Efficient Relational Symbolic Execution for Speculative Constant-Time at Binary-Level

- Efficient constant-time verification at binary-level (overview)
- Adaptation to detect Spectre attacks

Journée 2021 du GT "Méthodes Formelles pour la Sécurité"
March, 16th 2021

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

Sébastien Bardin
CEA, LIST, Université Paris-Saclay
France

Tamara Rezk
Inria
France

Binsec/Rel:
Efficient constant-time verification at binary-level

(overview)

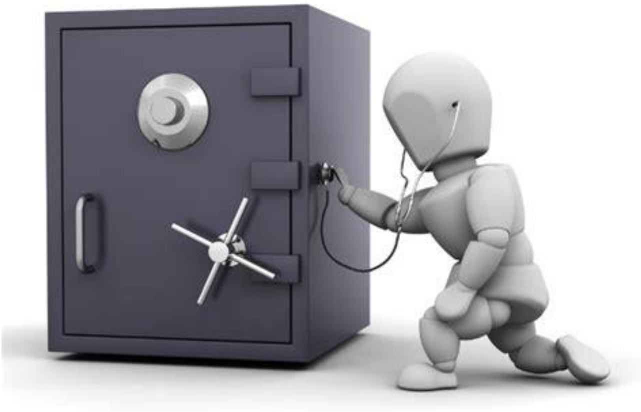
MAY 18-20, 2020

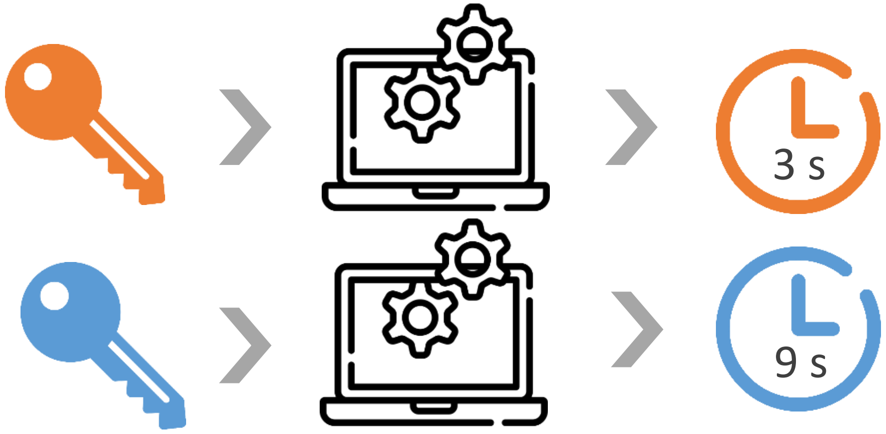
41st IEEE Symposium on Security and Privacy

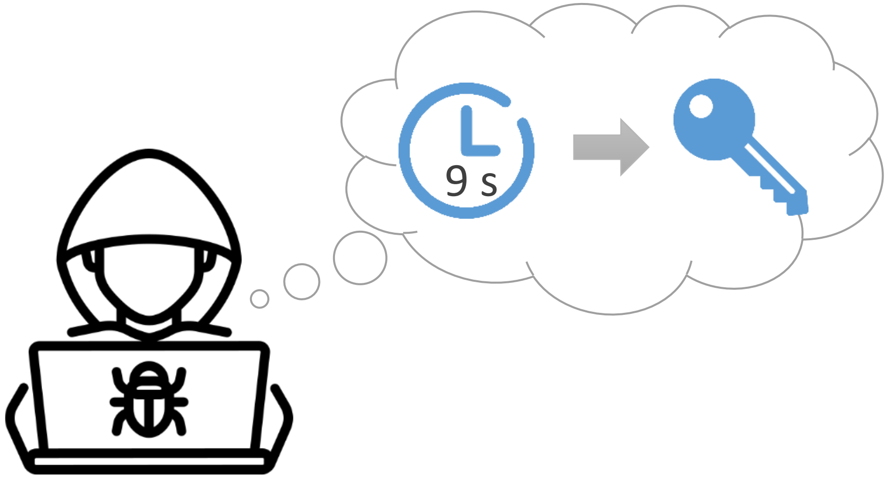
Context: Timing Attacks

Timing attacks: execution time of programs can leak secret information

First timing attack in **1996** by Paul Kocher: full recovery of **RSA encryption key**

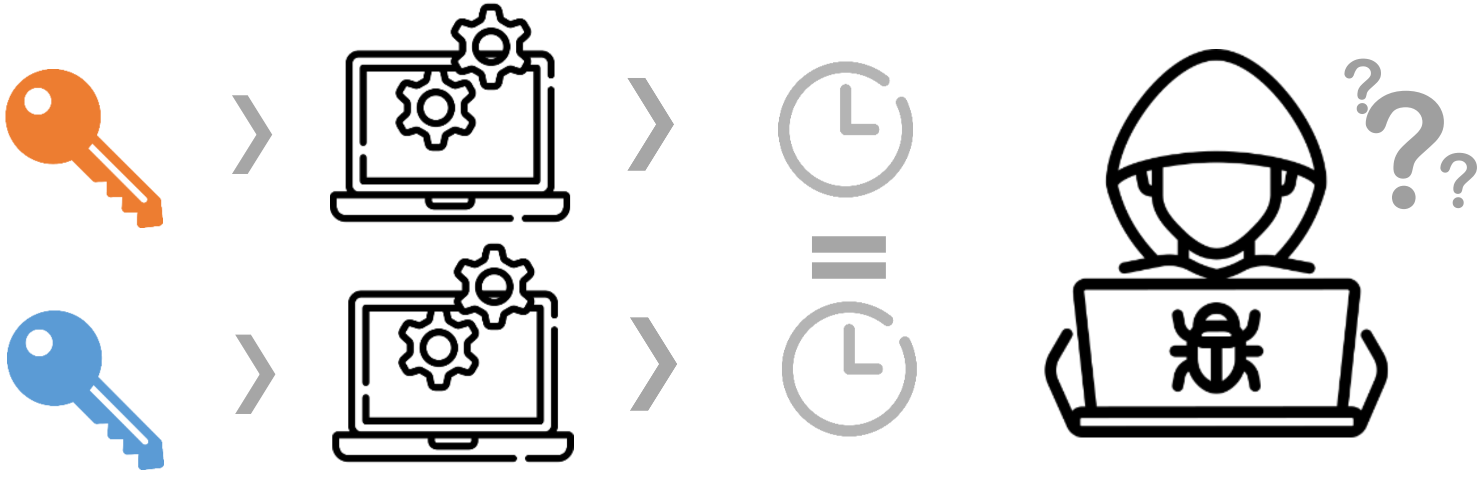






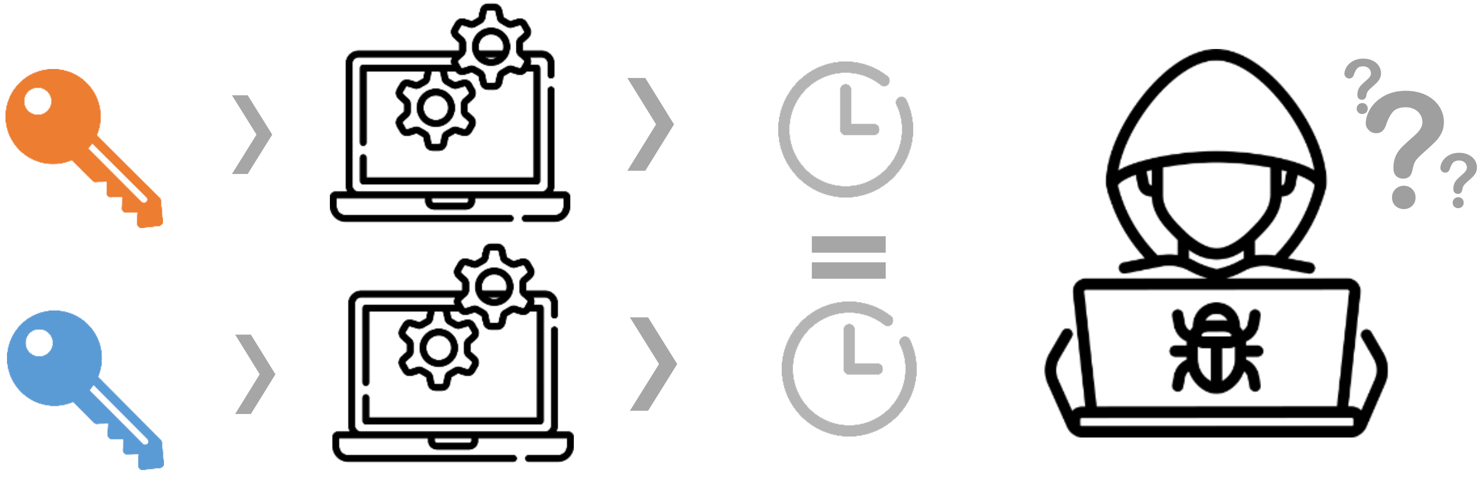
Protect Software with Constant-Time Programming

Constant-Time. Execution time is independent from secret input



Protect Software with Constant-Time Programming

Constant-Time. Execution time is independent from secret input

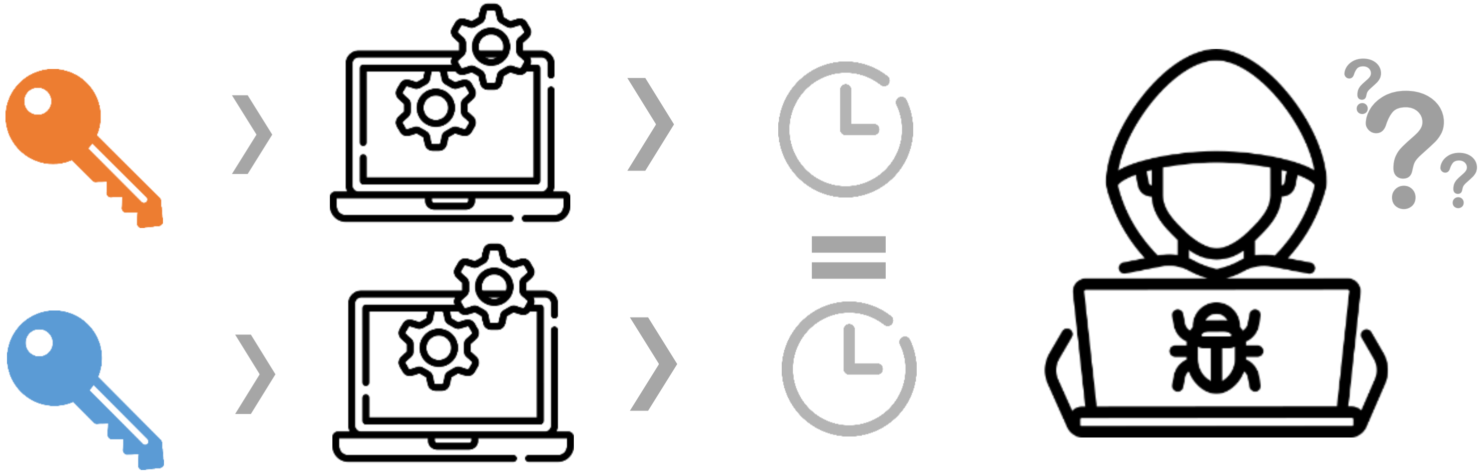


Property relating 2 execution traces (2-hypersafety)

Protect Software with Constant-Time Programming

Constant-Time. Execution time is independent from secret input

→ Control-flow
→ Memory accesses



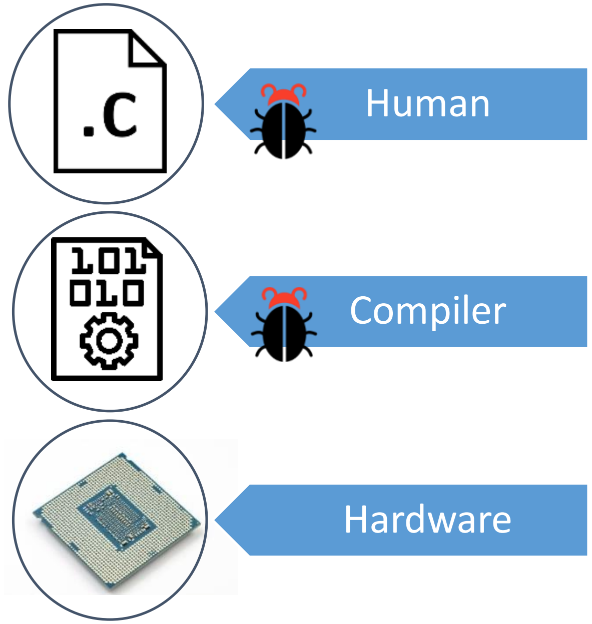
Property relating 2 execution traces (2-hypersafety)

Problem: Need Automated Verification Tools

Execution time is not easy to determine

- Sequence of instructions executed
- Memory accesses (Cache attacks, 2005)

Multiple failure points

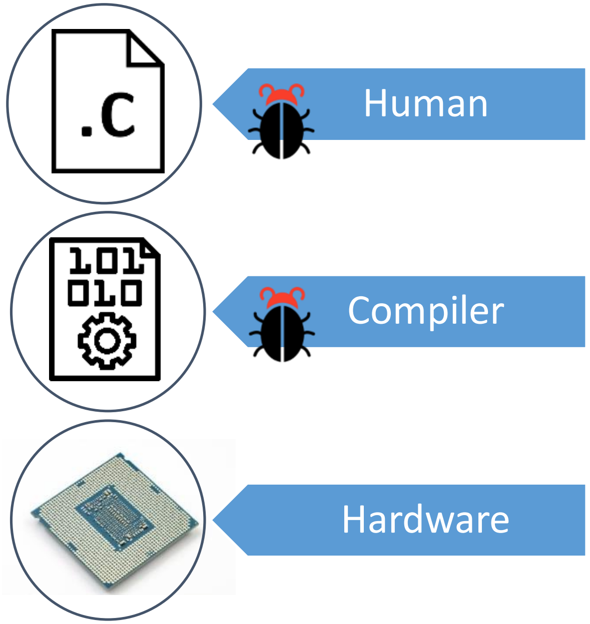


Problem: Need Automated Verification Tools

Execution time is not easy to determine

- Sequence of instructions executed
- Memory accesses (Cache attacks, 2005)

Multiple failure points

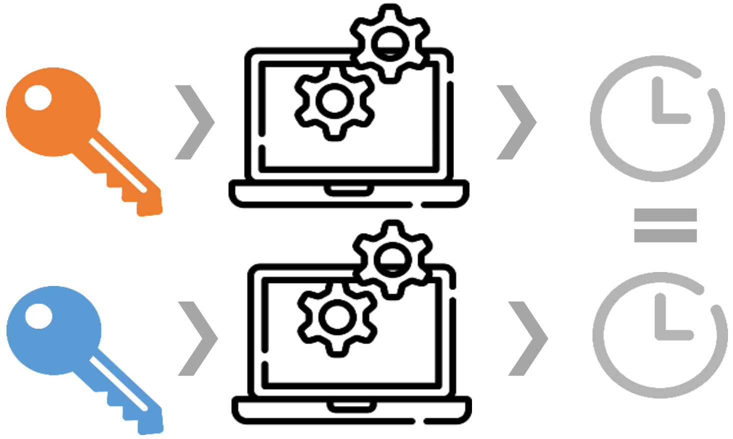


Not easy to write constant-time programs

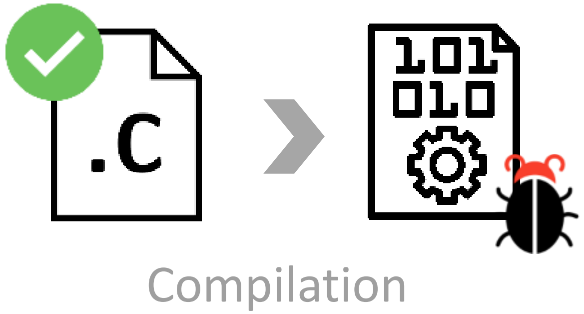
We need efficient automated verification tools!

Challenges for CT analysis

Property of 2 executions

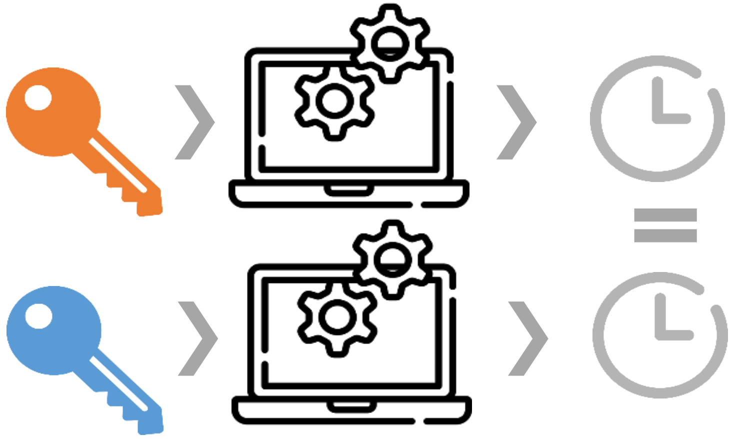


Not necessarily preserved by compilers



Challenges for CT analysis

Property of 2 executions

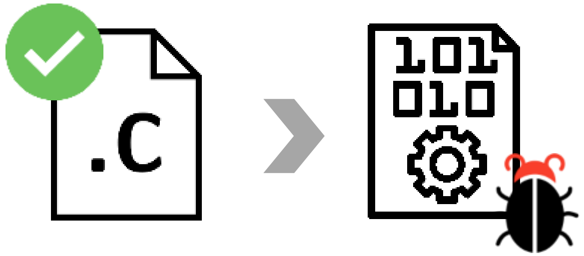


→ Efficiently model pairs of executions

RelSE (SE for pairs of traces with sharing)

for Bug-Finding & Bounded-Verif

Not necessarily preserved by compilers



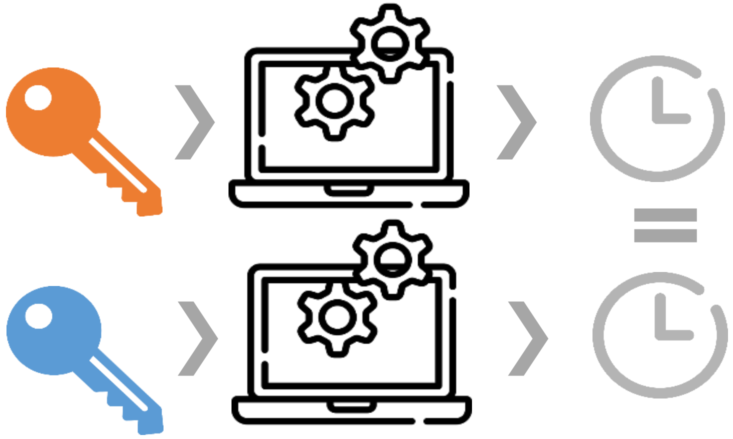
Compilation

→ Binary-analysis (harder)

BINSEC

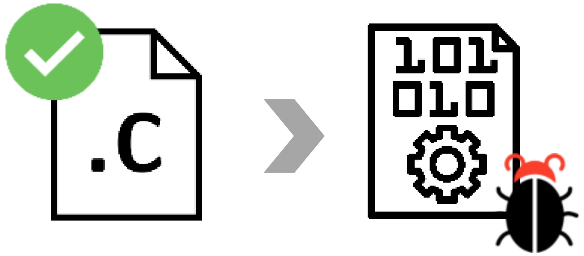
Challenges for CT analysis

Property of 2 executions



→ Efficiently model pairs of executions

Not necessarily preserved by compilers



Compilation

→ Binary-analysis (harder)

RelSE (SE for pairs of traces with sharing)
for Bug-Finding & Bounded-Verif

\+ BINSEC

= **Does not scale** ☹ (whole memory is duplicated, no sharing)

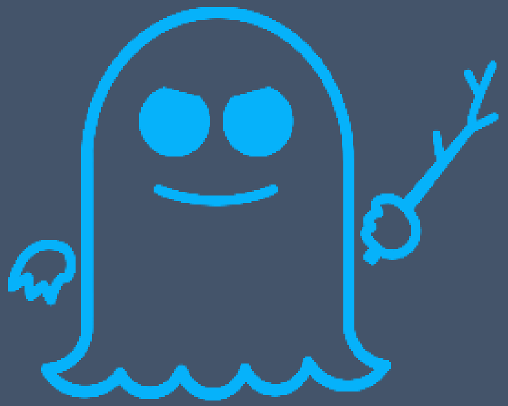
Contributions

Binsec/Rel https://github.com/binsec/rel

Efficient Relational Symbolic Execution for Constant-Time at Binary-Level

Optimizations	New Tool	Application: crypto verif.
Dedicated optimizations for RelSE at binary-level: maximize sharing in memory (x700 speedup)	**BINSEC/REL** First efficient tool for *BF & BV* of CT at *binary-level*	From OpenSSL, BearSSL, libsodium 296 verified binaries 3 new bugs introduced by compilers from verified source

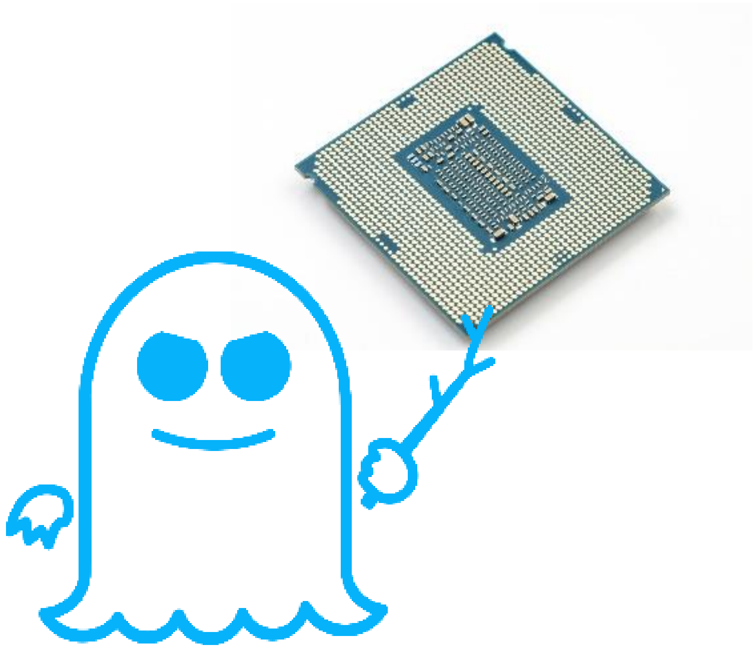
Haunted RelSE: detect Spectre vulnerabilities



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 $\neq$ 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)
```

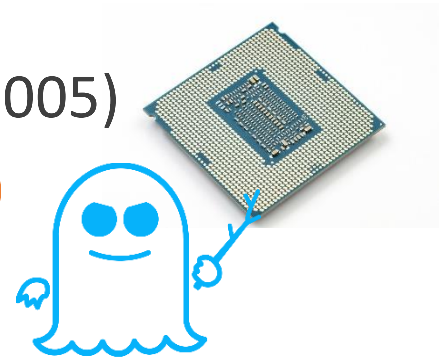
leak(init_mem[a])

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

Constant-time verification & Spectre attacks

Execution time is not easy to determine

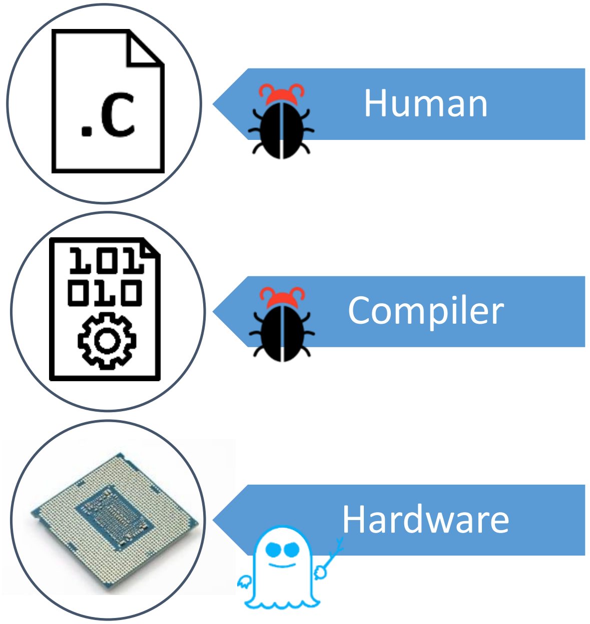
- Sequence of instructions executed
- Memory accesses (Cache attacks, 2005)
- Speculation (Spectre attacks, 2018)



Not easy to write constant-time programs

We need efficient automated verification tools that take into account speculation mechanisms in processors.

Multiple failure points



Detect Spectre attacks ?

Challenging !

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

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*
→ *Build on Relational Symbolic Execution (RelSE)*

Challenge. Model new transient behaviors avoiding path explosion

No efficient verification tools for Spectre ☹

	Target	Spectre-PHT	Spectre-STL
KLEESpectre [1]	LLVM	☺	-
SpecuSym [2]	LLVM	☺	-
FASS [3]	Binary	☹	-
Spectector [4]	Binary	☹	-
Pitchfork [5]	Binary	😐	☹

Legend

☺ Good perfs. on crypto

😐 Good on small programs
Limited perfs. On crypto

☹ Limited to small programs

LLVM analysis might miss SCT violations ☹

[1] G. Wang, et al "KLEESpectre: Detecting Information Leakage through Speculative Cache Atttacks via Symbolic Execution", ACM Trans. Softw. Eng. Methodol., vol. 29, no. 3, 2020.
[2] S. Guo, Y. Chen, P. Li, Y. Cheng, H. Wang, M. Wu, and Z. Zuo, "SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection", in ICSE 2020 Technical Papers, 2020.
[3] K. Cheang, C. Rasmussen, S. A. Seshia, and P. Subramanyan, "A Formal Approach to Secure Speculation", in CSF, 2019.
[4] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled Detection of Speculative Information Flows", in S&P, 2020
[5] S. Cauligi, C. Disselkoen, K. von Gleissenthall, D. M. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-Time Foundations for the New Spectre Era", in PLDI, 2020.

No efficient verification tools for Spectre ?

	Target	Spectre-PHT	Spectre-STL
KLEESpectre [1]	LLVM	☺	-
SpecuSym [2]	LLVM	☺	-
FASS [3]	Binary	☹	-
Spectector [4]	Binary	☹	-
Pitchfork [5]	Binary	😐	☹
Binsec/Haunted	**Binary**	☺	😐

Legend

☺ Good perfs. on crypto

😐 Good on small programs
Limited perfs. On crypto

☹ Limited to small programs

LLVM analysis might miss SCT violations ☹

[1] G. Wang, et al “KLEESpectre: Detecting Information Leakage through Speculative Cache Atttacks via Symbolic Execution”, ACM Trans. Softw. Eng. Methodol., vol. 29, no. 3, 2020.
[2] S. Guo, Y. Chen, P. Li, Y. Cheng, H. Wang, M. Wu, and Z. Zuo, “SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection”, in ICSE 2020 Technical Papers, 2020.
[3] K. Cheang, C. Rasmussen, S. A. Seshia, and P. Subramanyan, “A Formal Approach to Secure Speculation”, in CSF, 2019.
[4] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, “Spectector: Principled Detection of Speculative Information Flows”, in S&P, 2020
[5] S. Cauligi, C. Disselkoen, K. von Gleissenthall, D. M. Tullsen, D. Stefan, T. Rezk, and G. Barthe, “Constant-Time Foundations for the New Spectre Era”, in PLDI, 2020.

Contributions

Haunted RelSE optimization

- Model transient and regular behaviors at the same time
  - **Spectre-PHT**: pruning redundant paths
  - **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

New Spectre-STL violations

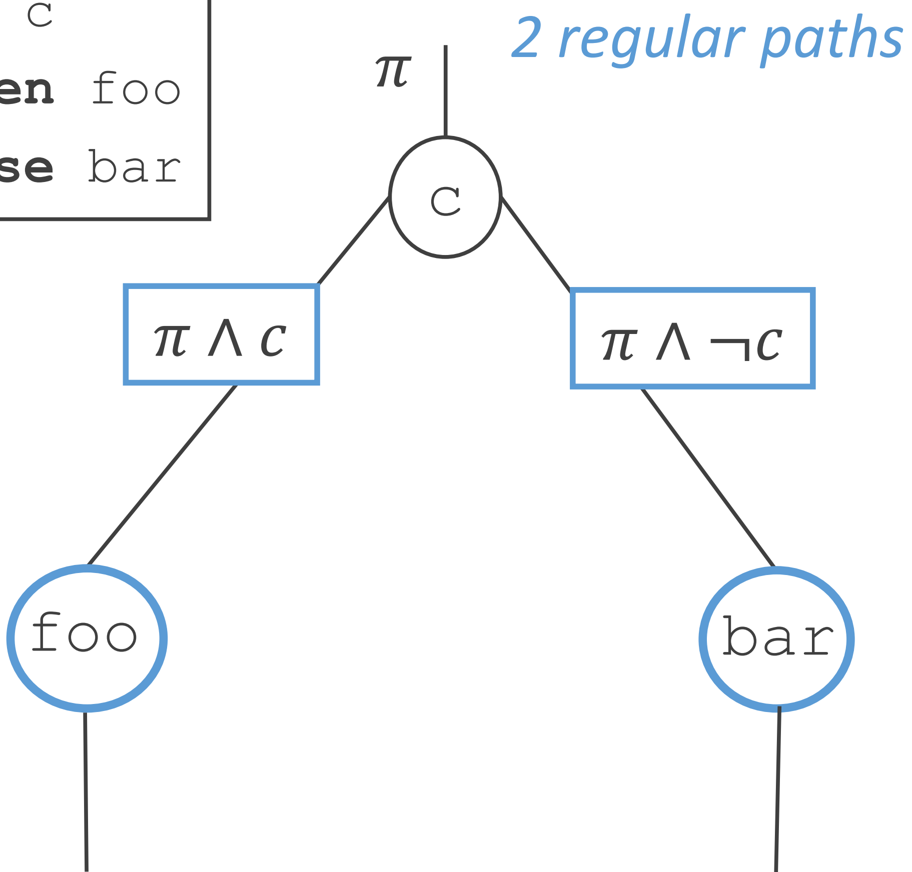
- Index-masking (countermeasure against Spectre-PHT) + proven mitigations
- Code introduced for Position-Independent-Code [in the paper]

Haunted RelSE for Spectre-PHT

Background: Symbolic Execution

Symbolic execution. An illustration.

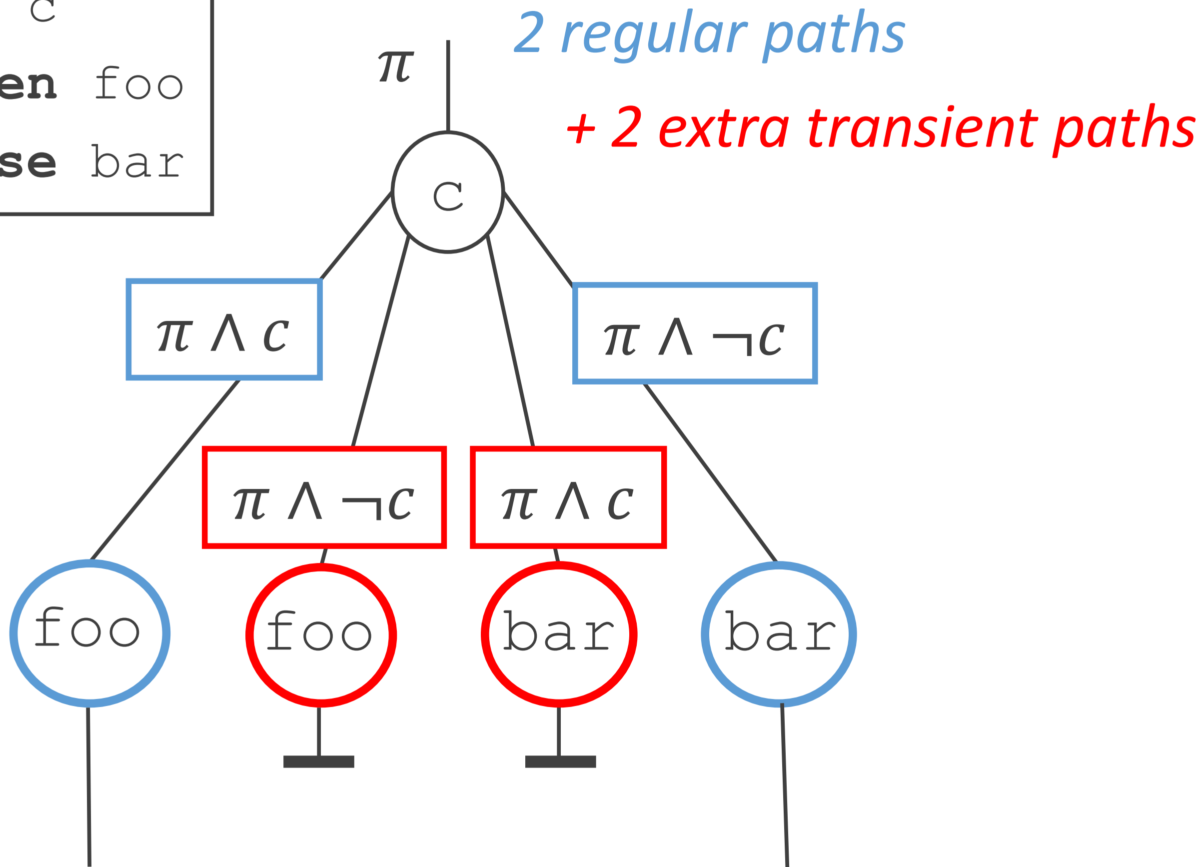
```
if c
then foo
else bar
```



Explicit RelSE for Spectre PHT

Spectre-PHT. Conditional branches can be executed speculatively

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



Explicit RelSE.

Fork execution into 4 at conditionals:

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

On regular and transient branches:

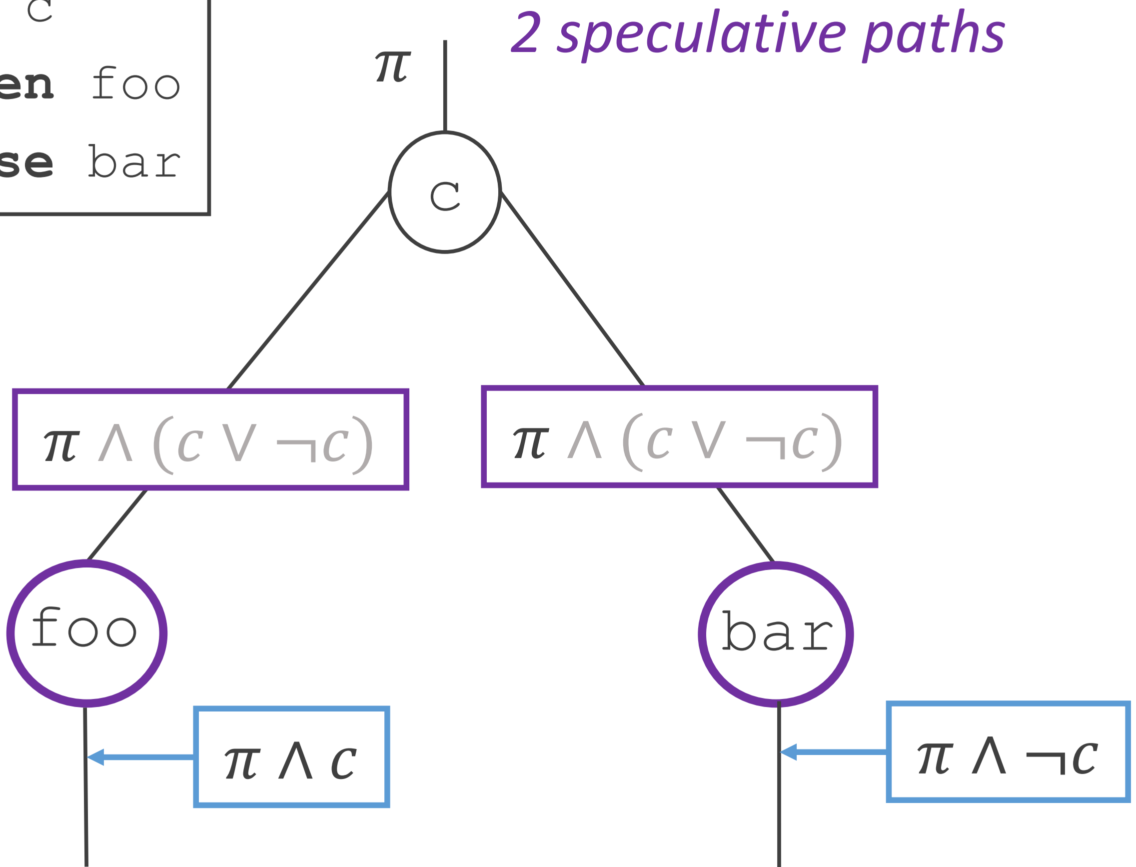
- Verify no secret can leak.

(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

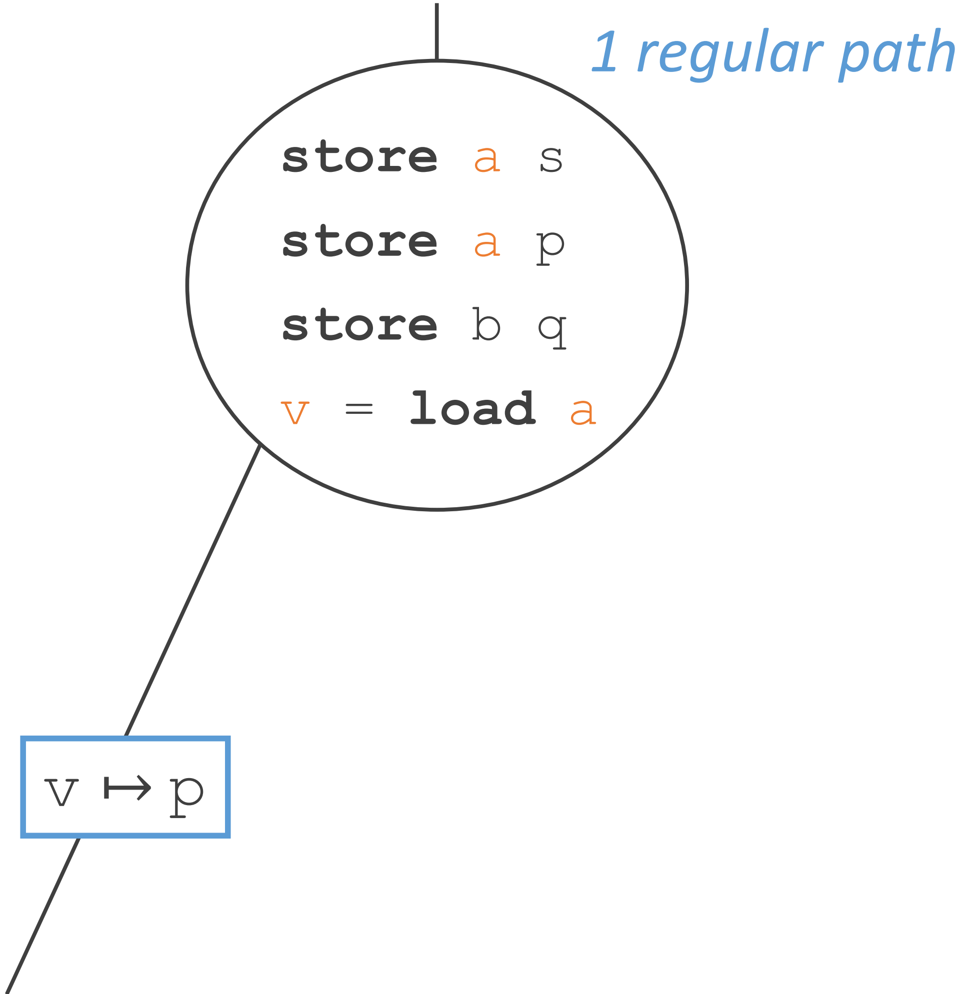
→ *can spare two paths at conditionals*

Haunted RelSE for Spectre-STL

Explicit RelSE for Spectre-STL

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

where a ≠ 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
```

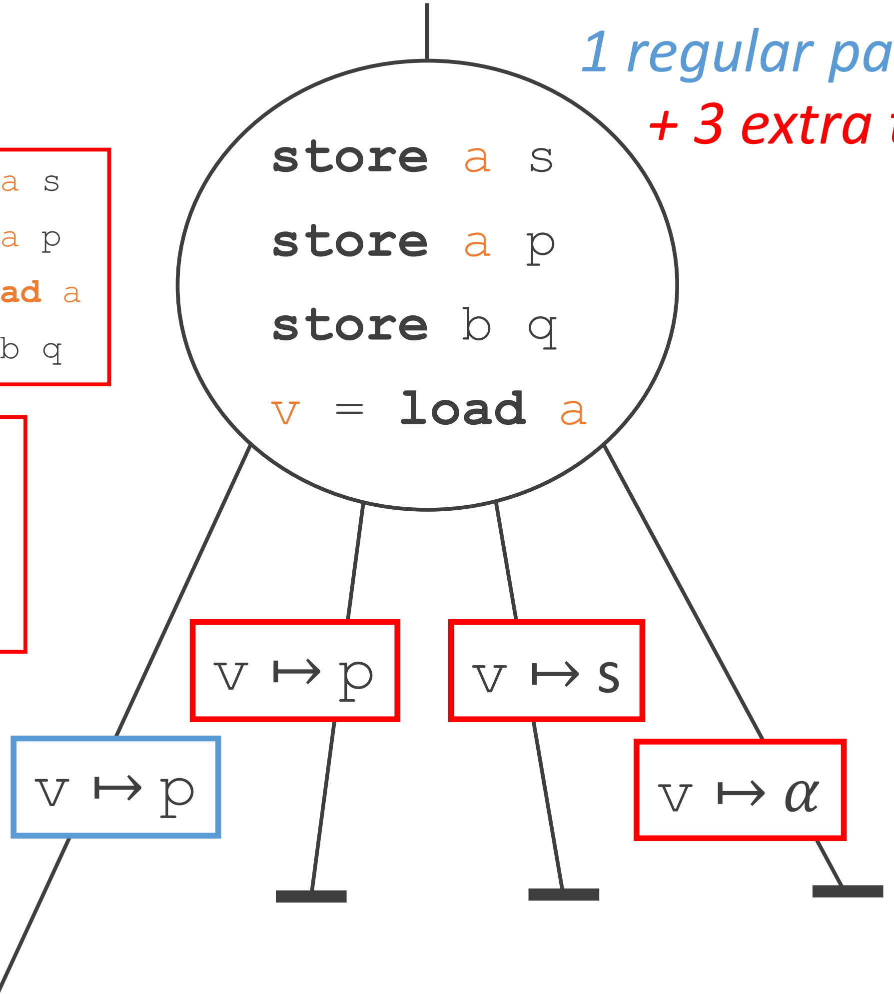
```
store a s
v = load a
store a p
store b q
```

```
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

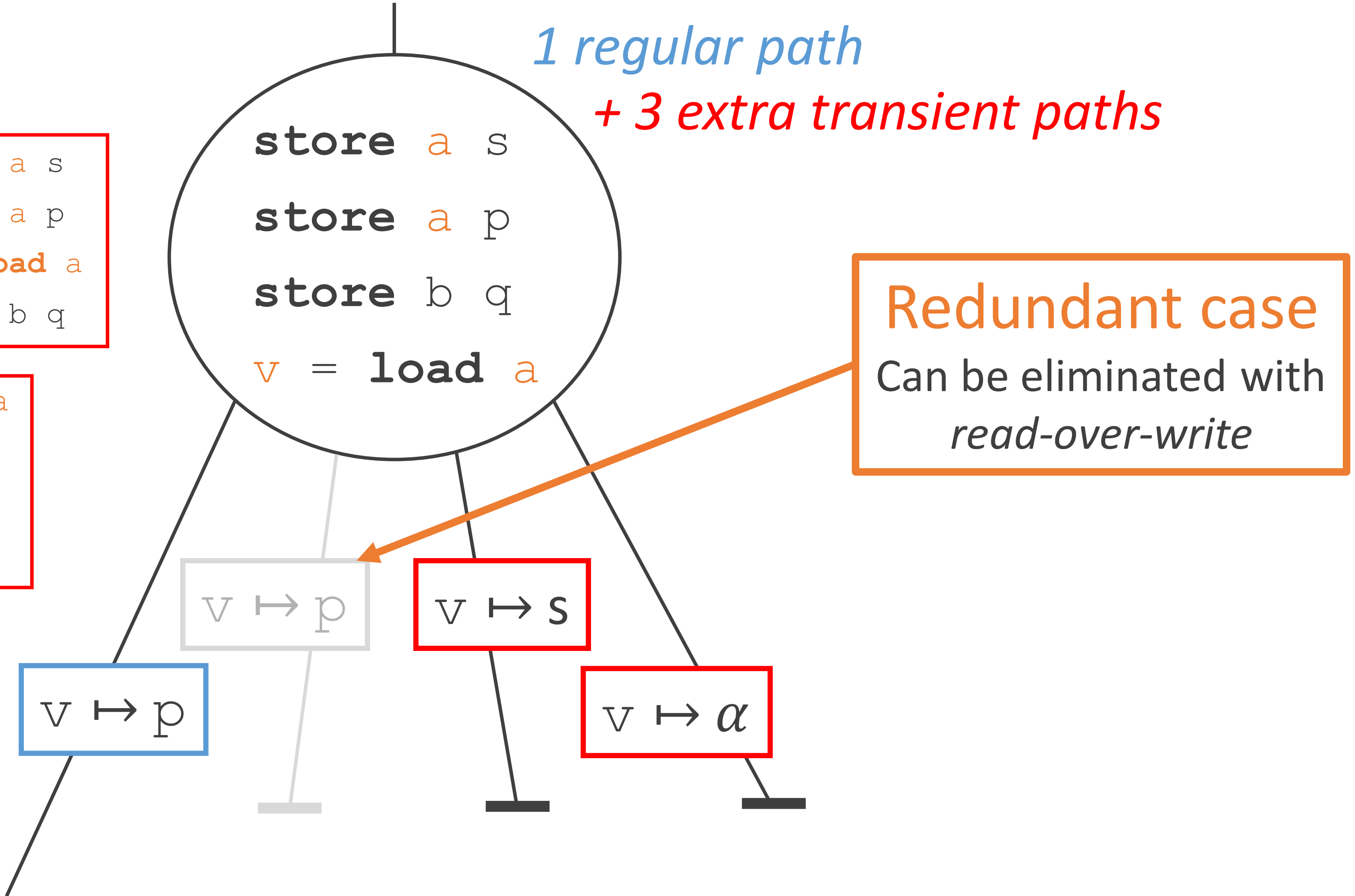
```
store a s
store a p
store b q
v = load a
```

```
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 $a \neq b$



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 speculative path

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

Haunted RelSE.

- Cut redundant cases
- Encode remaining ones in 1 path
  - symbolic *ite*
  - free booleans β_0, β_1

$v \mapsto$ *ite* β_0 *then* α *else* (*ite* β_1 *then* s *else* p)

$\beta_0 = false$

$\beta_1 = false$

Experimental evaluation

Experimental evaluation

Binsec/Haunted.
Implementation of Haunted RelSE

Binsec/Haunted

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-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 ↗

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 ! ↗

Comparison Binsec/Haunted against Pitchfork & KLEESpectre (RQ3)

	Target	Programs	PHT	STL
KLEESpectre	LLVM	Litmus tests Tea & donna	**Explicit** 😐 (≈240× slower) ☺ (≈equivalent)	NA
Pitchfork	Binary	Litmus tests Tea & donna	**Optims** ☺ (≈equivalent) ☹ (50× slower & TO)	**Explicit** ☹ 6/10 TO ☹ TO
Binsec/Haunted	Binary	Litmus tests Tea & donna	**Haunted** ☺ ☺	**Haunted** ☺ 😐

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");`

Wrap-up: detection of Spectre

- 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

Conclusion

Conclusion

Binsec/Rel

https://github.com/binsec/rel

Binsec/Haunted

https://github.com/binsec/haunted

- Dedicated optimizations for RelSE at binary-level
- Binsec/Rel, binary-level tool for bug-finding & bounded-verif. of CT
- Verif of crypto libraries at binary-level + new bugs introduced by compilers

- Haunted RelSE optimization for modelling speculative semantics
- Binsec/Haunted, binary-level tool to detect Spectre-PHT & STL
- New Spectre-STL violations with index masking and PIC