: transient executions
- Transient executions are reverted at architectural level
- But *not the microarchitectural state* (e.g. cache)



***Idea.*** *Force victim to encode secret data in cache during transient execution & recover them with cache attacks*

# Spectre-PHT

## Spectre-PHT

Exploits conditional branch predictor

```
if idx < size {
      v = tab[idx]
      leak(v)
}
```

- `idx` is attacker controlled
- content of `tab` is public
- `leak(v)` encodes `v` to cache

## Regular execution

- Conditional bound check ensures `idx` is in bounds
- `v` contains public data

# Spectre-PHT

## Spectre-PHT

Exploits conditional branch predictor

```
if idx < size {
      v = tab[idx]
      leak(v)
}
```

- `idx` is attacker controlled
- content of `tab` is public
- `leak(v)` encodes `v` to cache

## Regular execution

- Conditional bound check ensures `idx` is in bounds
- `v` contains public data

## Transient Execution

- Conditional is misspeculated
- Out-of-bound array access
  → load secret data in `v`
- `v` is leaked to the cache

# Spectre-STL

**Spectre-STL:** Loads can speculatively bypass prior stores

## Regular execution

```
store a s
store a p
store b q
v = load a
leak(v)
```

```
leak(p)
```

- where s is secret, p and q are public
- where a ≠ b
- leak(v) encodes v to cache

# Spectre-STL

**Spectre-STL:** Loads can speculatively bypass prior stores

**Regular execution** + **Transient Executions**

```
store a s
store a p
store b q
v = load a
leak(v)
```

leak(p)

+

```
store a s
store a p
v = load a
store b q
leak(v)
```

leak(p)

- where s is secret, p and q are public
- where a ≠ b
- leak(v) encodes v to cache

# Spectre-STL

**Spectre-STL:** Loads can speculatively bypass prior stores

**Regular execution** + **Transient Executions**

```
store a s
store a p
store b q
v = load a
leak(v)
```

leak(p)

+

```
store a s
store a p
v = load a
store b q
leak(v)
```

leak(p)

+

```
store a s
v = load a
store a p
store b q
leak(v)
```

leak(s)

- where s is secret, p and q are public
- where a ≠ b
- leak(v) encodes v to cache

# Spectre-STL

**Spectre-STL:** Loads can speculatively bypass prior stores

**Regular execution** + **Transient Executions**

```
store a s
store a p
store b q
v = load a
leak(v)
```

leak(p)

+

```
store a s
store a p
v = load a
store b q
leak(v)
```

leak(p)

+

```
store a s
v = load a
store a p
store b q
leak(v)
```

leak(s)

+

```
v = load a
store a s
store a p
store b q
leak(v)
```

leak(init_mem[a])

- where s is secret, p and q are public
- where a ≠ b
- leak(v) encodes v to cache

# Detect Spectre attacks ? Challenging !

- Counter-intuitive semantics
- Path explosion:
  - **Spectre-STL**: all possible load/store interleavings !
- Needs to hold at binary-level

| | Target | Spectre-PHT | Spectre-STL |
|---|---|---|---|
| **KLEESpectre** | LLVM | ☺ | - |
| **SpecuSym** | LLVM | ☺ | - |
| **FASS** | Binary | ☹ | - |
| **Spectector** | Binary | ☹ | - |
| **Pitchfork** | Binary | 😐 | ☹ |

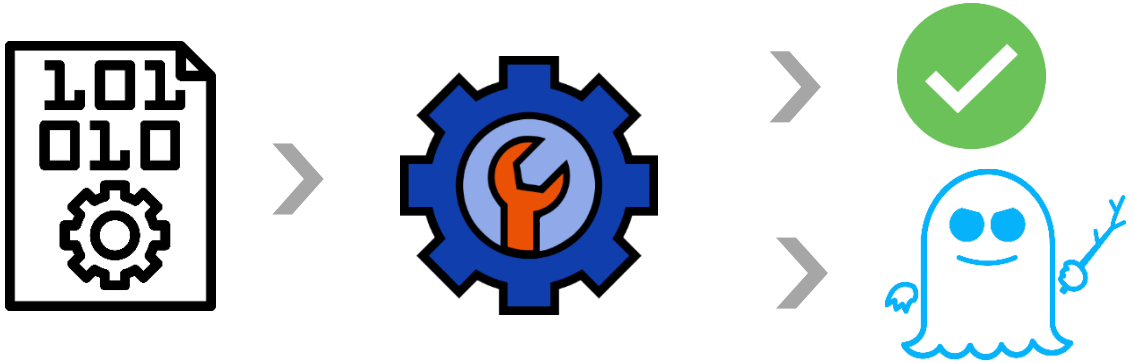
Verification tools for Spectre

Path explosion for Spectre-STL on Litmus tests (**328** instr.)

| Semantics | Paths |
|---|---|
| Regular semantics | **14** |
| Speculative semantics (Spectre-STL) | **37M** |

# Goal: New verification tools for Spectre

**Goal.** We need new verification tools to detect Spectre attacks !



**Proposal.** → *Verify Speculative Constant Time (SCT) property*
→ *Use Relational Symbolic Execution (RelSE)*

**Challenge**. Model new transient behaviors avoiding path explosion

# Contributions

## Haunted RelSE optimization

- Model transient and regular behaviors at the same time
  - **Spectre-PHT**: pruning redundant paths [in the paper]
  - **Spectre-STL**: pruning + encoding to merge paths
- Formal proof: equivalence with explicit exploration [in the paper]

## Binsec/Haunted, binary-level verification tool

- Experimental evaluation on real world crypto (donna, libsodium, OpenSSL)
- Efficient on real-wold crypto for Spectre-PHT 😐 → 🙂
- Efficient on small programs for Spectre-STL ☹ → 😐
- Comparison with SoA: faster & more vulnerabilities found [in the paper]

**New Spectre-STL violations** [in the paper]

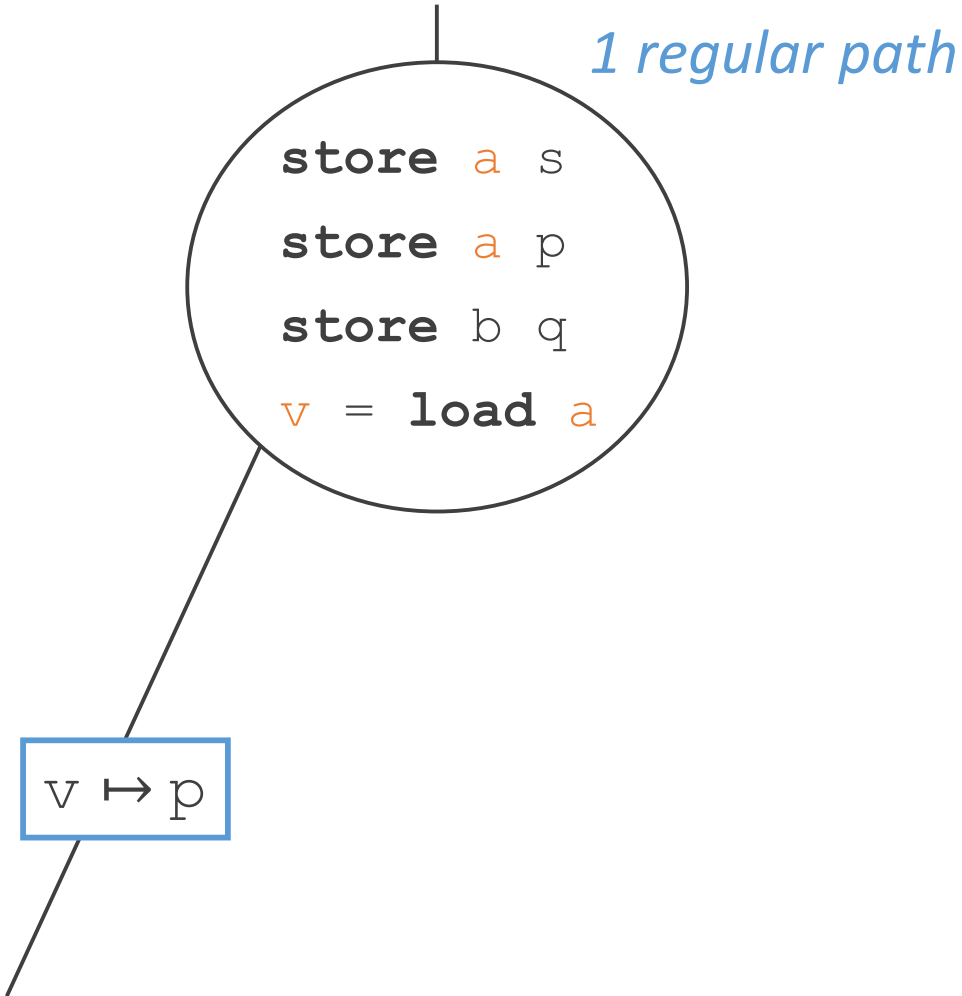
- Index-masking (countermeasure against Spectre-PHT) + proven mitigations
- Code introduced for Position-Independent-Code

# Explicit RelSE for Spectre-STL

**Symbolic execution.** Execute program with symbolic input

```
store a s
store a p
store b q
v = load a
```

where $a \neq b$



# Explicit RelSE for Spectre-STL

**Spectre-STL.** Loads can speculatively bypass prior stores

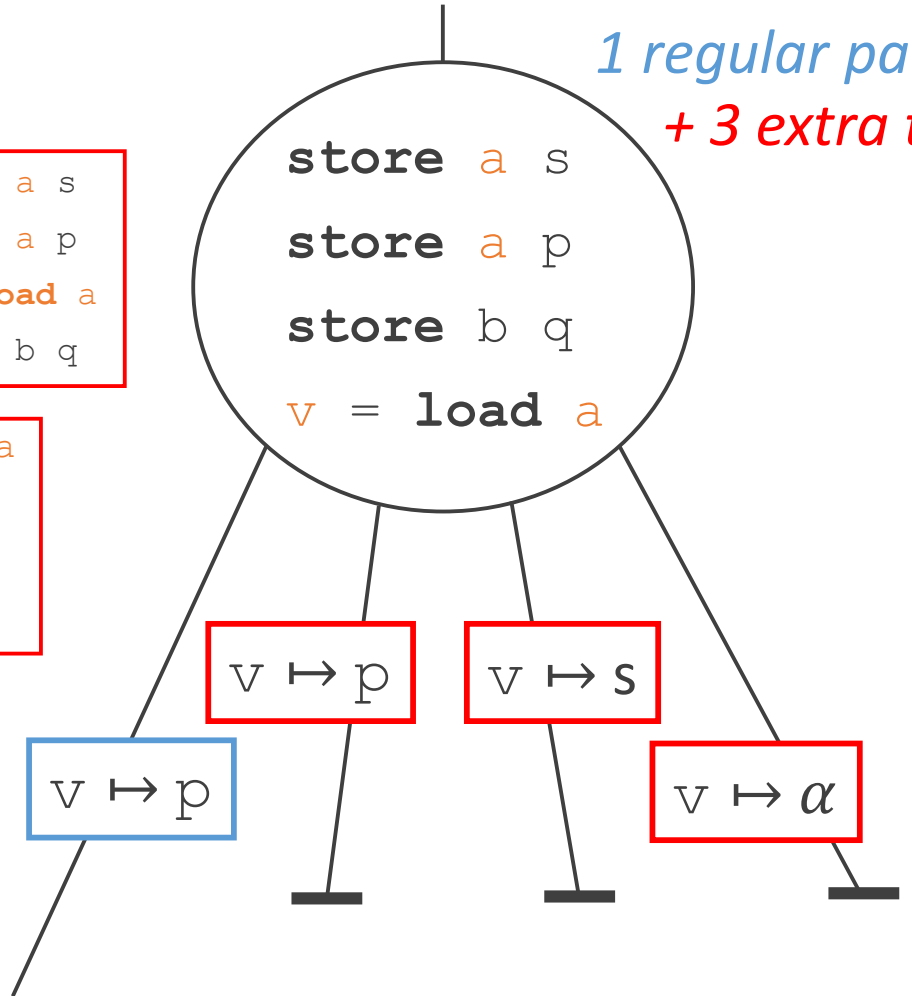
```
store a s
store a p
store b q
v = load a
```

```
store a s
store a p
v = load a
store b q
```

```
store a s
v = load a
store a p
store b q
```

```
v = load a
store a s
store a p
store b q
```

where $a \neq b$




*1 regular path*

*+ 3 extra transient paths*

**Explicit RelSE.**

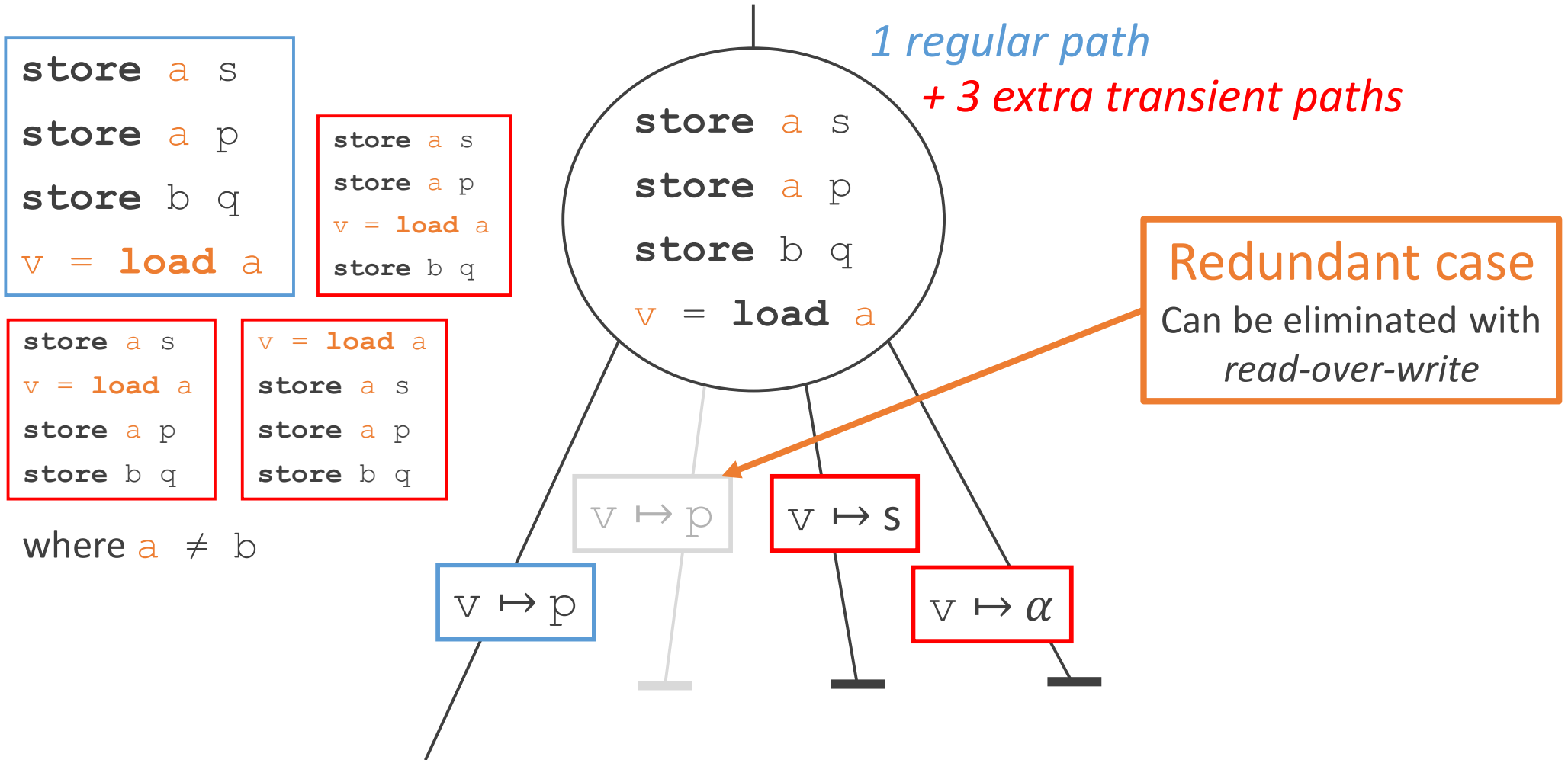
At load instructions: fork execution for each load/store interleaving.

→ Path explosion

(e.g. Pitchfork)

# Explicit RelSE for Spectre-STL

**Spectre-STL.** Loads can speculatively bypass prior stores



# Explicit RelSE for Spectre-STL

**Spectre-STL.** Loads can speculatively bypass prior stores

```
store a s
store a p
store b q
v = load a
```

```
store a s
store a p
v = load a
store b q
```

```
store a s
v = load a
store a p
store b q
```

```
v = load a
store a s
store a p
store b q
```

where a ≠ b

*1 speculative path*

```
store a s
store a p
store b q
v = load a
```

$v \mapsto ite\ \beta_0\ then\ \alpha\ else\ (ite\ \beta_1\ then\ s\ else\ p)$

$\beta_0 = false$

$\beta_1 = false$

**Haunted RelSE.**

- Cut redundant cases
- Encode remaining ones in 1 path
  - symbolic *ite*
  - free booleans $\beta_0$, $\beta_1$

# Experimental evaluation

**Binsec/Haunted.**
Implementation of Haunted RelSE

*More details in paper*


https://github.com/binsec/haunted

## Benchmark.

- **Litmus tests** (46 small test cases)
- Cryptographic primitives **tea** & **donna**
- More complex cryptographic primitives
  - **Libsodium** secretbox
  - **OpenSSL** ssl3-digest-record
  - **OpenSSL** mee-cdc-decrypt

## Experiments.

**RQ1.** Effective on real code ?

→ *Spectre-PHT* ☺ & *Spectre-STL* 😐

**RQ2.** Haunted vs. Explicit ?

→ *Spectre-PHT:* ≈ *or* ↗ & *Spectre-STL: always* ↗

**RQ3.** Comparison against KLEESpectre & Pitchfork

→ *Spectre-PHT:* ≈ *or* ↗ & *Spectre-STL: always* ↗

# Haunted vs. Explicit for Spectre-STL

| | Paths | X86 Ins. | Time | Timeouts | Bugs | Secure | Insecure |
|---|---|---|---|---|---|---|---|
| **Explicit** | **93M** | **2k** | 30h | **15** | **22** | 3/4 | 13/23 |
| **Haunted** | **42** | **17k** | 24h | **8** | **148** | 4/4 | 23/23 |

- Avoids paths explosion
- More unique instruction explored
- Faster

- Less timeouts
- More bugs found
- More programs proven secure / insecure

**Take away, Haunted RelSE vs Explicit RelSE.**

*Always wins ! ↗*

# Conclusion

- Haunted RelSE optimization
  - Model transient and regular behaviors at the same time
  - Significantly improves SoA methods
- Binsec/Haunted, binary-level verification tool
  - Spectre-PHT: efficient on real world crypto 😐 → ☺
  - Spectre-STL: efficient on small programs ☹ → 😐
- New Spectre-STL violations with index masking and PIC

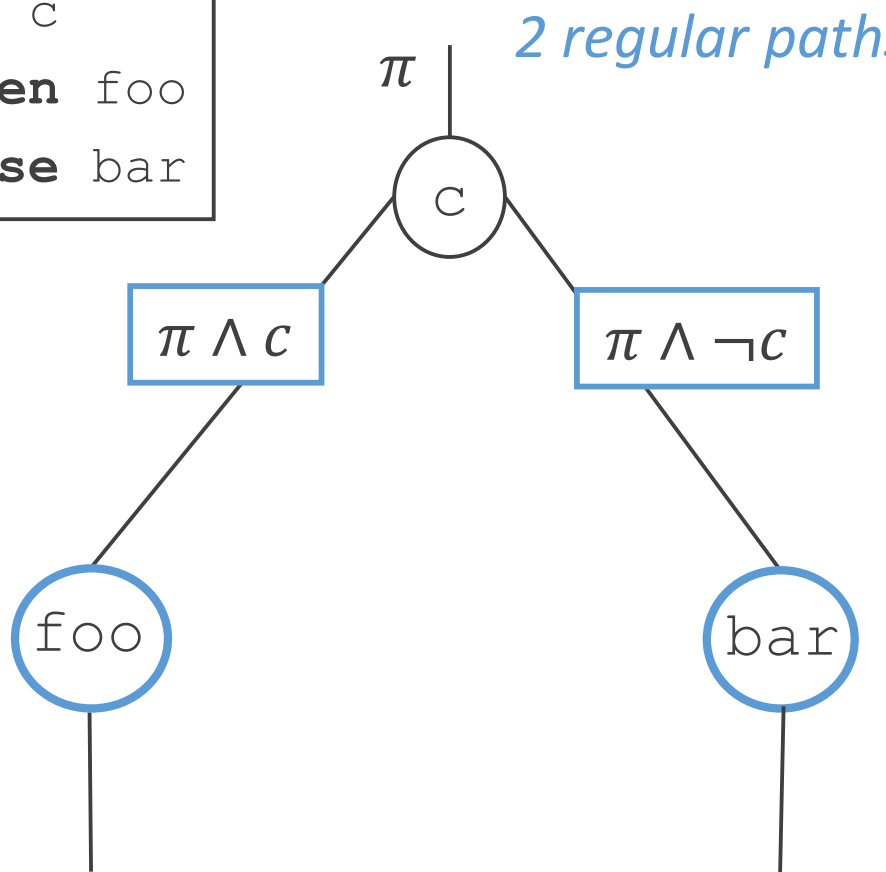
https://github.com/binsec/haunted
https://github.com/binsec/haunted_bench

# Haunted RelSE for Spectre-PHT

# Background: Symbolic Execution

**Symbolic execution.** An illustration.

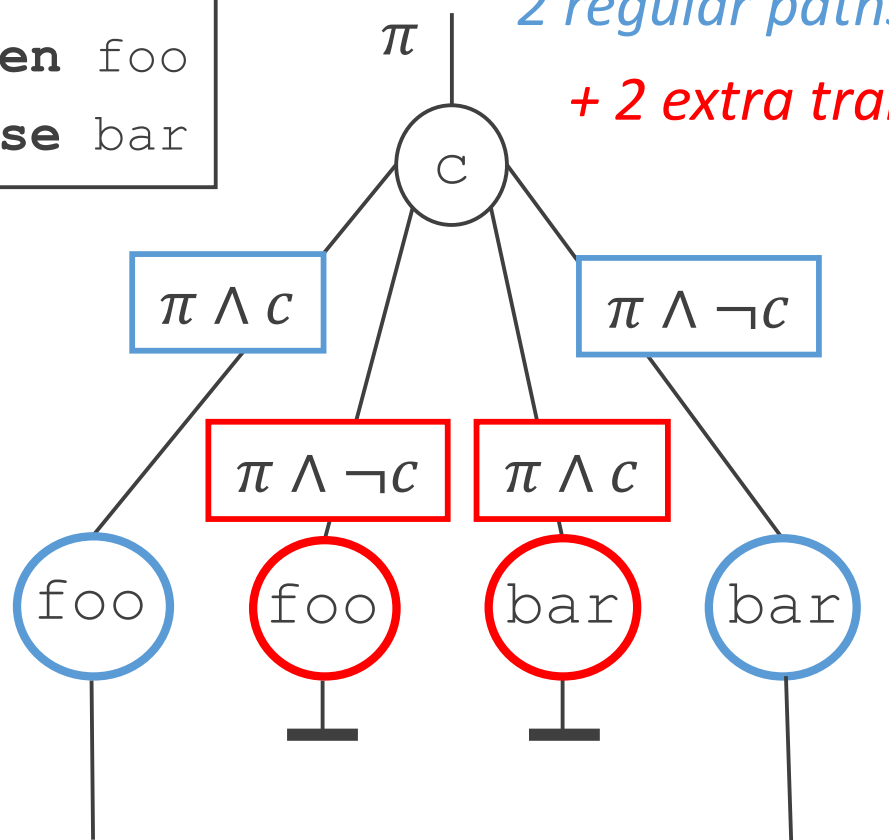
```
if c
then foo
else bar
```



# Explicit RelSE for Spectre PHT

**Spectre-PHT.** Conditional branches can be executed speculatively

```
if c
then foo
else bar
```




*2 regular paths*

*+ 2 extra transient paths*

**Explicit RelSE.**

Fork execution into 4 at conditionals:

- 2 regular branches
- 2 transient branches (until max speculation depth)

On regular and transient branches:

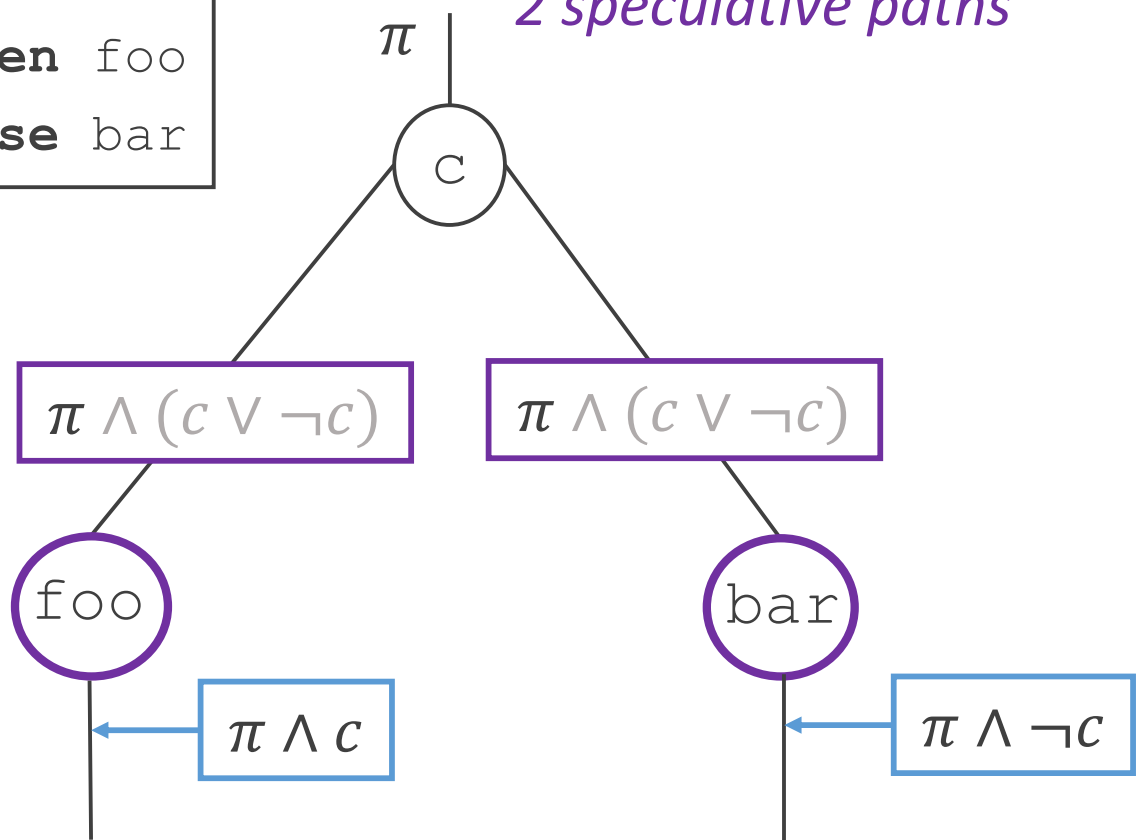
- Verify no secret can leak.

(e.g. KLEESpectre)

# Haunted RelSE for Spectre PHT

**Spectre-PHT.** Conditional branches can be executed speculatively

```
if c
then foo
else bar
```




## Haunted RelSE.

Fork execution into 2 speculative paths:

- speculative = regular ∨ transient
- After max spec. depth, add constraint to invalidate transient path

→ *can spare two paths at conditionals*

# Haunted vs. Explicit for Spectre-PHT

**Litmus tests** (32 programs) ↗

| | Paths | Time | Timeout | Bugs |
|---|---|---|---|---|
| **Explicit** | 1546 | ≈3h | 2 | 21 |
| **Haunted** | 370 | 15s | 0 | 22 |

**Libsodium & OpenSSL** (3 programs) ↗

| | X86 Instr. | Time | Timeout | Bugs |
|---|---|---|---|---|
| **Explicit** | 2273 | 18h | 3 | 43 |
| **Haunted** | 8634 | ≈8h | 1 | 47 |

**Tea and donna (10 programs).** No difference between Explicit and Haunted ≈

**Take away, Haunted RelSE vs Explicit RelSE.**

- At worse: no overhead compared to Explicit ≈
- At best: faster, more coverage, less timeouts ↗

# Weakness of index-masking countermeasure

# Weakness of Spectre-PHT countermeasure

**Index masking**. Add branchless bound checks

Program vulnerable to Spectre-PHT

```
if (idx < size) { // size = 256

      v = tab[idx]
      leak(v)
}
```

# Weakness of Spectre-PHT countermeasure

**Index masking**. Add branchless bound checks

Index masking countermeasure

```
if (idx < size) { // size = 256
        idx = idx & (0xff)
        v = tab[idx]
        leak(v)
}
```

# Weakness of Spectre-PHT countermeasure

**Index masking**. Add branchless bound checks

Index masking countermeasure

```
if (idx < size) { // size = 256
        idx = idx & (0xff)
        v = tab[idx]
        leak(v)
}
```

Compiled version with gcc –O0 –m32

```
store  @idx (load @idx & 0xff)
eax = load @idx
al = [@tab + eax]
leak (al)
```

- Masked index stored in memory
- Store may be bypassed with Spectre-STL !

# Weakness of Spectre-PHT countermeasure

**Index masking**. Add branchless bound checks

Index masking countermeasure

```
if (idx < size) { // size = 256
        idx = idx & (0xff)
        v = tab[idx]
        leak(v)
}
```

Compiled version with gcc –O0 –m32

```
store  @idx (load @idx & 0xff)
eax = load @idx
al = [@tab + eax]
leak (al)
```

- Masked index stored in memory
- Store may be bypassed with Spectre-STL !

**Verified mitigations:**

- Enable optimizations (depends on compiler choices)
- Explicitly put masked index in a register `register uint32_t ridx asm ("eax");`