Symbolic Binary-Level Code Analysis for Security

Application to the Detection of Microarchitectural Attacks in Cryptographic Code

PhD defense - Lesly-Ann Daniel
CEA List and Université Côte d'Azur
Rejeu à RESSI 2022

Supervised by:
- Sébastien Bardin, CEA List
- Tamara Rezk, INRIA

Programs manipulate secret data

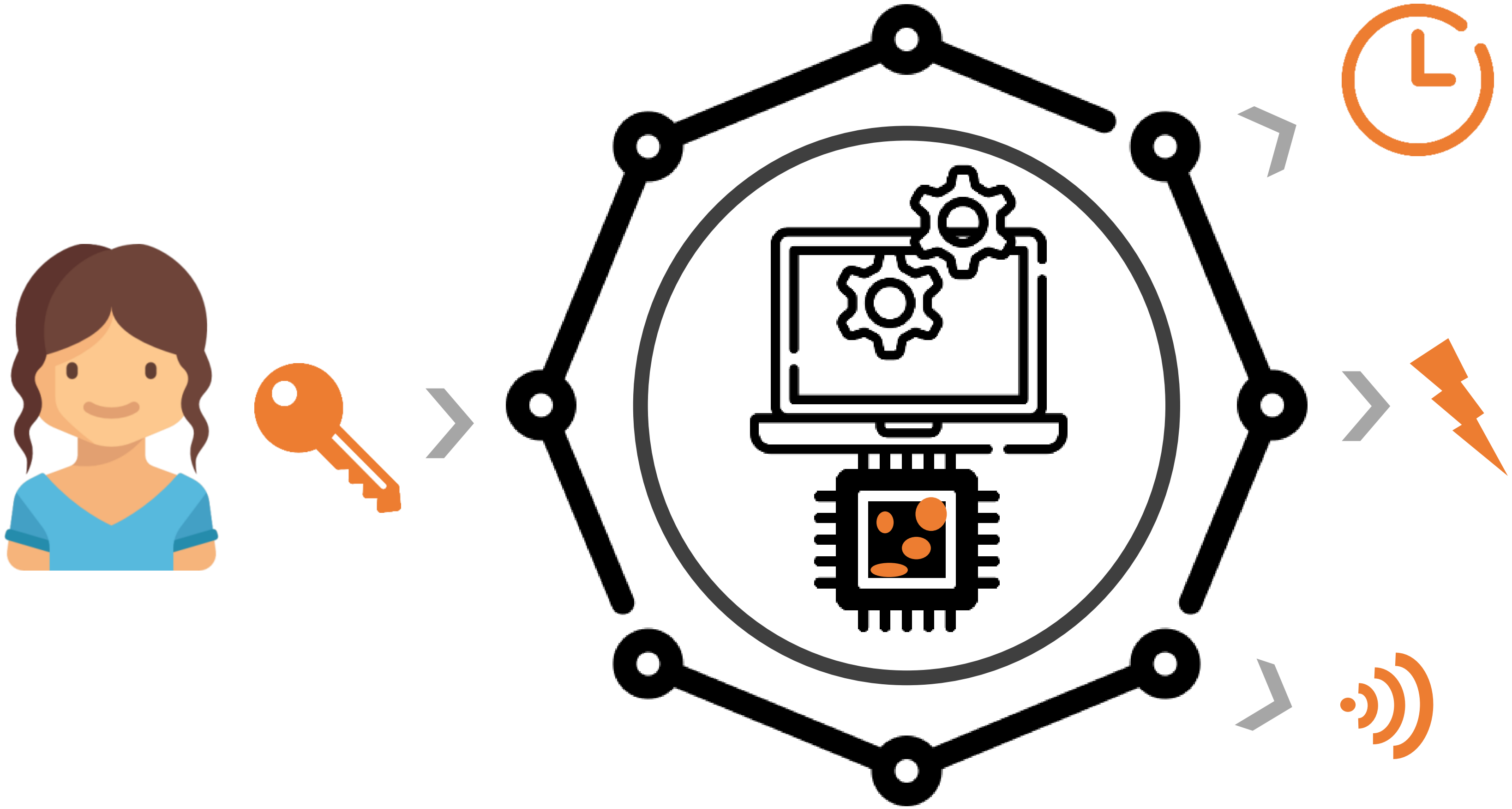
Critical software is prevalent:

- Secure communications
- Online banking
- Protect health data

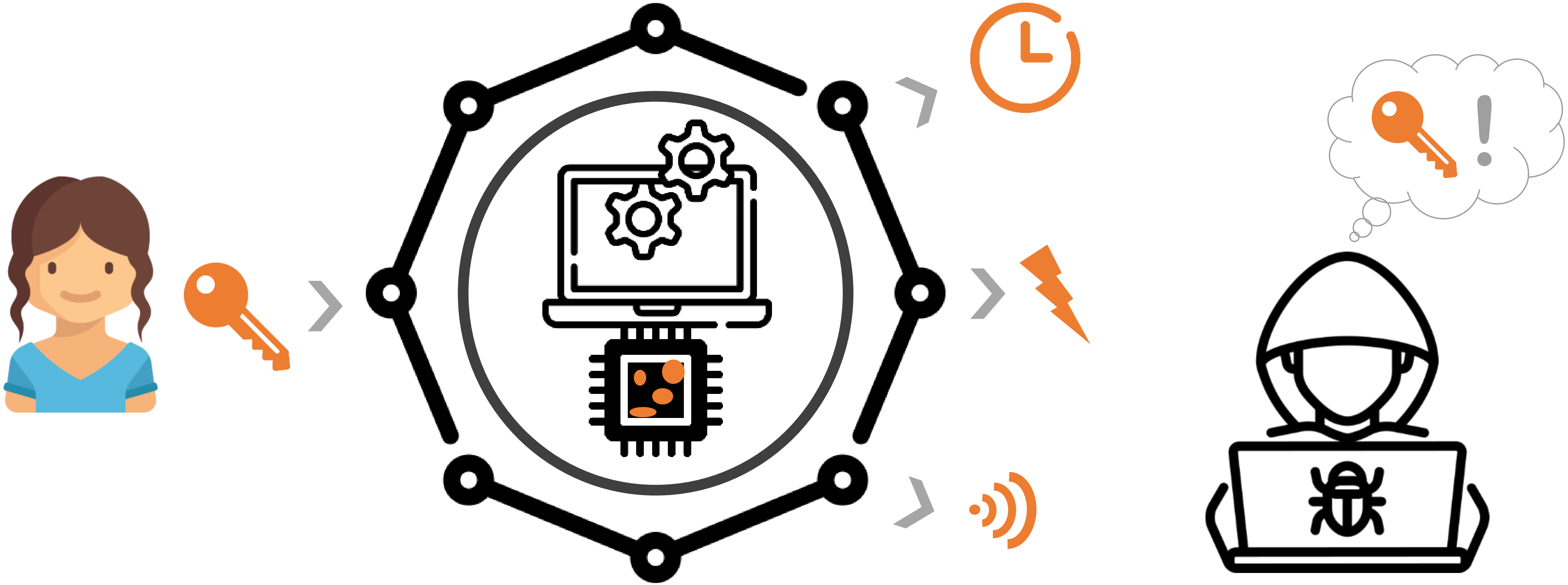
Their security relies on cryptography:

- Mathematical guarantees
- Verified implementations (no bugs, functional)
- *But what about their execution in the physical world?*

Computations have physical side effects

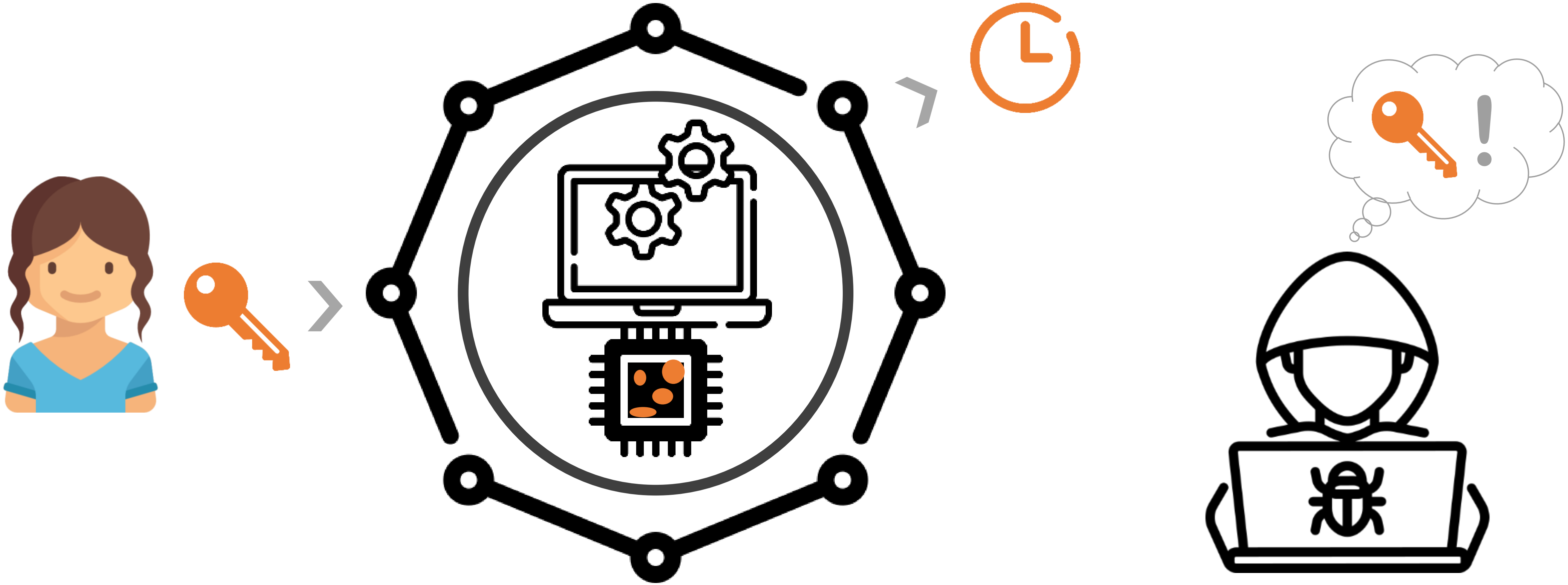


Computations have physical side effects



These side-effects can be exploited via side-channel attacks to recover secret data

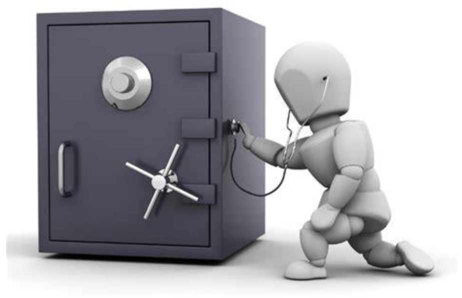
Computations have physical side effects



Timing and microarchitectural attacks can be run remotely [1]

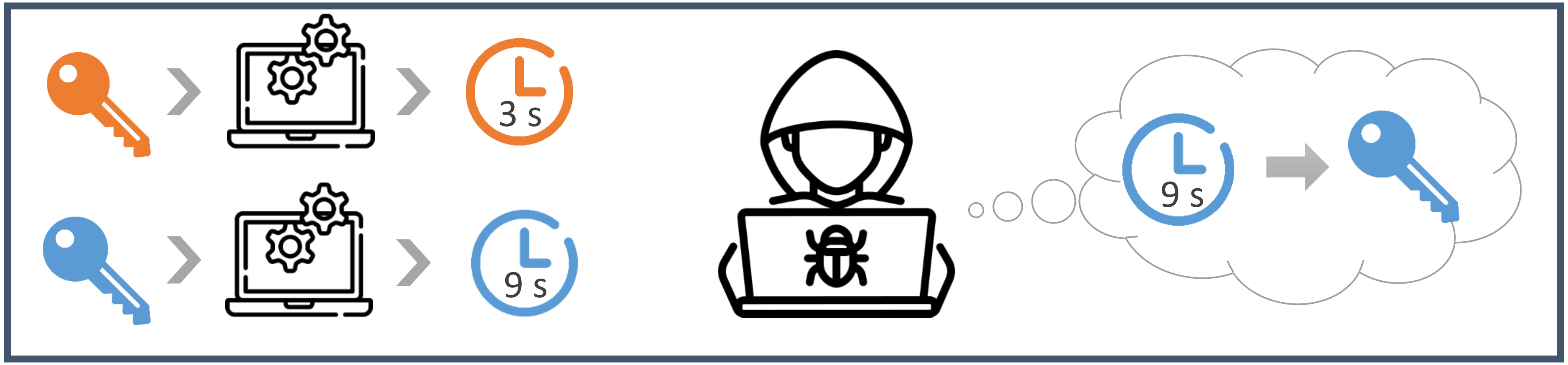
[1] Remote Timing Attacks Are Practical, David Brumley and Dan Boneh at USENIX 2003

Timing and Microarchitectural Attacks



Timing and microarchitectural attacks:

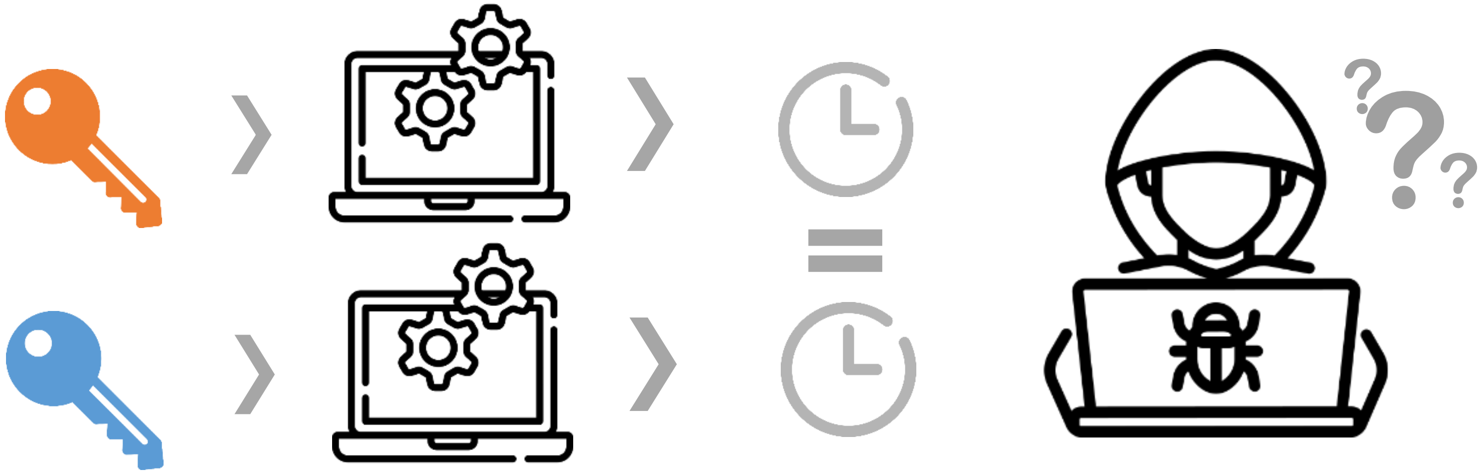
Execution time & microarchitectural state depends on secret data



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

Protect software with constant-time programming

Constant-Time. Execution time / changes to microarchitectural state must be independent from secret input

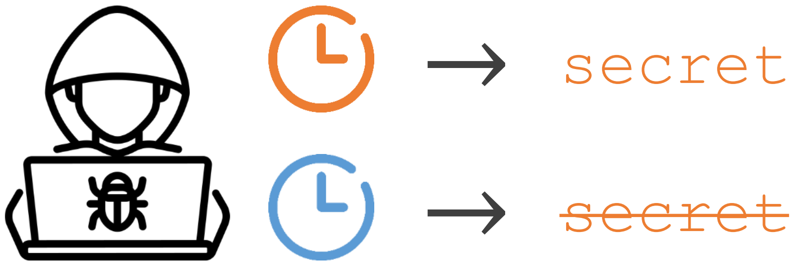


Already used in many cryptographic implementations

What can influence execution time/microarchitecture?

Control Flow

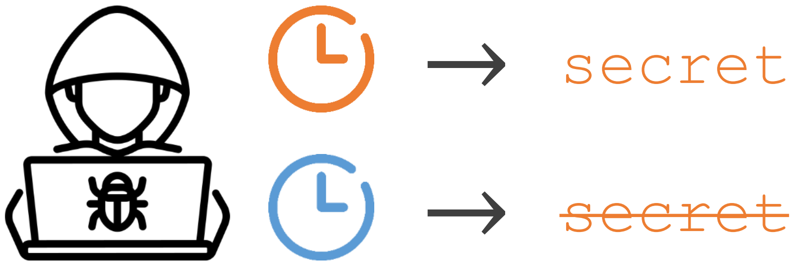
```
if secret
then foo()
else bar()
```



What can influence execution time/microarchitecture?

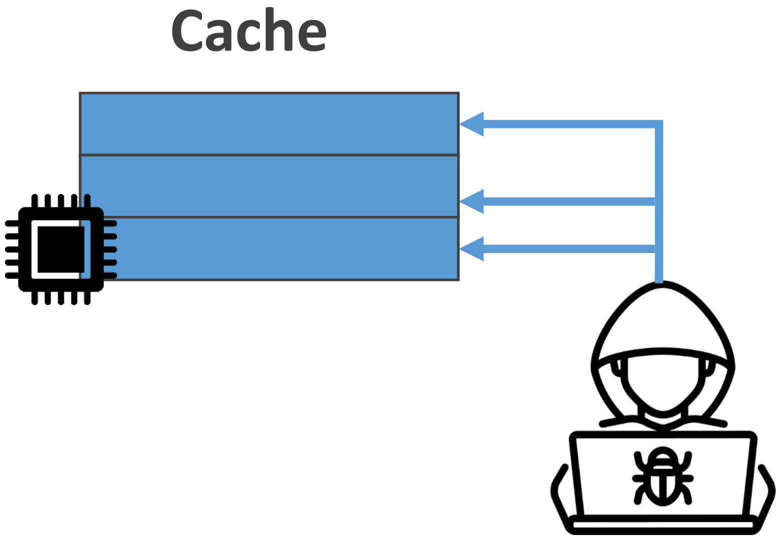
Control Flow

```
if secret
then foo()
else bar()
```

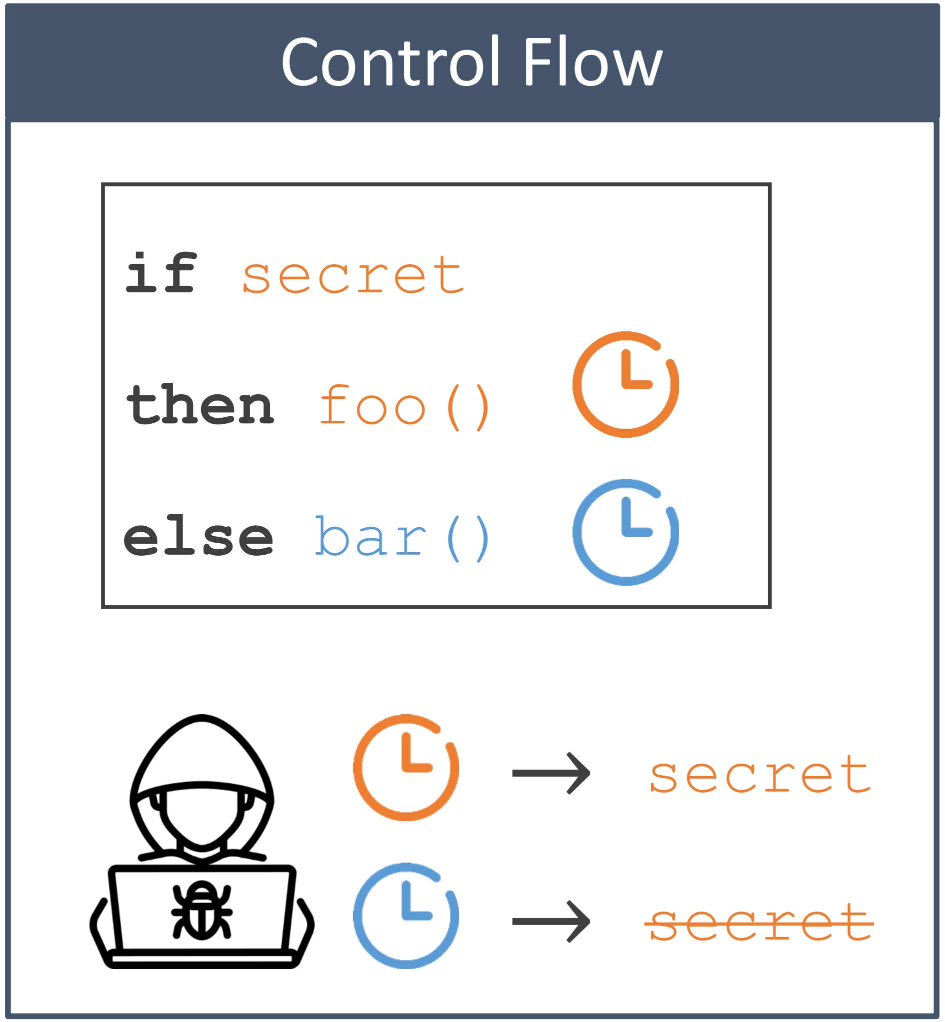


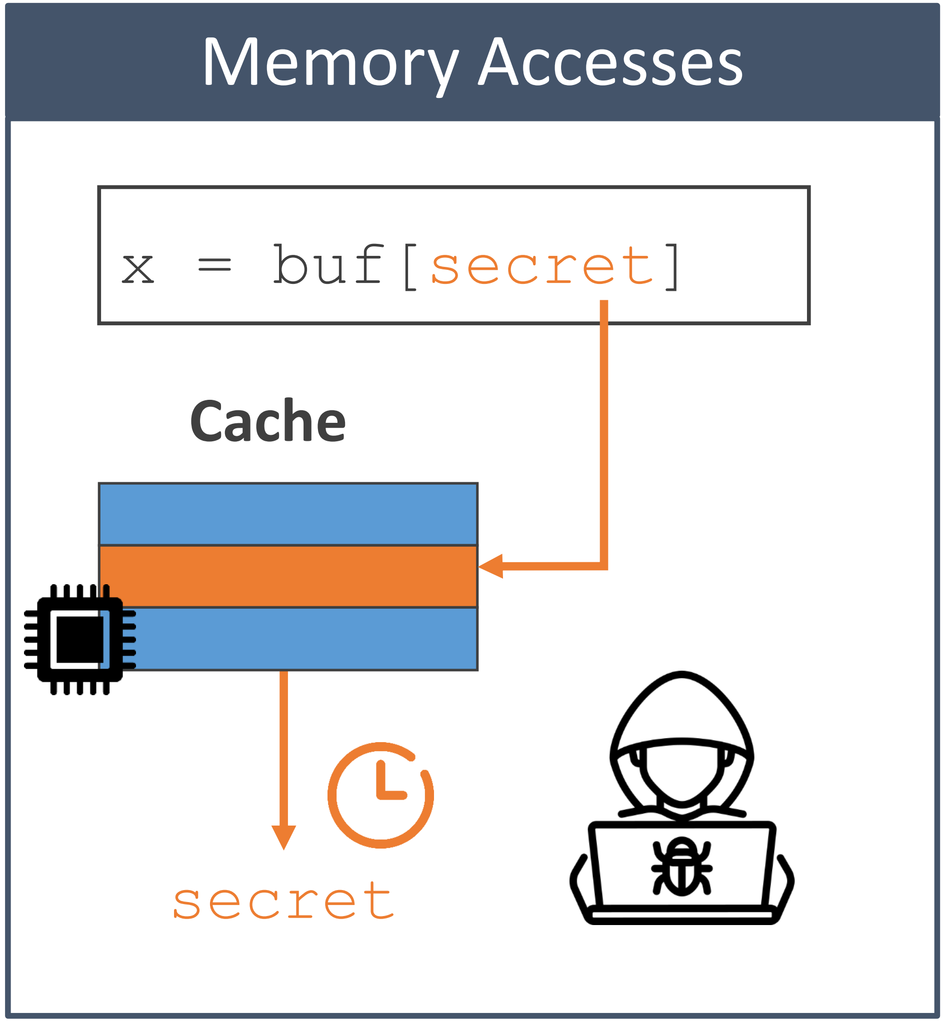
Memory Accesses

```
x = buf[secret]
```



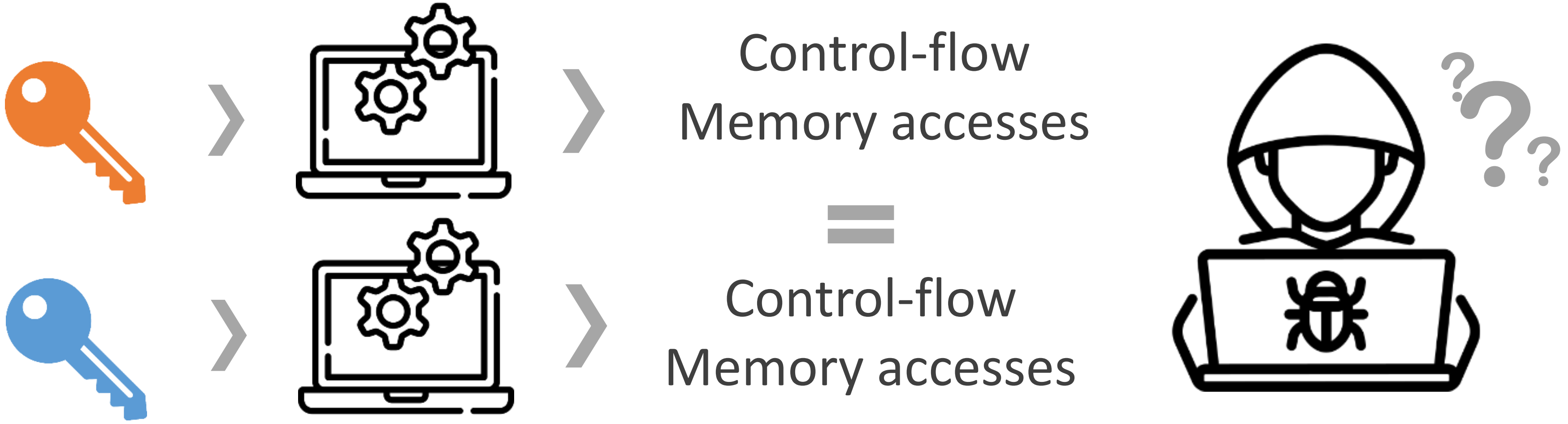
What can influence execution time/microarchitecture?





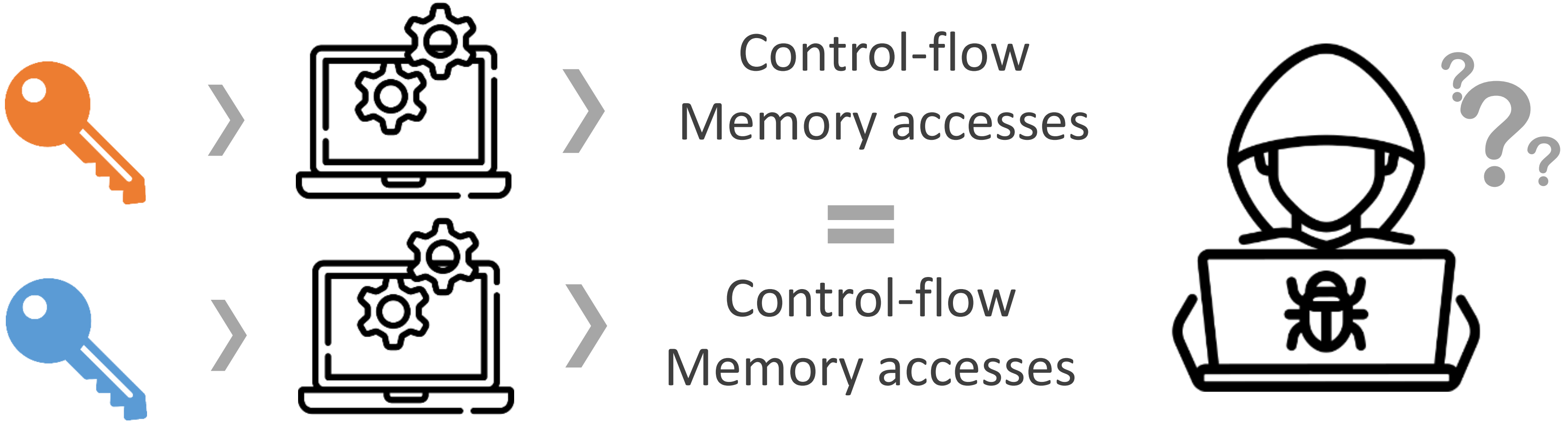
Protect software with constant-time programming

Constant-Time. Control-flow and memory accesses must be independent from secret input



Protect software with constant-time programming

Constant-Time. Control-flow and memory accesses must be independent from secret input



Property relating 2 execution traces (2-hypersafety)

Constant-time is not easy to implement

```c
uint32_t select(uint32_t x, uint32_t y, bool secret) {
  if (secret) return x;
  else return y;
}
```

```c
uint32_t ct_select(uint32_t x, uint32_t y, bool secret) {
  signed b = 0 - secret;
  return (x & b) | (y & ~b);
}
```

Compilers can break constant-time!

```c
uint32_t ct_select(uint32_t x, uint32_t y, bool secret) {
  signed b = 0 - secret;
  return (x & b) | (y & ~b);
}
```

clang-3.0 –O0

```
public ct_select_u32_v4
ct_select_u32_v4 proc near

var_14= dword ptr -14h
var_D= byte ptr -0Dh
var_C= dword ptr -0Ch
var_8= dword ptr -8
arg_0= dword ptr  4
arg_4= dword ptr  8
arg_8= byte ptr  0Ch

push    esi
sub     esp, 10h
mov     al, [esp+14h+arg_8]
mov     ecx, [esp+14h+arg_4]
mov     edx, [esp+14h+arg_0]
mov     [esp+14h+var_8], edx
mov     [esp+14h+var_C], ecx
and     al, 1
mov     [esp+14h+var_D], al
mov     al, [esp+14h+var_D]
and     al, 1
movzx   ecx, al
mov     edx, 0
sub     edx, ecx
mov     [esp+14h+var_14], edx
mov     ecx, [esp+14h+var_8]
and     ecx, [esp+14h+var_14]
mov     edx, [esp+14h+var_C]
mov     esi, [esp+14h+var_14]
xor     esi, 0FFFFFFFFh
and     esi, edx
or      esi, ecx
mov     eax, esi
add     esp, 10h
pop     esi
retn
ct_select_u32_v4 endp
```

clang-3.0 –O3

```
public ct_select_u32_v4
ct_select_u32_v4 proc near

arg_0= byte ptr  4
arg_4= byte ptr  8
arg_8= byte ptr  0Ch

mov     al, [esp+arg_8]
test    al, al
jz      short loc_804842F
```

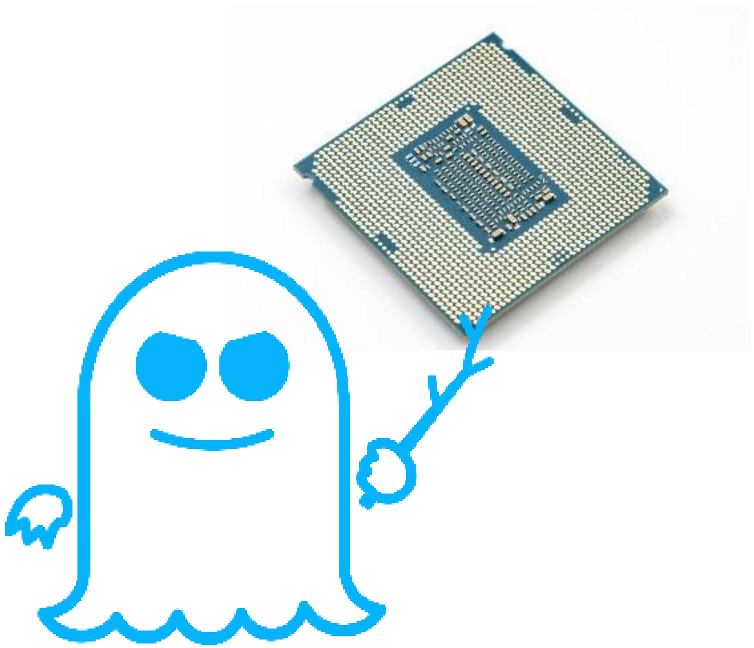
```
lea     eax, [esp+arg_0]
mov     eax, [eax]
retn
```

```
loc_804842F:
lea     eax, [esp+arg_4]
mov     eax, [eax]
retn
ct_select_u32_v4 endp
```

Spectre haunting our code

Spectre attacks (2018)

- Exploit speculations in processors
- Affect almost all processors
- Speculation may lead to 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*

Need automated verification for constant-time

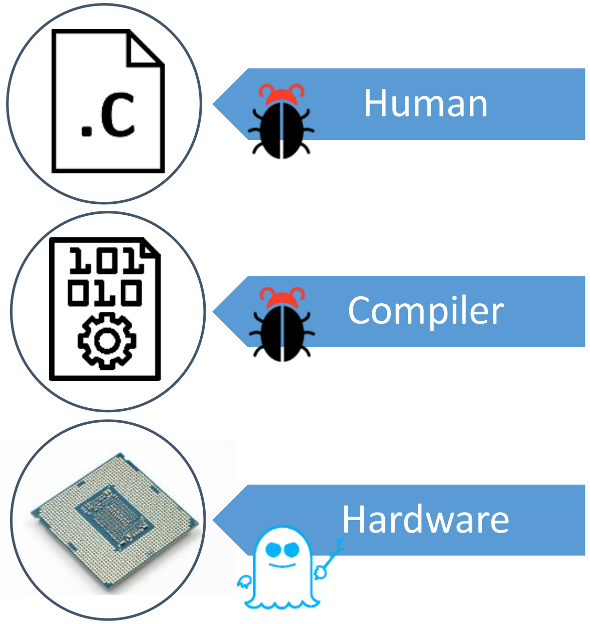
Constant time is crucial for security

Not easy to write constant-time programs:

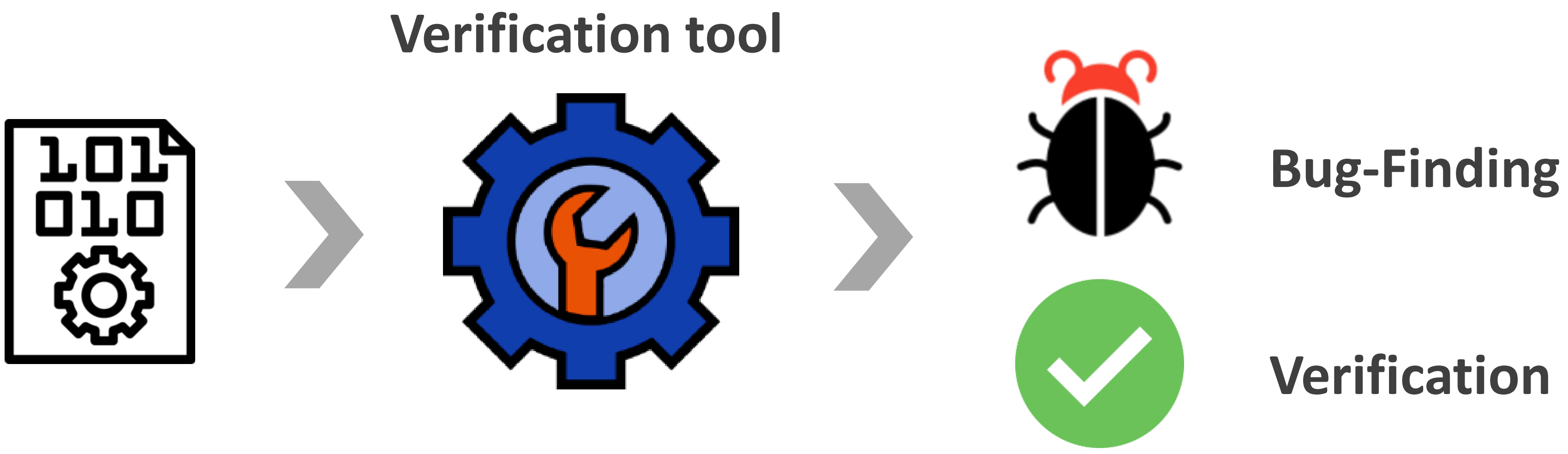
- Control-flow
  → First timing attacks by Paul Kocher, 1996
- Memory accesses
  → Cache attacks, 2005
- Processors optimizations
  → Spectre attacks, 2018

Efficient automated verification tools for constant-time at binary-level & modelling processor speculations

Multiple failure points



Automated program verification



Perfect verification tool:

- Reject only insecure programs
- Accept only secure programs
- Always terminate
- Be fully automatic

Not possible:
Non trivial semantic properties of programs are undecidable
Rice Theorem (1951)

Automated program verification

Verification tool

Bug-Finding

Bounded-Verification

Perfect verification tool:

- Reject only insecure programs
- Accept only secure programs up to a given bound
- Always terminate
- Be fully automatic

Symbolic Execution (SE)

The KeY Project

BINSEC

Contributions

- Optimizations: symbolic execution for constant-time, secret-erasure, detection of Spectre vulnerabilities at binary level
- Implementation into two open source tools

Binsec/Rel

MAY 18-20, 2020
41st IEEE Symposium on Security and Privacy

https://github.com/binsec/rel

Binsec/Haunted

NDSS SYMPOSIUM/2021

https://github.com/binsec/haunted

- Application to cryptographic primitives
  - Violations introduced by compilers from verified llvm code
  - Spectre-PHT defenses can be bypassed using Spectre-STL

Background:
Efficient SE for pairs of traces with Relational SE

MAY 18-20, 2020
41st IEEE Symposium on Security and Privacy

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

NDSS
SYMPOSIUM/2021

Haunted RelSE: detect Spectre vulnerabilities

Symbolic Execution [1,2]

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

Can error be reached?

[1] James C. King. *Symbolic execution and program testing,* Communications of the ACM, 1976
[2] Cristian Cadar and Sen Koushik. *Symbolic execution for software testing: three decades later.* Communications of the ACM, 2013

Symbolic Execution [1,2]

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

Can error be reached?

Symbolic store

$\mathtt{p} \mapsto p$

$\mathtt{s} \mapsto s$

[1] James C. King. *Symbolic execution and program testing,* Communications of the ACM, 1976
[2] Cristian Cadar and Sen Koushik. *Symbolic execution for software testing: three decades later.* Communications of the ACM, 2013

Symbolic Execution [1,2]

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

Can error be reached?

Symbolic store

p $\mapsto$ p

s $\mapsto$ s

c $\mapsto$ $p \times s - 48$

[1] James C. King. *Symbolic execution and program testing,* Communications of the ACM, 1976

[2] Cristian Cadar and Sen Koushik. *Symbolic execution for software testing: three decades later.* Communications of the ACM, 2013

Symbolic Execution [1,2]

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

Can error be reached?

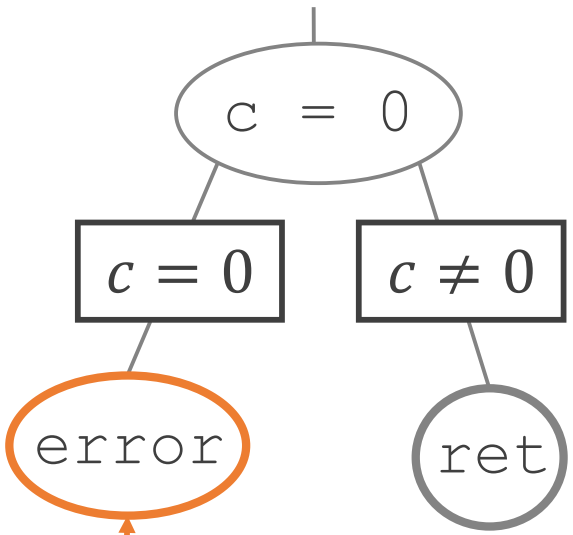
Symbolic store

$\mathtt{p} \mapsto p$

$\mathtt{s} \mapsto s$

$\mathtt{c} \mapsto p \times s - 48$

Path predicate



Formula $\mathrm{F}(p, s)$

$c = p \times s - 48 \wedge c = 0$

[1] James C. King. *Symbolic execution and program testing,* Communications of the ACM, 1976

[2] Cristian Cadar and Sen Koushik. *Symbolic execution for software testing: three decades later.* Communications of the ACM, 2013

Symbolic Execution [1,2]

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

Can error be reached?

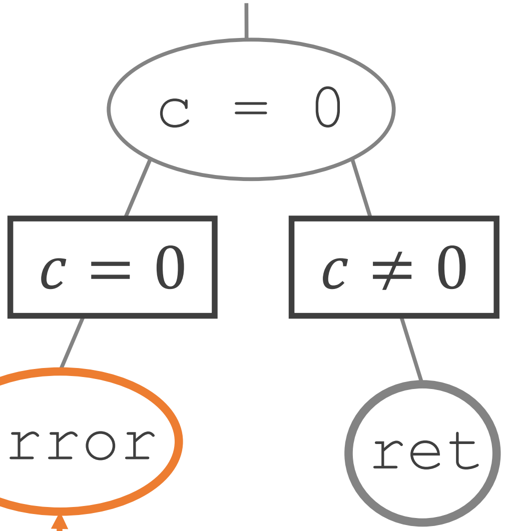
Symbolic store

$p \mapsto p$

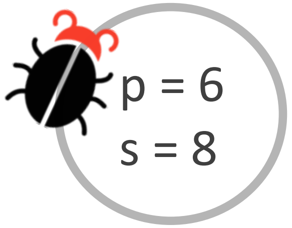
$s \mapsto s$

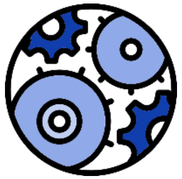
$c \mapsto p \times s - 48$

Path predicate



SMT-Solver





Formula $F(p, s)$

$c = p \times s - 48 \land c = 0$

[1] James C. King. *Symbolic execution and program testing,* Communications of the ACM, 1976

[2] Cristian Cadar and Sen Koushik. *Symbolic execution for software testing: three decades later.* Communications of the ACM, 2013

SE for constant-time via self-composition [1]

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

Symbolic Execution

Formula $\mathrm{F}(p, s)$

$c = p \times s - 48 \wedge c = 0$

Can c = 0 depend on s?

[1] Barthe G, D'Argenio PR, Rezk T. Secure Information Flow by Self-Composition. Computer Security Foundations Workshop 2004

SE for constant-time via self-composition [1]

```
foo(public p, secret s){
   c := p * s - 48;
   if(c = 0) error();
   else return s/c;
}
```

Symbolic Execution

Formula $\mathrm{F}(p, s)$

$$c = p \times s - 48 \wedge c = 0$$

Can c = 0 depend on s?

Self-composition: $\mathrm{F}(p, s, p', s')$

$$p = p' \wedge \begin{matrix} c = p \times s - 48 \\ c' = p' \times s' - 48 \end{matrix} \wedge c = 0 \neq c' = 0$$

SMT-Solver

p = 6, s = 8
p’ = 6, s’=1

[1] Barthe G, D'Argenio PR, Rezk T. Secure Information Flow by Self-Composition. Computer Security Foundations Workshop 2004

SE for constant-time via self-composition

Limitations

- Whole formula is duplicated $\mathrm{F}(p, s, p', s')$
- High number of insecurity queries to the solver

Relational Symbolic Execution to overcome these limitation

Better approach: Relational SE [1,2]

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

Symbolic store

Sharing in SE 👍

p $\mapsto < p >$

s $\mapsto < s \mid s' >$

c $\mapsto < p \times s - 48 \mid p \times s' - 48 >$

[1] "Shadow of a doubt", Palikareva, Kuchta, and Cadar 2016
[2] "Relational Symbolic Execution", Farina, Chong, and Gaboardi 2017

Better approach: Relational SE [1,2]

```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

Symbolic store

Sharing in SE 👍

$$\texttt{p} \mapsto <p>$$

$$\texttt{s} \mapsto <s \mid s'>$$

$$\texttt{c} \mapsto <p \times s - 48 \mid p \times s' - 48>$$

Relational formula: $F(p, s, s')$

$$\begin{array}{l} c = p \times s - 48 \\ c' = p \times s' - 48 \end{array} \wedge c = 0 \neq c' = 0$$

SMT-Solver

p = 6

s = 8   s'=1

[1] "Shadow of a doubt", Palikareva, Kuchta, and Cadar 2016

[2] "Relational Symbolic Execution", Farina, Chong, and Gaboardi 2017

Better approach: Relational SE [1,2]

```
foo(public p, secret s){
  c := p - 48;
  if(c = 0) error();
  else return s/c;
}
```

Symbolic store

p $\mapsto < p >$

s $\mapsto < s \mid s' >$

c $\mapsto < p - 48 >$

[1] “Shadow of a doubt”, Palikareva, Kuchta, and Cadar 2016
[2] “Relational Symbolic Execution”, Farina, Chong, and Gaboardi 2017

Better approach: Relational SE [1,2]

```
foo(public p, secret s){
  c := p - 48;
  if(c = 0) error();
  else return s/c;
}
```

Symbolic store

p $\mapsto < p >$

s $\mapsto < s \mid s' >$

c $\mapsto < p - 48 >$

Spared query !

Sharing in SE 👍
Secret tracking 👍

[1] "Shadow of a doubt", Palikareva, Kuchta, and Cadar 2016
[2] "Relational Symbolic Execution", Farina, Chong, and Gaboardi 2017

Limitations of RelSE

Problem:

- Memory = symbolic array $< \mu \mid \mu' >$
- Duplicate load operations $< select\ \mu\ (esp - 4) \mid select\ \mu'(esp - 4) >$
- Many loads in binary code ☹

RelSE is inefficient at binary-level
RelSE cannot efficiently model speculations

PART 1

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

MAY 18-20, 2020

41st IEEE Symposium on Security and Privacy

Many verification tools for constant-time but…

	Target	Bounded-Verif	Bug-Finding
CT-SC [1] & CT-AI [2]	C	✓+	✗
Casym [4] & CT-Verif [3]	LLVM	✓+	✗
CacheAudit [5]	**Binary**	✓+	✗
CacheD [6]	**Binary**	✗	✓

C/LLVM analysis might miss constant-time violations ☹

+ Full proof

[1] J. Bacelar Almeida, M. Barbosa, J. S. Pinto, and B. Vieira, “Formal verification of side-channel countermeasures using self-composition,” in Science of Computer Programming, 2013
[2] S. Blazy, D. Pichardie, and A. Trieu, “Verifying Constant-Time Implementations by Abstract Interpretation,” in ESORICS, 2017
[3] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, “Verifying Constant-Time Implementations.,” in USENIX, 2016
[4] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir, “CaSym: Cache aware symbolic execution for side channel detection and mitigation,” in IEEE SP, 2019
[5] G. Doychev and B. Köpf, “Rigorous analysis of software countermeasures against cache attacks,” in PLDI, 2017
[6] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, “CacheD: Identifying cache-based timing channels in production software,” in USENIX, 2017

Many verification tools for constant-time but…

	Target	Bounded-Verif	Bug-Finding
CT-SC [1] & CT-AI [2]	C	✓+	✗
Casym [4] & CT-Verif [3]	LLVM	✓+	✗
CacheAudit [5]	**Binary**	✓+	✗
CacheD [6]	**Binary**	✗	✓
Binsec/Rel	**Binary**	✓	✓

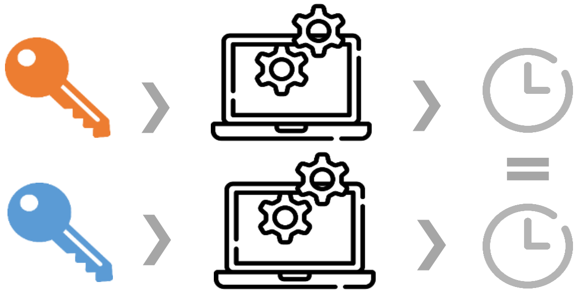
C/LLVM analysis might miss constant-time violations ☹

+ Full proof

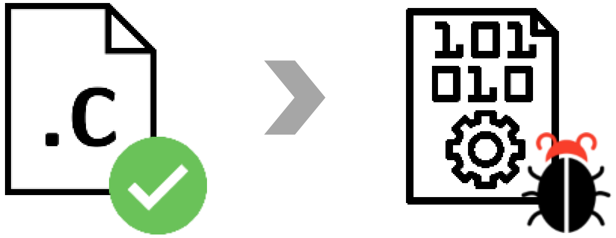
[1] J. Bacelar Almeida, M. Barbosa, J. S. Pinto, and B. Vieira, "Formal verification of side-channel countermeasures using self-composition," in Science of Computer Programming, 2013
[2] S. Blazy, D. Pichardie, and A. Trieu, "Verifying Constant-Time Implementations by Abstract Interpretation," in ESORICS, 2017
[3] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying Constant-Time Implementations.," in USENIX, 2016
[4] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir, "CaSym: Cache aware symbolic execution for side channel detection and mitigation," in IEEE SP, 2019
[5] G. Doychev and B. Köpf, "Rigorous analysis of software countermeasures against cache attacks," in PLDI, 2017
[6] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "CacheD: Identifying cache-based timing channels in production software," in USENIX, 2017

Challenges SE for constant-time analysis

Property of 2 executions



Not necessarily preserved by compilers



RelSE

SE for pairs of traces with sharing

+

Binary-analysis

Reason explicitly about memory

= **Does not scale ☹**

Binary-level RelSE

On-the-fly read-over-write

- Relational expressions in memory
- Builds on *read-over-write* [1]
- Simplify loads on-the-fly

[1] Farinier B, David R, Bardin S, Lemerre M. *Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing.* LPAR 2018

Binary-level RelSE

On-the-fly read-over-write

- Relational expressions in memory
- Builds on *read-over-write* [1]
- Simplify loads on-the-fly

Memory as the history of stores.

$$< \mu \mid \mu' >$$

$$esp - 4 \;\big|\; < p >$$

$$esp - 8 \;\big|\; < s \mid s' >$$

[1] Farinier B, David R, Bardin S, Lemerre M. *Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing.* LPAR 2018

Binary-level RelSE

On-the-fly read-over-write

- Relational expressions in memory
- Builds on *read-over-write* [1]
- Simplify loads on-the-fly

Memory as the history of stores.

$< \mu \mid \mu' >$

$esp - 4 \mid < p >$

$esp - 8 \mid < s \mid s' >$

Example.

`load esp-4` returns $< p >$ instead of

$< select\ \mu\ (esp - 4) \mid select\ \mu' (esp - 4) >$

[1] Farinier B, David R, Bardin S, Lemerre M. *Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing.* LPAR 2018

Dedicated optimizations for constant-time

Untainting

Use solver response to transform $< a \mid a' >$ to $< a >$

- Better sharing
- Better secret tracking

Fault-Packing

Pack queries along basic-blocks

- Reduces number of queries
- Useful for constant-time analysis (many queries)

Formalization and theorems

Formal proofs

Theorem: Correct for Bug-Finding

$$\exists\ s_0 \rightsquigarrow^{k} s_k \not\rightsquigarrow\!\!\!\!\twoheadrightarrow \implies \exists\ c_0 \simeq_l c_0' \ \wedge \ \begin{matrix} c_0 \xrightarrow[t]{}^{k+1} c_{k+1} \\ c_0' \xrightarrow[t']{}^{k+1} c_{k+1}' \end{matrix} \ \wedge \ t \neq t'$$

Theorem: Correct for Bounded-Verification

$$\forall\ \neg(s_0 \rightsquigarrow^{k} s_k \not\rightsquigarrow\!\!\!\!\twoheadrightarrow) \implies \forall\ c_0 \simeq_l c_0' \ \wedge \ \begin{matrix} c_0 \xrightarrow[t]{}^{k} c_{k} \\ c_0' \xrightarrow[t']{}^{k} c_{k}' \end{matrix} \implies t = t'$$

+ Generalization to other leakage models

Experimental evaluation

Binsec/Rel

https://github.com/binsec/rel

Ablation study: Binsec/Rel vs. vanilla RelSE

	Instructions	Instructions / sec	Time	Timeouts
RelSE	349k	**6.2**	**15h47**	**13**
Binsec/Rel	23M	**4429**	**1h26**	**0**

Total on 338 cryptographic samples (secure & insecure)
Timeout set to 1h

Binsec/Rel 700× faster than RelSE
No timeouts even on large programs (e.g. donna)

Preservation of constant-time by compilers

Prior *manual* study on constant-time bugs introduced by compilers [1]

- We *automate* this study with Binsec/Rel
- We extend this study:

  29 new functions & 2 gcc compilers + clang v7.1 & ARM binaries

Total
408 binaries

- gcc –O0 can introduce violations in programs
- clang backend passes introduce violations in programs deemed secure by constant-time verification tools for llvm
- + other fun facts in thesis

[1] “What you get is what you C”, Simon, Chisnall, and Anderson 2018

PART 2

NDSS SYMPOSIUM/2021

Haunted RelSE: detect Spectre vulnerabilities

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

Sequential 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

Sequential 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

Sequential 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

Sequential 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

Sequential 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

Sequential 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)
```

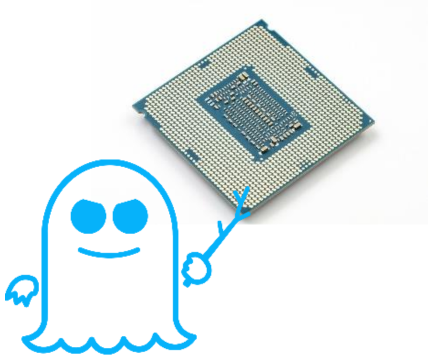
leak(init_mem[a])

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

Constant-time verification in the Spectre era

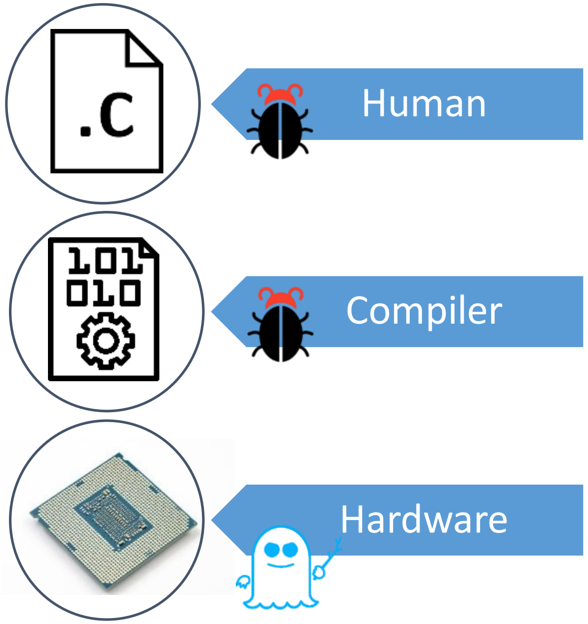
Not easy to write constant-time programs

- Sequence of instructions executed
  - → First timing attacks by Paul Kocher, 1996
- Memory accesses
  - → Cache attacks, 2005
- Processors optimizations
  - → Spectre attacks, 2018



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

Multiple failure points



Modelling speculative semantics

Litmus tests (328 instrutions):

- Sequential semantics
  → **14 paths**
- Speculative semantics (Spectre-STL)
  → **37M paths**

Modelling all transient paths explicitly is intractable

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 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 violations ☹

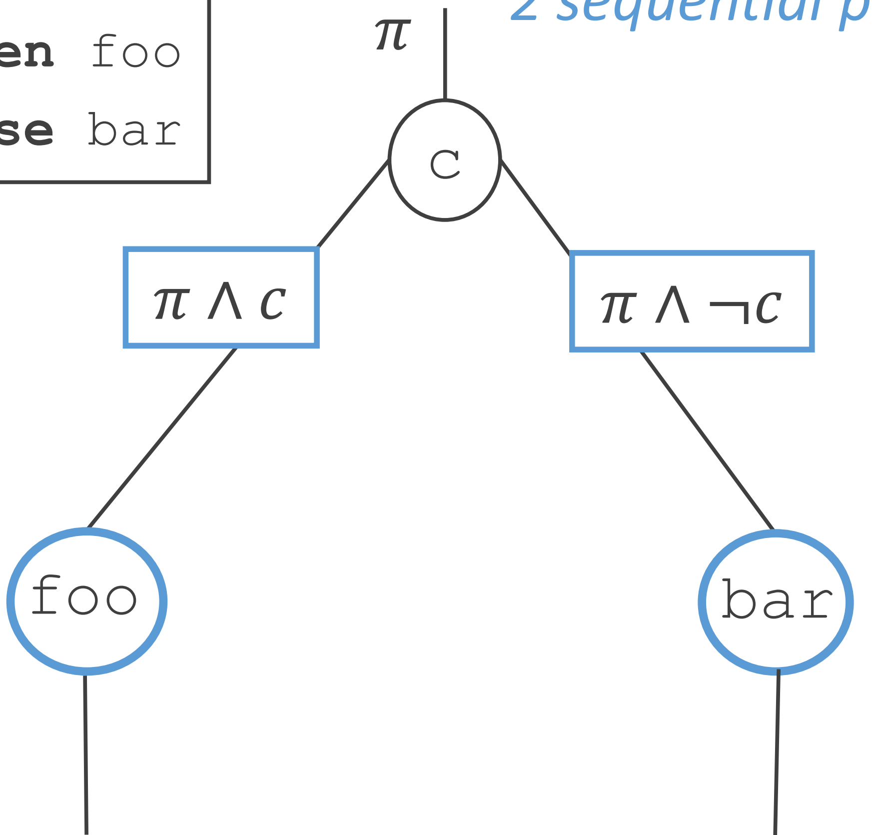
[1] G. Wang, et al "KLEESpectre: Detecting Information Leakage through Speculative Cache Atttacks via Symbolic Execution", ACM Trans. Softw. Eng. Methodol., vol. 29, no. 3, 2020.
[2] S. Guo, Y. Chen, P. Li, Y. Cheng, H. Wang, M. Wu, and Z. Zuo, "SpecuSym: Speculative Symbolic Execution for Cache Timing 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.

Haunted RelSE

Explicit RelSE for Spectre PHT

Symbolic execution with sequential semantics

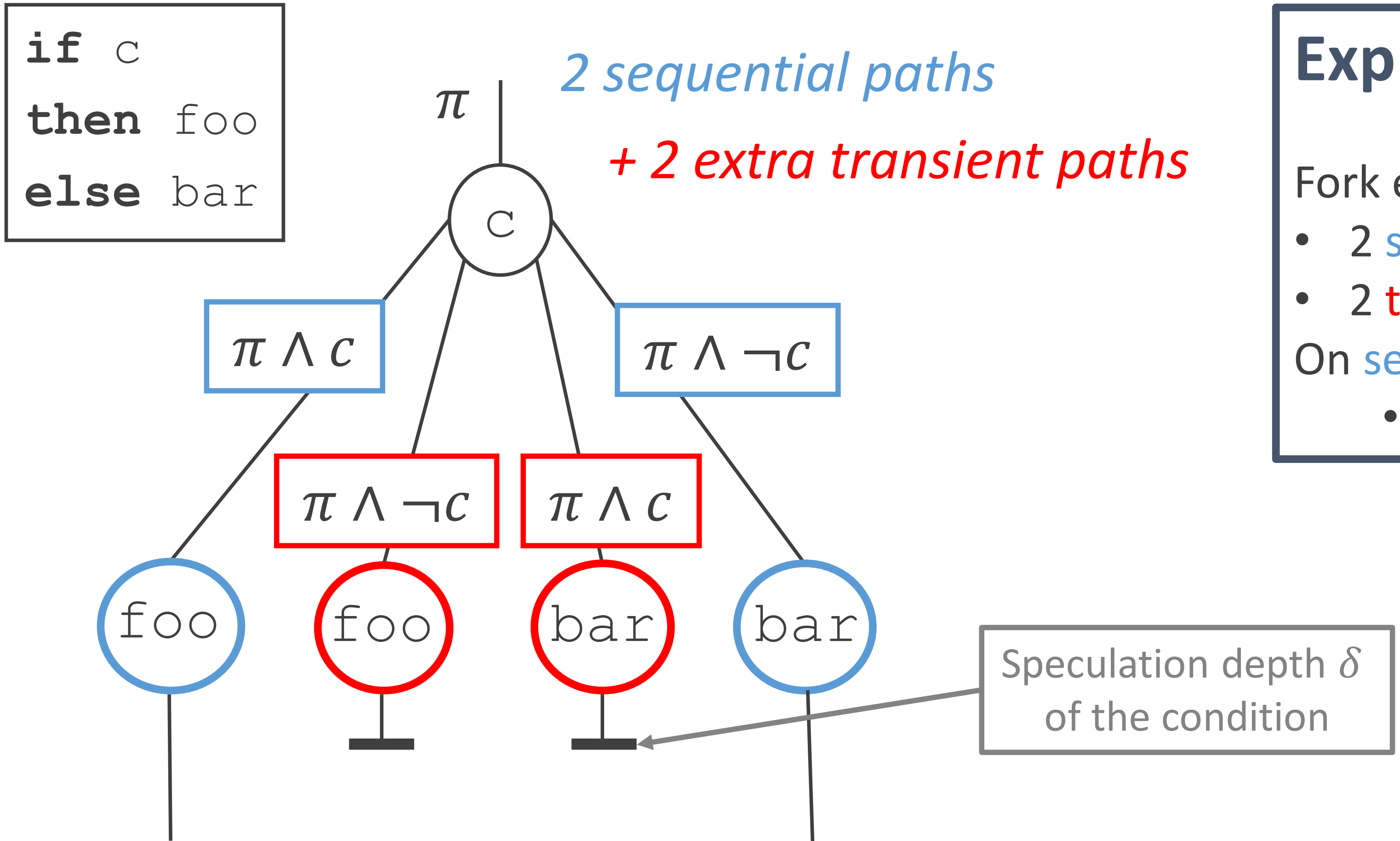
```
if c
then foo
else bar
```



2 sequential paths

Explicit RelSE for Spectre PHT

Spectre-PHT. Conditional branches can be executed speculatively



Explicit RelSE.

Fork execution into 4 at conditionals:

- 2 sequential branches
- 2 transient branches

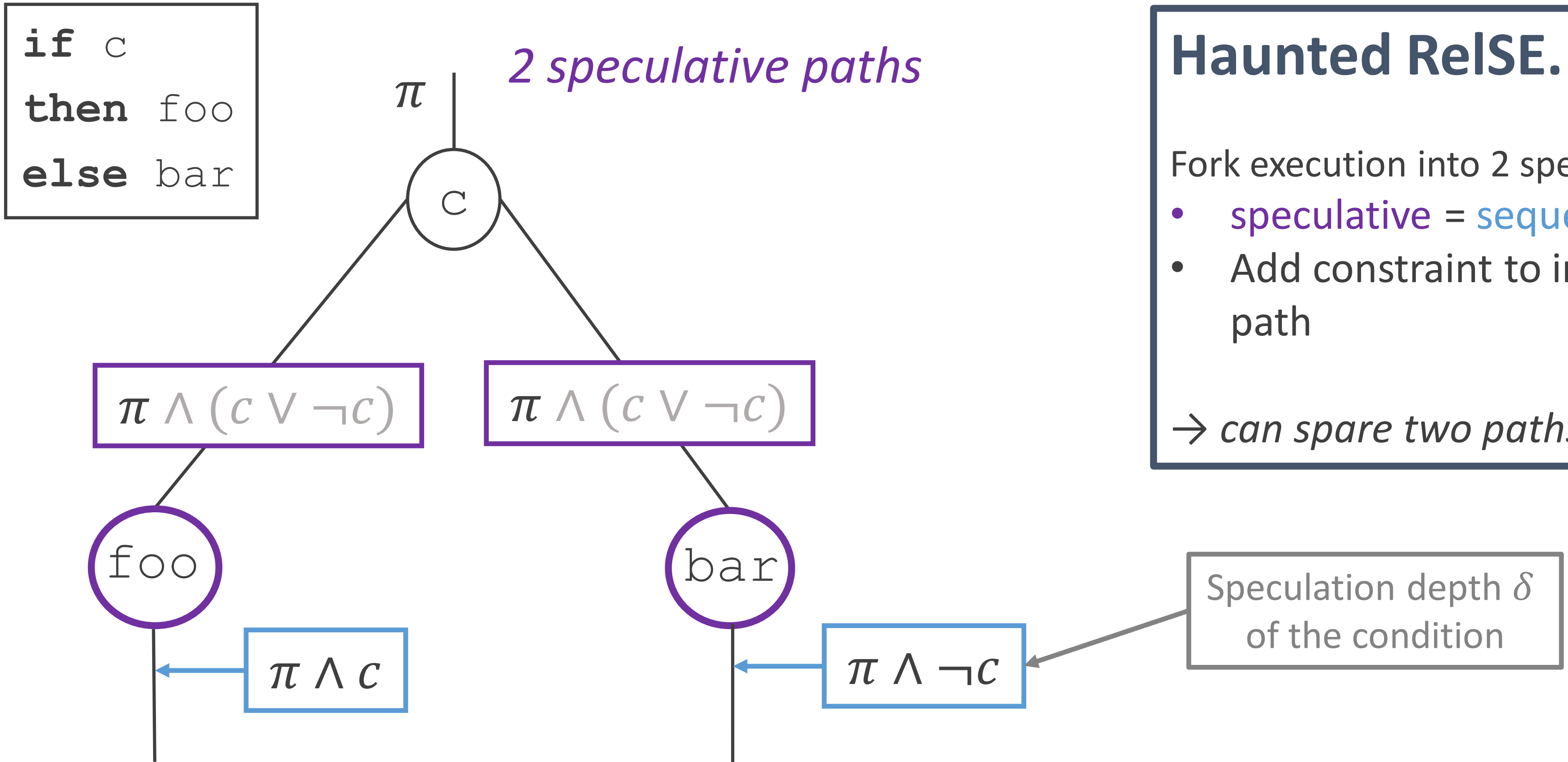
On sequential and transient branches:

- Verify no secret can leak.

(e.g. KLEESpectre)

Haunted RelSE for Spectre PHT

Spectre-PHT. Conditional branches can be executed speculatively



Haunted RelSE.

Fork execution into 2 speculative paths:

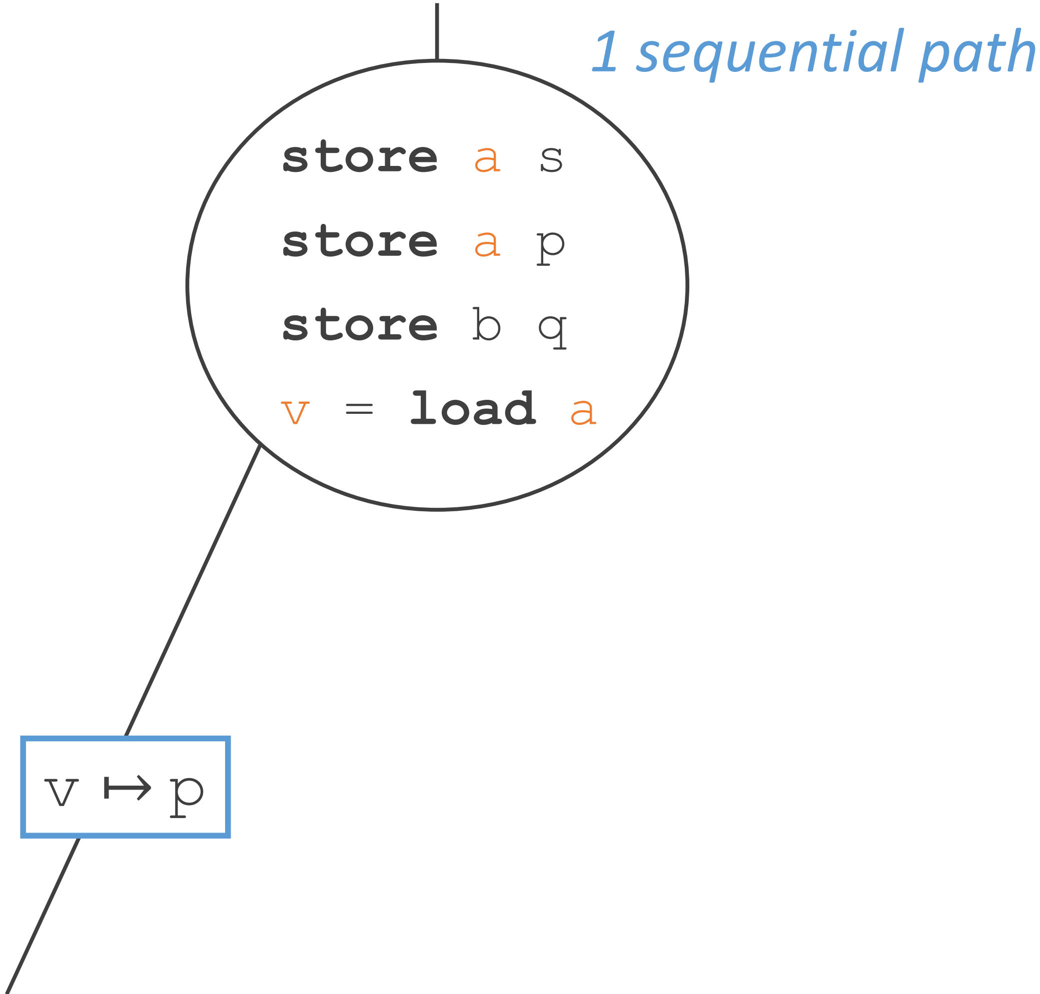
- speculative = sequential ∨ transient
- Add constraint to invalidate transient path

→ *can spare two paths at conditionals*

Explicit RelSE for Spectre-STL

```
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

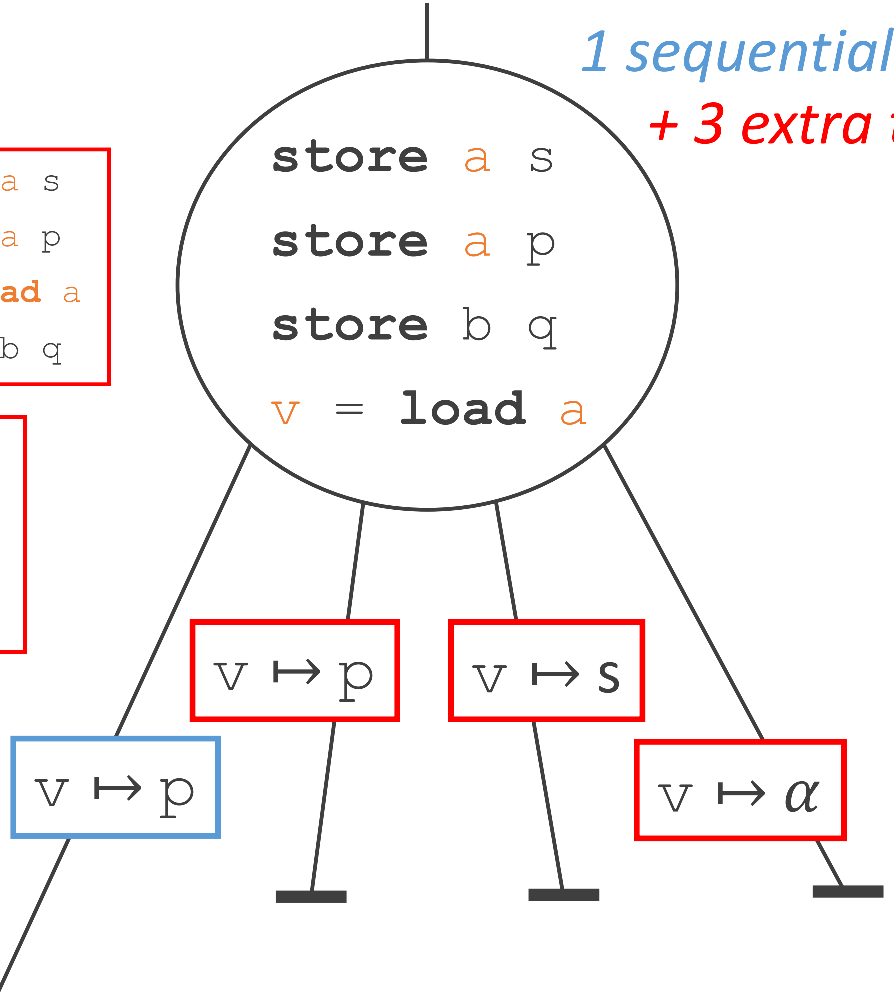
```
store a s
store a p
store b q
v = load a
```

```
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 sequential path

+ 3 extra transient paths

Explicit RelSE.

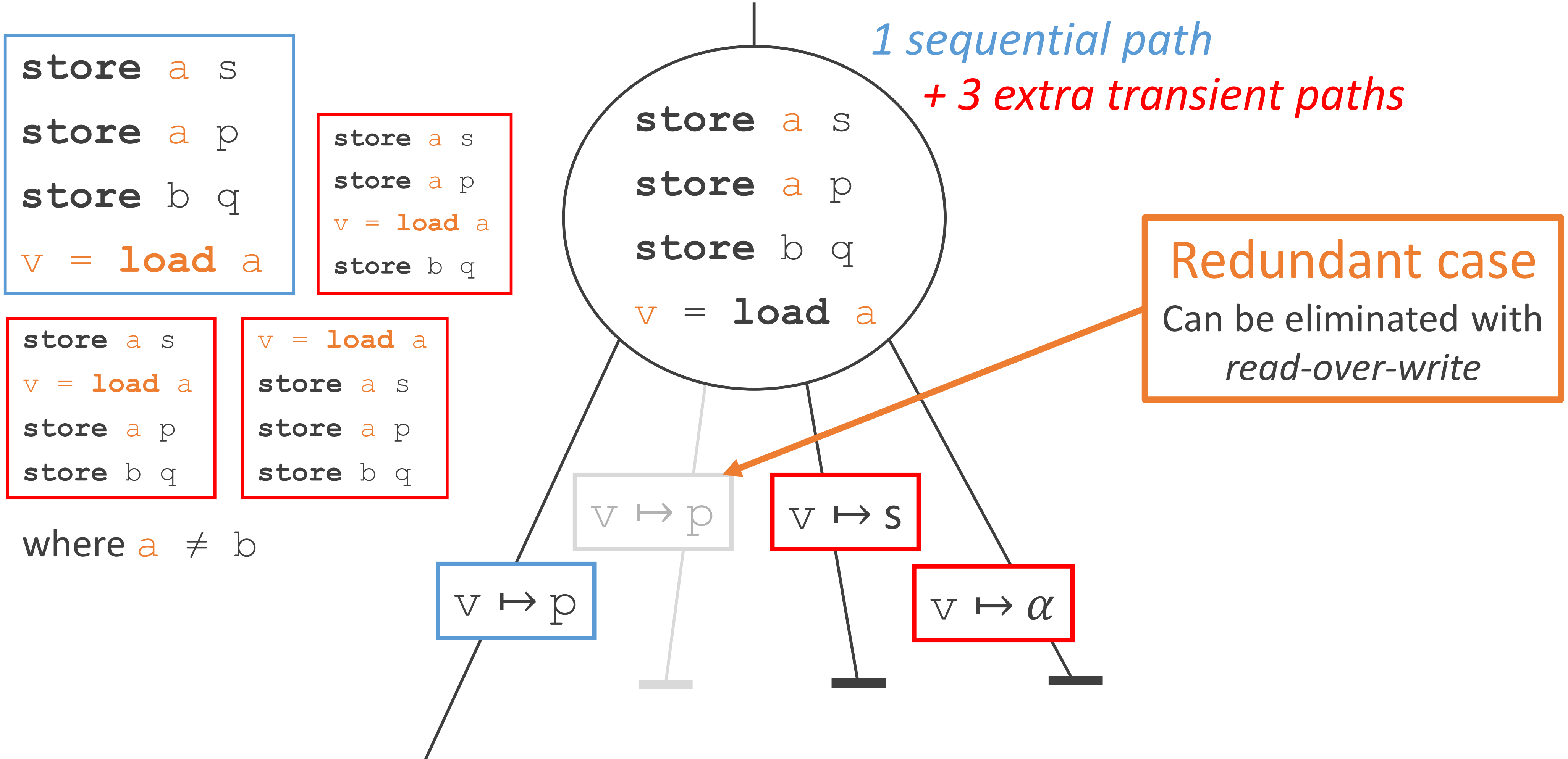
At load instructions: fork execution for each load/store interleaving.

→ Path explosion

(e.g. Pitchfork)

Haunted RelSE for Spectre-STL

Spectre-STL. Loads can speculatively bypass prior stores



Haunted 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* β_0 *then* α *else (ite* β_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 β_0, β_1

Experimental evaluation

Binsec/Haunted

https://github.com/binsec/haunted

Experimental evaluation

Benchmark

Litmus tests: Spectre-PHT = Paul Kocher standard, Spectre-STL = **new** set of litmus tests

Cryptographic primitives: tea, donna, Libsodium secretbox, OpenSSL ssl3-digest-record & mee-cdc-decrypt

Effective on real code?

→ *Spectre-PHT* ☺ & *Spectre-STL* 😐

Haunted RelSE vs. Explicit RelSE?

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

Comparison against KLEESpectre & Pitchfork

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

PHT	STL
Litmus:	
Paths: 1546 → 370	Paths: 93M → 42
Time: 3h → 15s	Coverage: 2k → 17k
Libsodium + OpenSSL:	Timeouts: 15 → 8
Coverage: 2273 → 8634	Bugs: 22 → 148
Total:	
Timeouts: 5 → 1	

Weakness of index-masking countermeasure

+ Position independent code

Weakness of Spectre-PHT countermeasure

Program vulnerable to Spectre-PHT

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

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

Weakness of Spectre-PHT countermeasure

Index masking countermeasure

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

Weakness of Spectre-PHT countermeasure

Index masking countermeasure

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

Compiled version with gcc –O0 –m32

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

- Store + load masked index
- Store may be bypassed with Spectre-STL !

Weakness of Spectre-PHT countermeasure

Index masking countermeasure

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

Compiled version with gcc –O0 –m32

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

- Store + load masked index
- 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");`

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: bug-finding & bounded-verif. of constant-time & secret-erasure at binary-level
- Analysis of crypto libraries at binary-level: constant-time llvm may yield vuln. binary

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

Future work

Extensible framework: check property preservation by compilers:

New countermeasures (lfence, speculative load hardening, Spectre RSB/BTB)

Exploitability: Too conservative property? load ebp-4 cannot bypass store ebp-4

General noninterference: challenge → model diverging paths

Hardware extension for secure speculation:

Formal design and security proof of a hardware monitor

Publications

Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level

Lesly-Ann Daniel, Sébastien Bardin, Tamara Rezk

IEEE Symposium on Security and Privacy (SP), 2020

Hunting the Haunter—Efficient Relational Symbolic Execution for Spectre with Haunted RelSE

Lesly-Ann Daniel, Sébastien Bardin, Tamara Rezk

Network and Distributed System Security Symposium (NDSS), 2021

Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure

Lesly-Ann Daniel, Sébastien Bardin, Tamara Rezk

[Major revision] ACM Transactions on Privacy and Security (TOPS), 2021

Reflections on the Experimental Evaluation of a Binary-Level Symbolic Analyzer for Spectre

Lesly-Ann Daniel, Sébastien Bardin, Tamara Rezk

[Under review] Learning from Authoritative Security Experiments Results (Proceedings LASER workshop), 2021

Backup

Beyond Constant-Time

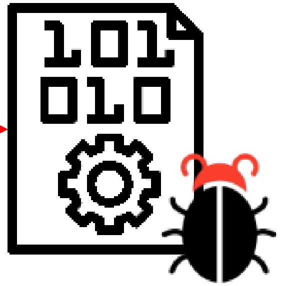
Secret-erasure

```c
void scrub(char * buf, size_t size){
  memset(buf, 0, size );
}

int critical_function () {
  char secret [SIZE];
  read_secret(secret, SIZE);
  process_secret(secret, SIZE); // computation on secret
  scrub(secret, SIZE); // erase secret from memory
  return 0;
}
```

Secret-erasure

```c
void scrub(char * buf, size_t size){
  memset(buf, 0, size );
}

int critical_function () {
  char secret [SIZE];
  read_secret(secret, SIZE);
  process_secret(secret, SIZE); // computation on secret
  scrub(secret, SIZE); // erase secret from memory
  return 0;
}
```

gcc –O2
Dead store elimination pass
removes memset call



- Crucial for cryptographic code
- Property of 2 executions
- Not always preserved by compilers

Generalizing Binary-level RelSE

- Binary-level RelSE parametric in the leakage model
  - → *Symbolic leakage predicate instantiated according to leakage model*
  - → *For IF properties restricting to pairs of traces following same path*

$$\frac{\mathbb{P}[l] = \texttt{halt} \qquad \boxed{\tilde{\lambda}_{\perp}(\pi, \widehat{\mu})}}{(l, \rho, \widehat{\mu}, \pi) \rightsquigarrow (l, \rho, \widehat{\mu}, \pi)}$$

- New leakage model + property for capturing secret-erasure
  - → *Leaks value of all store operations that are not overwritten*
  - → *Forbids secret dependent control-flow*

- Adaptation of Binsec/Rel to secret-erasure

Application: Secret-Erasure

New framework to check secret-erasure

Easilly extensible with new *compilers* and new *scrubbing functions*

- We analyze 17 scrubbing functions
- 5 versions of clang & 5 versions of gcc
- 4 optimization levels

Total

680 binaries - 1’20

– Dedicated secure scrubbing functions (e.g. `memset_s`) are secure (but not always available)

– Volatile function pointers can introduce additional register spilling that might break secret-erasure with gcc -O2 and gcc -O3

Haunted RelSE for Spectre-STL

Dynamic speculation depth

Most tools:

Speculate until maximum speculation depth Δ

Dynamic speculation depth:

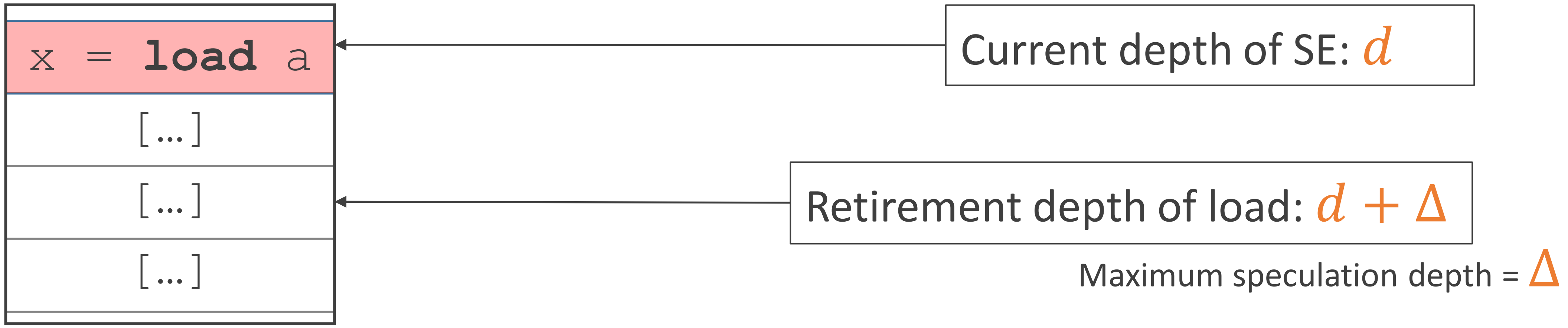
Speculate on conditions only when they depend on memory [1]

→ Model processor more precisely

But what does it means to depend on the memory ?

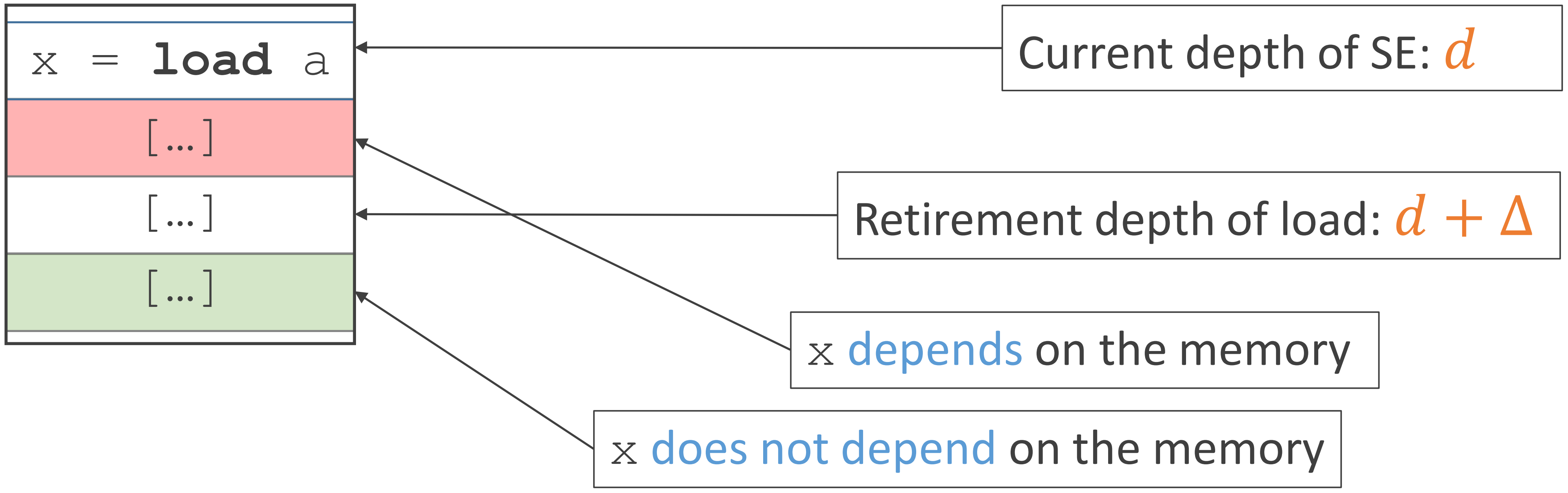
[1] Abstract Interpretation under Speculative Execution, Meng Wu and Chao Wang, PLDI 2019

Dynamic speculation depth



[1] Abstract Interpretation under Speculative Execution, Meng Wu and Chao Wang, PLDI 2019

Dynamic speculation depth



$$x = x^{d+\Delta}$$

Memory dependency depth of x

[1] Abstract Interpretation under Speculative Execution, Meng Wu and Chao Wang, PLDI 2019

Dynamic speculation depth

Speculation depth of conditions = memory dependency depth

`if c > 0` and c $= c^{d'}$ in SE

$\pi := \pi \wedge c > 0$
when $d' \leq$ current depth

Stop speculation

Memory dependency depth of c has been reached

[1] Abstract Interpretation under Speculative Execution, Meng Wu and Chao Wang, PLDI 2019

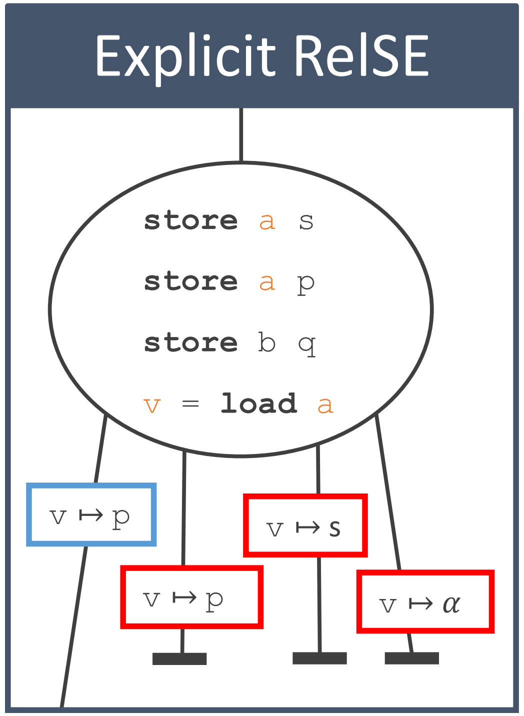
Haunted vs. Explicit RelSE for Spectre-STL

Spectre-STL.
Model multiple load/store interleavings

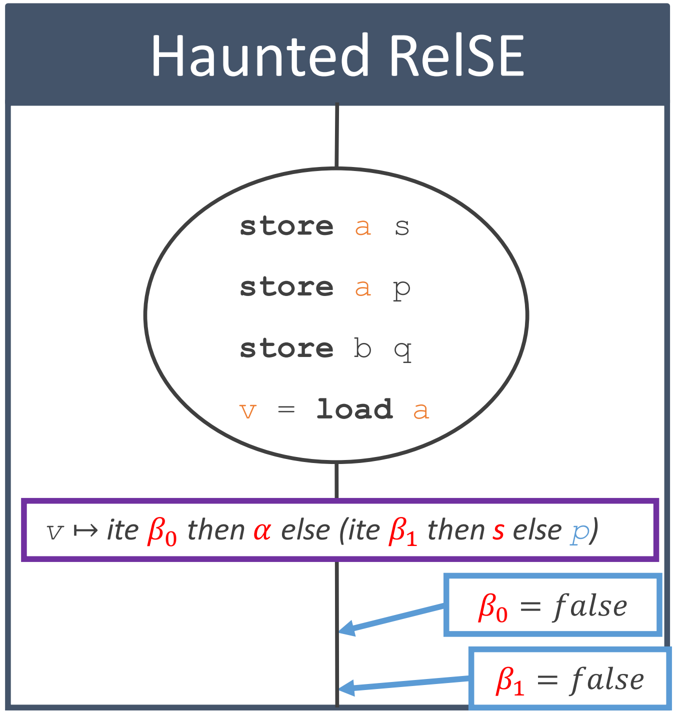
Instead of forking SE:

- Prune redundant values
- Encode values in 1 path

\+ Formal proof:
Equivalence Haunted/Explicit



1 sequential path
+ 3 extra transient paths



1 speculative path

Experimental evaluation: Binsec/Haunted

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

KLEESpectre

Target: LLVM

Spectre-PHT: Explicit

- Litmus tests: 😐 (240× slower)
- Tea & donna: ☺ (≈equivalent)

Take away

Spectre-PHT: ≈ or ↗
Spectre-STL: always ↗

Pitchfork

Target: Binary

Spectre-PHT: Optims

- Litmus tests: (≈equivalent)
- Tea & donna: ☹ (50× slower & TO)

Spectre-STL: Explicit

- Litmus tests: ☹ 6/10 TO (vs. 0 TO)
- Tea & donna: ☹ 10/10 TO (vs. 5 TO + 99 vulns)

Vulnerability introduced by PIC

Position Independent Code & Spectre-STL

PIC: addess global variables = offset from global pointer

Global pointer: set up at the beginning of a function relatively to current location

```
call  __x86_get_pc_thunk_ax
add   eax, 0x9E0FA
mov   edx, (publicarray_size)[eax]
```

- `call __x86_get_pc_thunk_ax` ← eax = current location
- `add eax, 0x9E0FA` ← eax = global pointer
- `mov edx, (publicarray_size)[eax]` ← edx = global variable

```
__x86_get_pc_thunk_ax:
  mov eax, [esp+0]
  retn
```

- `__x86_get_pc_thunk_ax:` ← current location pushed on stack at call
- `mov eax, [esp+0]` ← load current location from stack

Position Independent Code & Spectre-STL

PIC: addess global variables = offset from global pointer

Global pointer: set up at the beginning of a function relatively to current location

```
call  __x86_get_pc_thunk_ax
add   eax, 0x9E0FA
mov   edx, (publicarray_size)[eax]
```

eax = any value

load data from arbitrary @

... leak edx

```
__x86_get_pc_thunk_ax:
   mov eax, [esp+0]
   retn
```

current location pushed on stack at call

load bypasses prior store