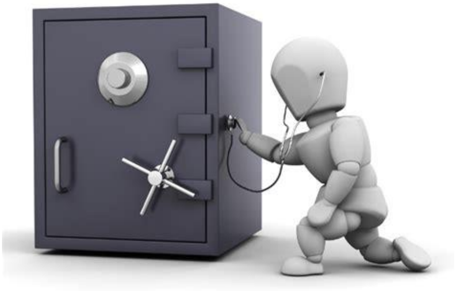
Binsec/Rel Symbolic Binary Analyzer for Security

Application to Constant-Time & Secret-Erasure

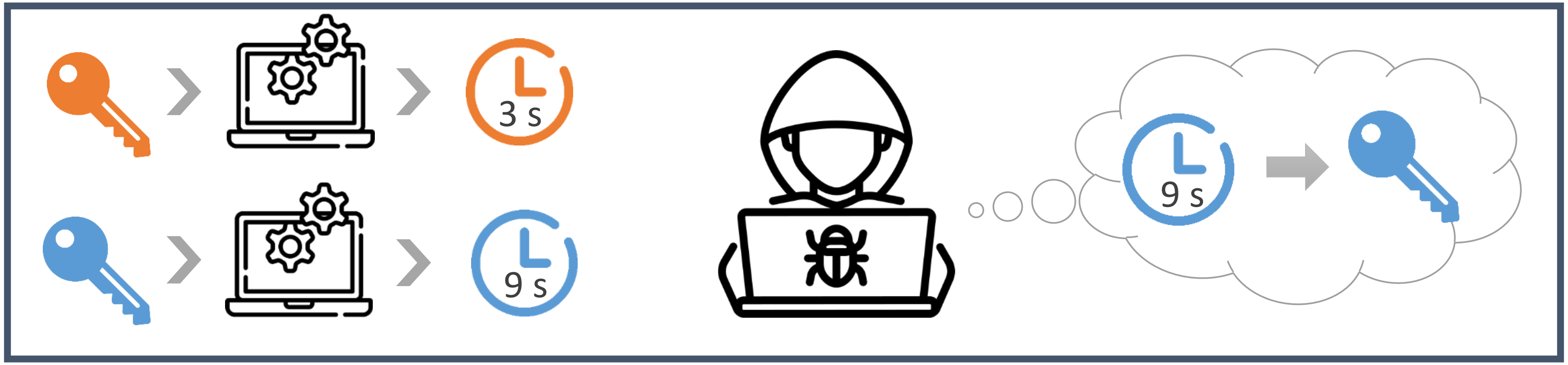
Lesly-Ann Daniel, KU Leuven
Sébastien Bardin, CEA List
Tamara Rezk, INRIA

Timing and Microarchitectural Attacks



Timing and microarchitectural attacks:

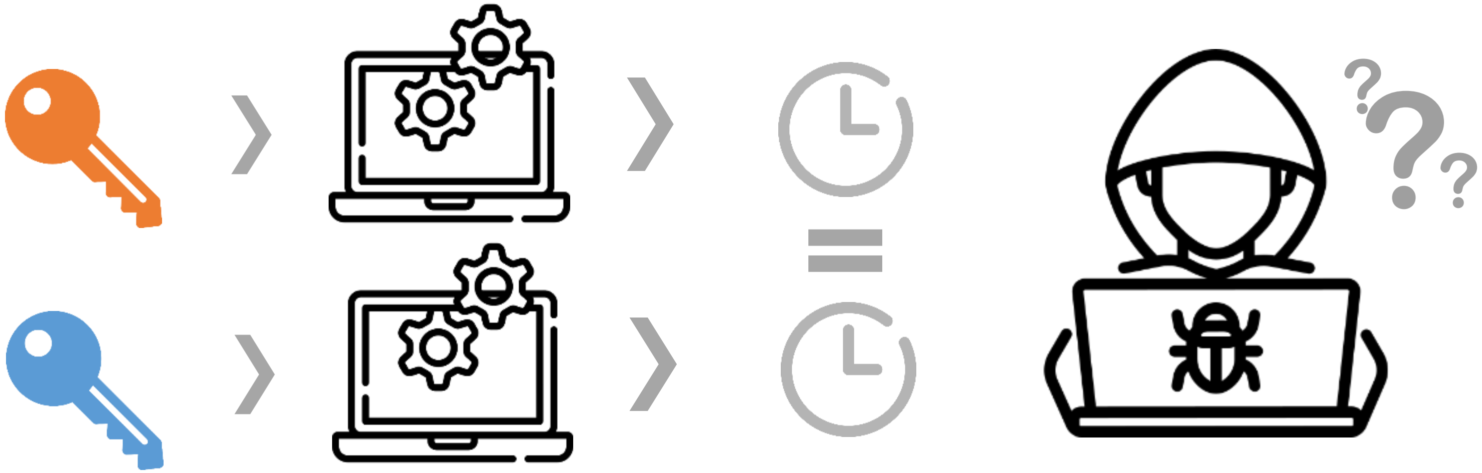
Execution time & microarchitectural state depends on secret data



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

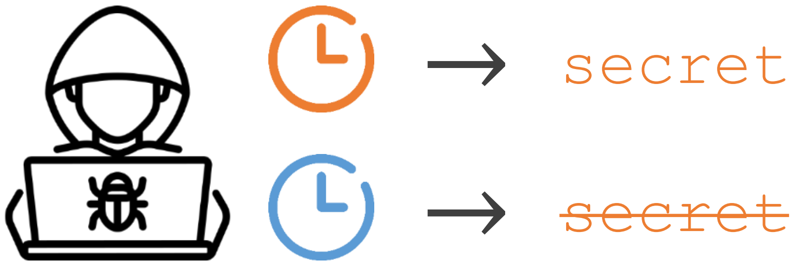


Already used in many cryptographic implementations

What can influence execution time/microarchitecture?

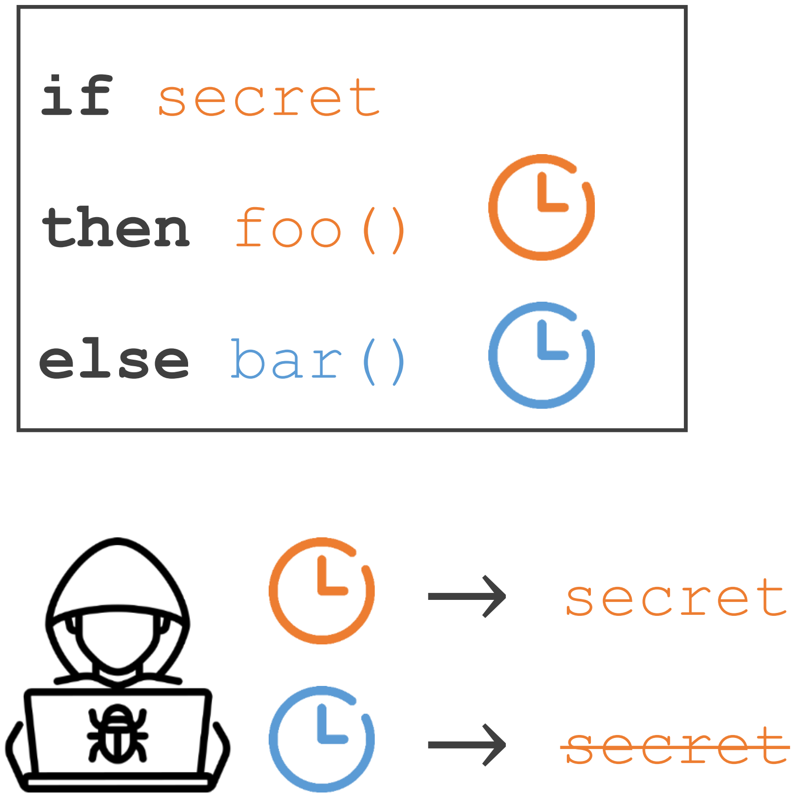
Control Flow

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

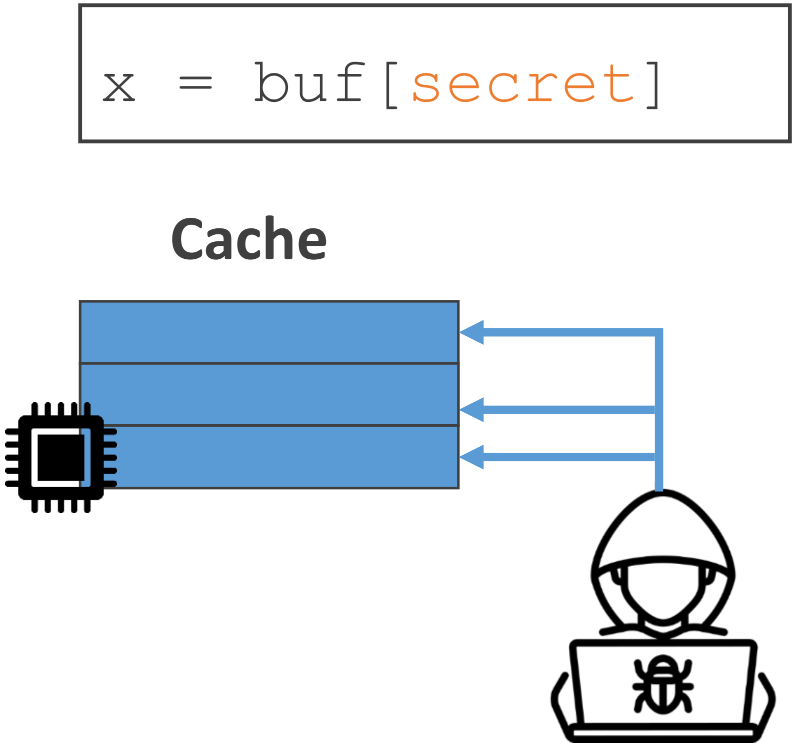


What can influence execution time/microarchitecture?

Control Flow

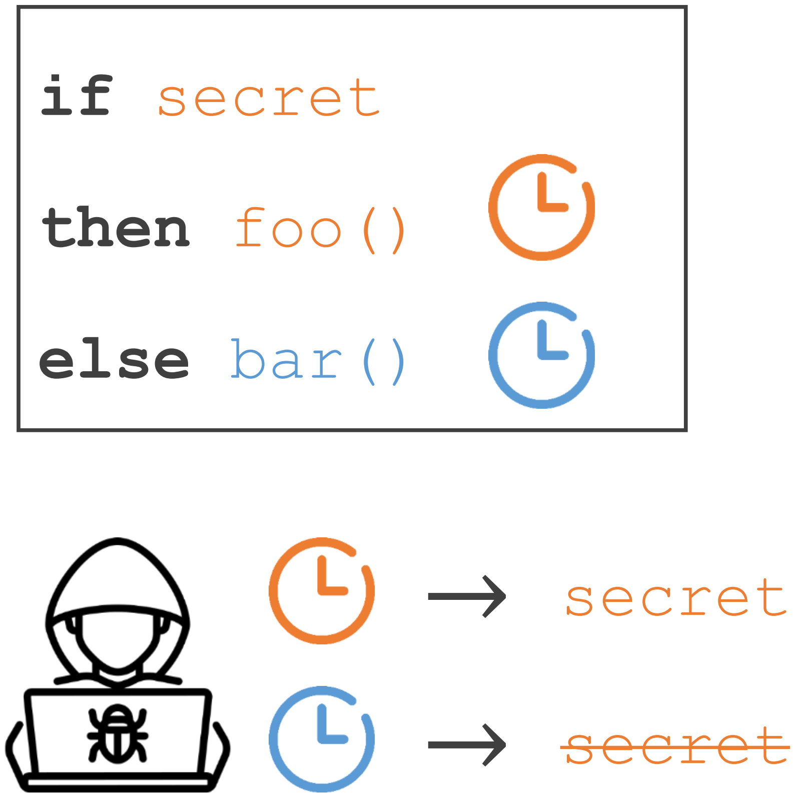


Memory Accesses

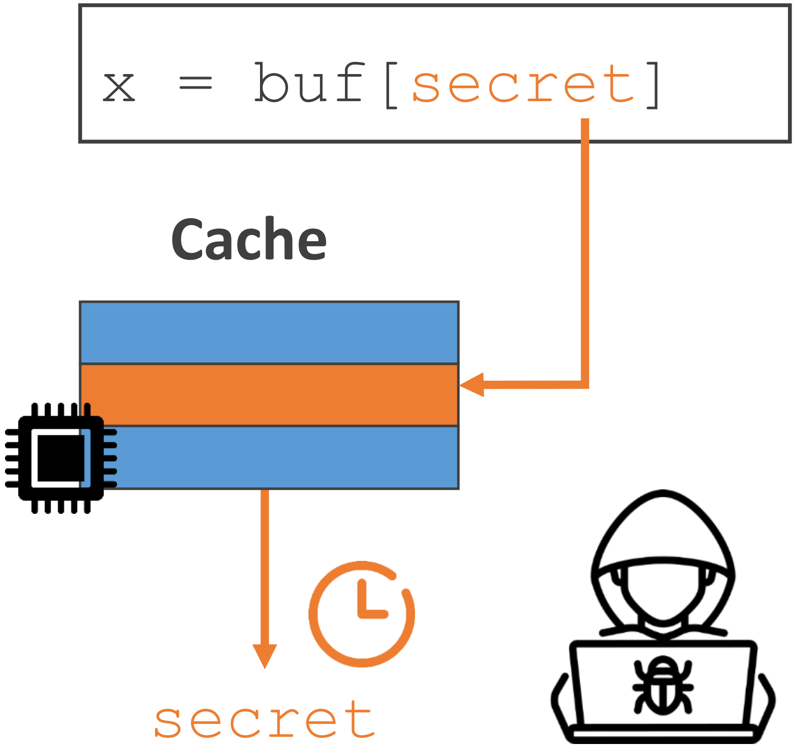


What can influence execution time/microarchitecture?

Control Flow

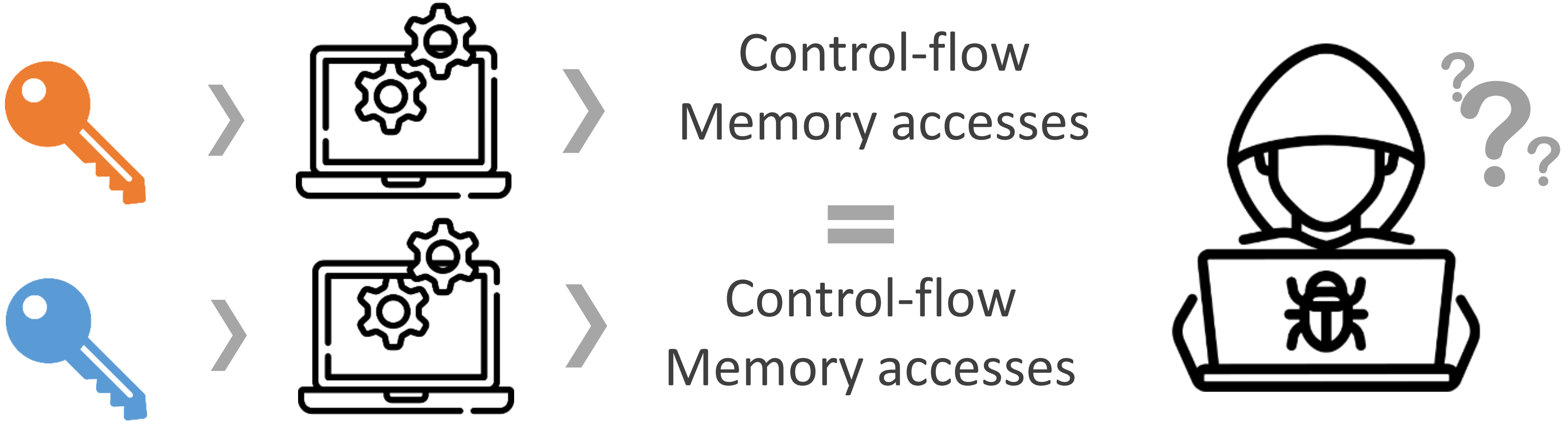


Memory Accesses



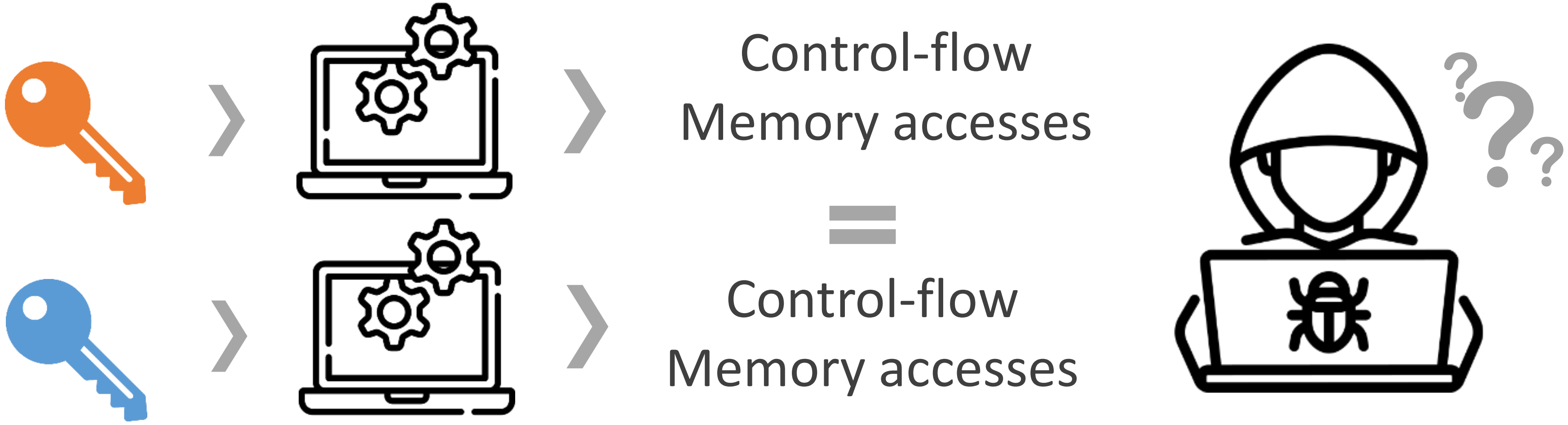
Protect software with constant-time programming

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



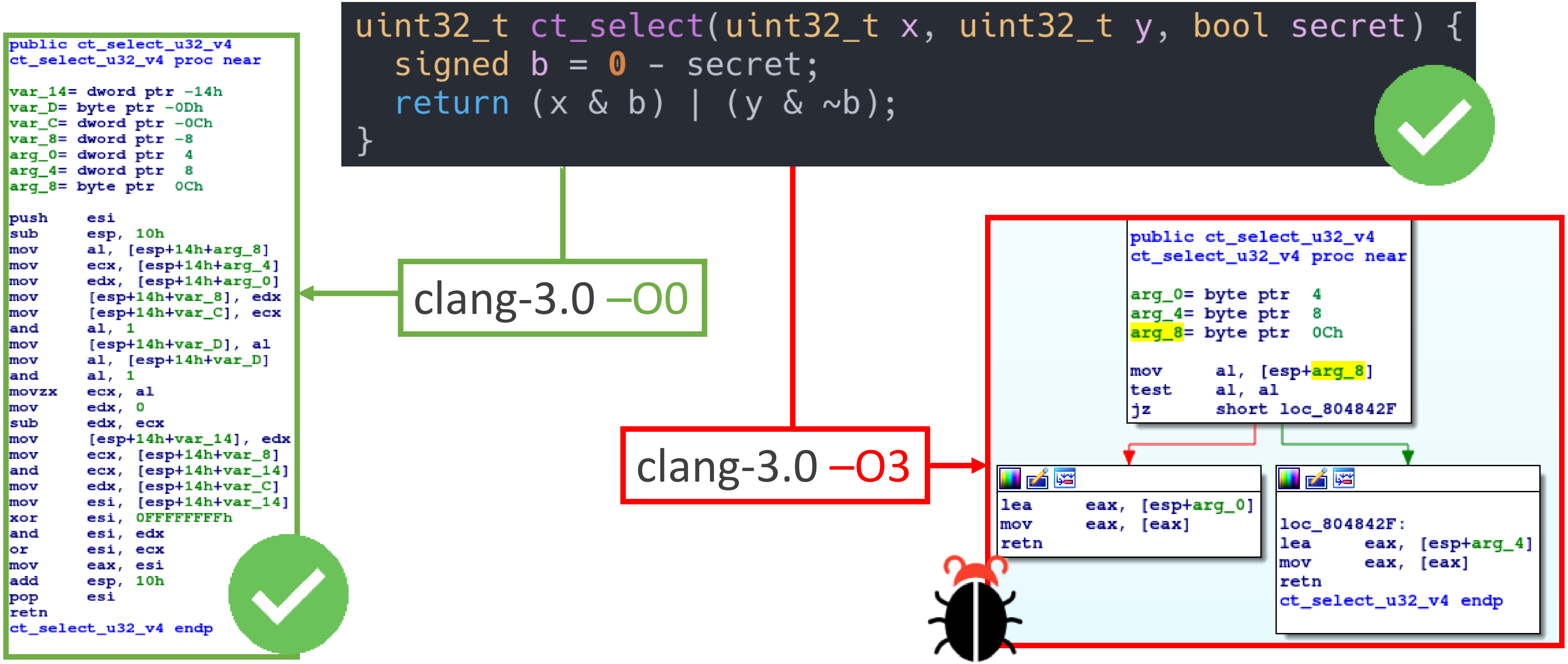
Property relating 2 execution traces (2-hypersafety)

Constant-time is not easy to implement

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

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

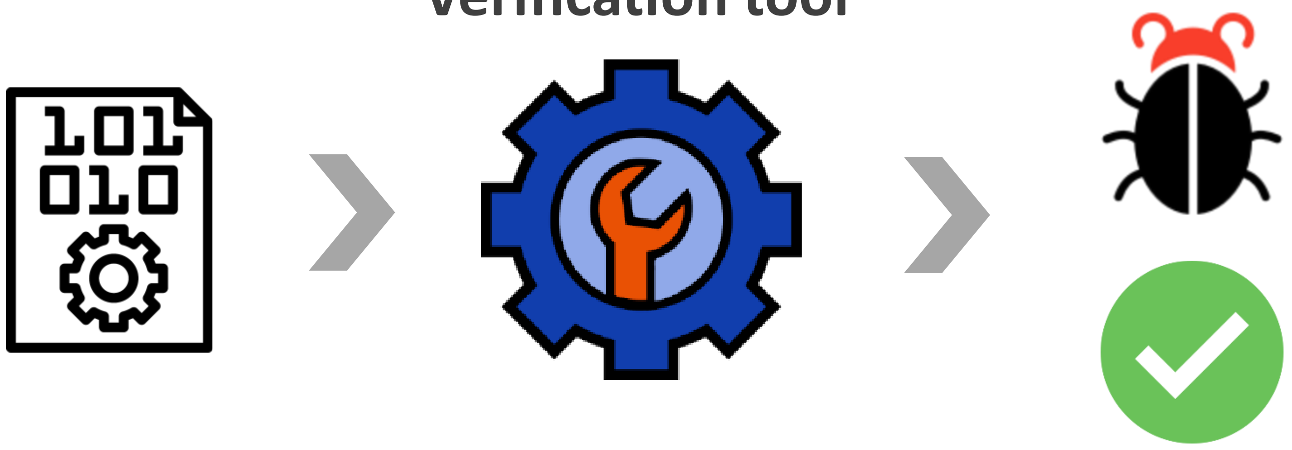
Compilers can break constant-time!



Simon, Laurent, David Chisnall, and Ross Anderson. "What you get is what you C: Controlling side effects in mainstream C compilers.“ 2018 IEEE European Symposium on Security and Privacy (EuroS&P).

Automated program verification

Verification tool



Bug-Finding

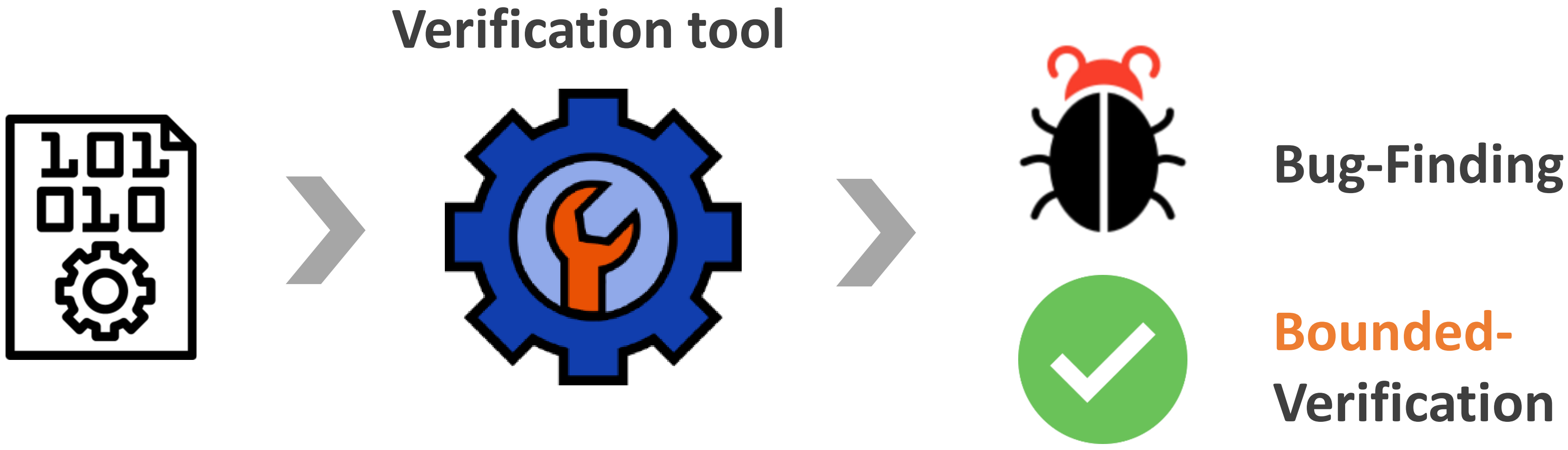
Verification

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



Perfect verification tool:

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

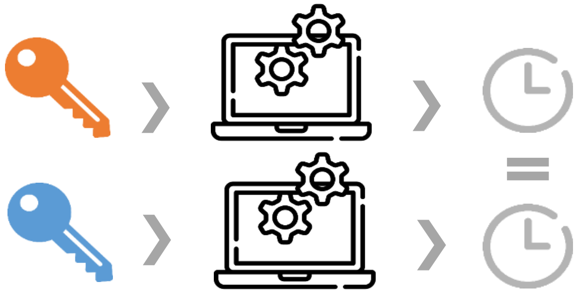
Symbolic Execution (SE)

The KeY Project

BINSEC

Challenges: SE for constant-time analysis

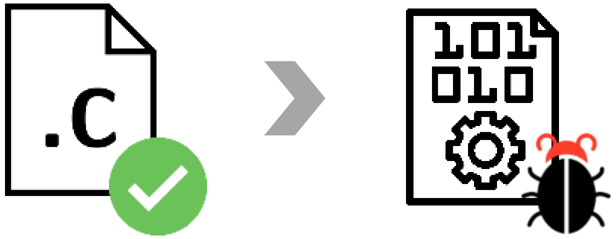
Property of 2 executions



RelSE

SE for pairs of traces with sharing

Not necessarily preserved by compilers



Binary-analysis

Reason explicitly about memory

= **Does not scale ☹**

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

Contributions

- Optimizations: symbolic execution for constant-time + secret-erasure

- Implementation in an open source tools

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

- Application to cryptographic primitives
  - Violations introduced by compilers from verified llvm code

Background: SE for constant-time

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?

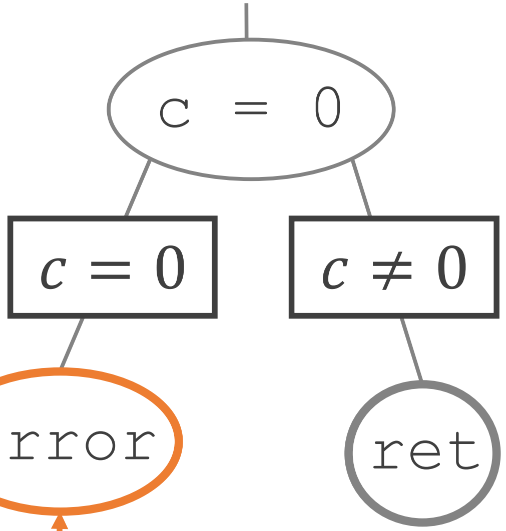
Symbolic store

$p \mapsto p$

$s \mapsto s$

$c \mapsto p \times s - 48$

Path predicate



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

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

[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

Symbolic Execution [1,2]

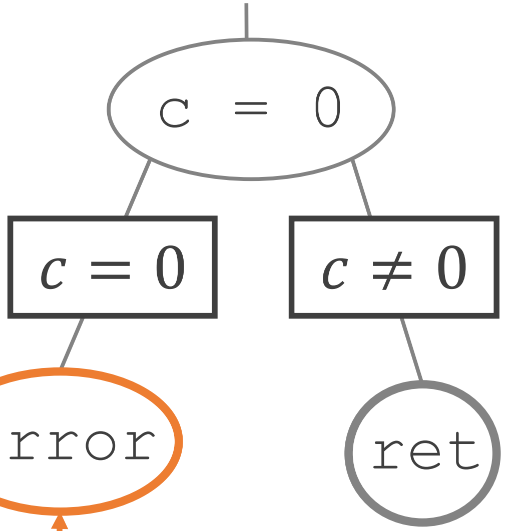
```
foo(public p, secret s){
  c := p * s - 48;
  if(c = 0) error();
  else return s/c;
}
```

Can error be reached?

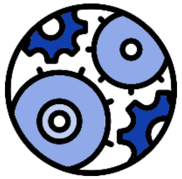
Symbolic store

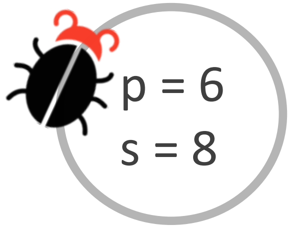
p $\mapsto$ p
s $\mapsto$ s
c $\mapsto$ $p \times s$ - 48

Path predicate



SMT-Solver





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

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

[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

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;
}
```

Can c = 0 depend on s?

Symbolic Execution

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

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

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

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

Can c = 0 depend on s?

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

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

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

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

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

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

[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

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

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

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

SMT-Solver

p = 6
s = 8 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

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.

$< \mu \mid \mu' >$

$esp - 4 \quad < p >$

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

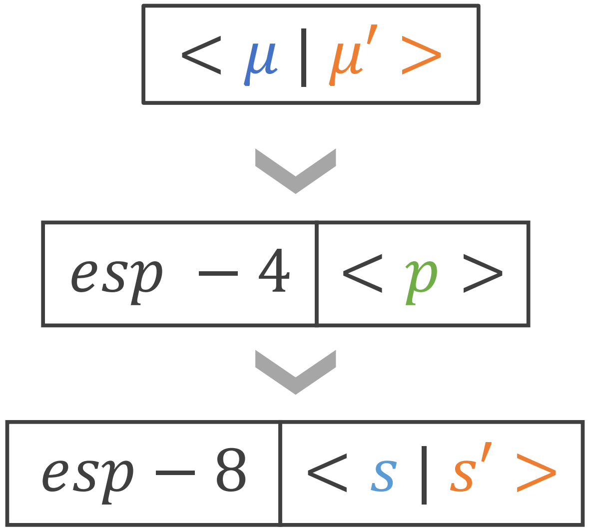
[1] Farinier B, David R, Bardin S, Lemerre M. *Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing.* LPAR 2018

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.



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)

Implementation

Binsec/Rel

https://github.com/binsec/rel

Binsec Framework

BINSEC

Binary

X86-32 (64 incoming)
ARMv7/AARCH64/AMD64
RISC-V 32 (64 incoming)

IR

DBA

Analysis

Symbolic execution
Backward-bounded SE
Relational SE
Abstract interpretation
Concrete interpretation

Helpers

Loader for ELF/PE
Build & simplify formulas
[...]

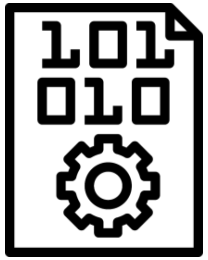
SMT-Solver

Boolector
Bitwuzla
z3, cvc4, yices

https://binsec.github.io/

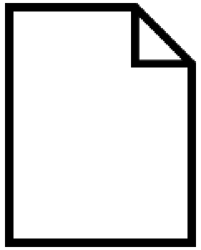
Binsec/Rel

Binary



Easilly specify secrets using dedicated stubs

Initial memory configuration



Concretize esp, .data, canaries, …
Default = symbolic

Binsec/Rel

Exploration module	Insecurity module

• Updates sym. state
• Build path predicate
• Check satisfiability
	• Build insecurity queries
	  • Constant-time
	  • Secret-erasure
	• Ensure unsatisfiability


1


2


3

[1] Only if exhaustive exploration

[2] Violation + counterexample (concrete input)

[3] No violations but non-exhaustive exploration

Limitations

Bounded-verification

- Loop & recursion by unrolling
- Bounded enumeration of jump targets

❯ Can miss violations so Binsec/Rel reports “Unknown”

Implementation

- No dynamic libraries
- No dynamically allocated memory
- No syscall stubs
- No floating-point instructions

Keep in mind: when you *concretize* something (e.g. input size, initial memory, etc.) it might lead to unexplored behavior & *missed violations*

Experimental evaluation

https://github.com/binsec/rel_bench

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:
  - 34 functions
  - i386 / i686 / ARM architectures
  - 6 gcc + 6 clang version
  - 4 optimization level
  - impact of `-x86-cmov-converter` & `if-conversion`

Total
4148 binaries

- clang backend passes introduce violations in programs deemed secure by llvm analyzers
- clang use of `cmov` can introduce secret-dependent memory accesses
- gcc optimizations tend to preserve CT (`if-conversion` can even make secure non-ct source)
- Depend on multiple factors, hard to predict: compiler-support remains the best option

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

Conclusion

Conclusion

- Dedicated optimizations for RelSE at binary-level
  - Sharing between pairs of executions
- Open source tool Binsec/Rel
  - Bug-finding & bounded-verification of constant-time at binary-level
- Analysis of crypto libraries at binary-level
  - Constant-time llvm may yield vulnerable binary

Binsec/Rel

https://github.com/binsec/rel

Extensions

- Binsec/Rel for secret erasure
  - Framework to check preservation of secret-erasure by compilers

    17 scrubbing functions / 10 compilers / 4 opt. level + DSE pass / total = 1156 binaries

    Open source & easy to extend on https://github.com/binsec/rel_bench

- Binsec/Haunted to find Spectre-PHT/STL vulnerabilities

Binsec/Haunted

https://github.com/binsec/haunted

Future of Binsec/Rel

- Binsec/Rel not really maintained but…

- Binsec team is working on integrating Binsec/Rel in Binsec
  - Better (relational) symbolic execution engine
  - Better maintenance
  - Tutorials

- Any feedback is welcome:
  - sebastien.bardin@cea.fr
  - frederic.recoules@cea.fr

BINSEC

https://binsec.github.io/

Backup

Extension: Secret-erasure

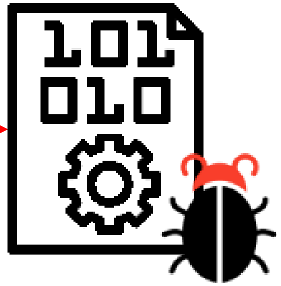
Secret-erasure

```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 + DSE pass

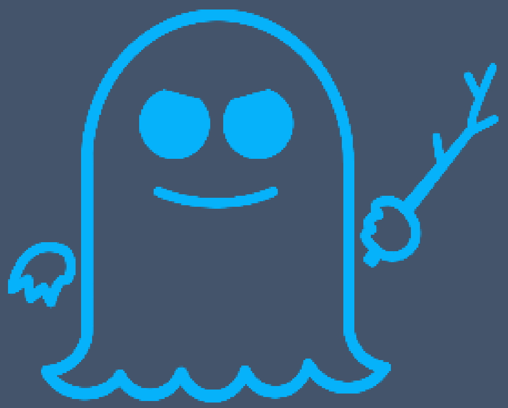
Total

1156 binaries

- Dedicated secure scrubbing functions (e.g. `memset_s`) are secure
- Disabling DSE sometimes works but is *not always sufficient*
- Volatile function pointers can introduce additional register spilling that might break secret-erasure with gcc -O2 and gcc -O3

Extension: Spectre

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)
```

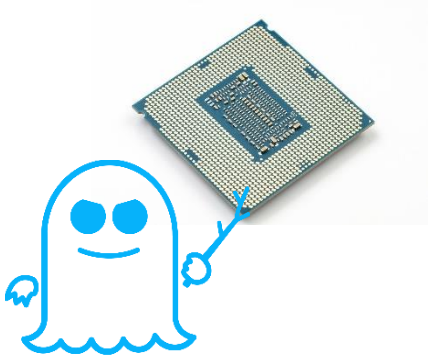
leak(init_mem[a])

- 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

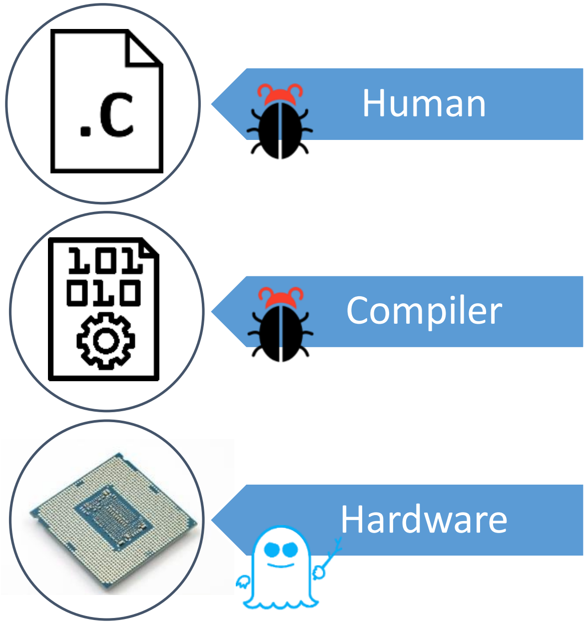
Not easy to write constant-time programs

- 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 ☹

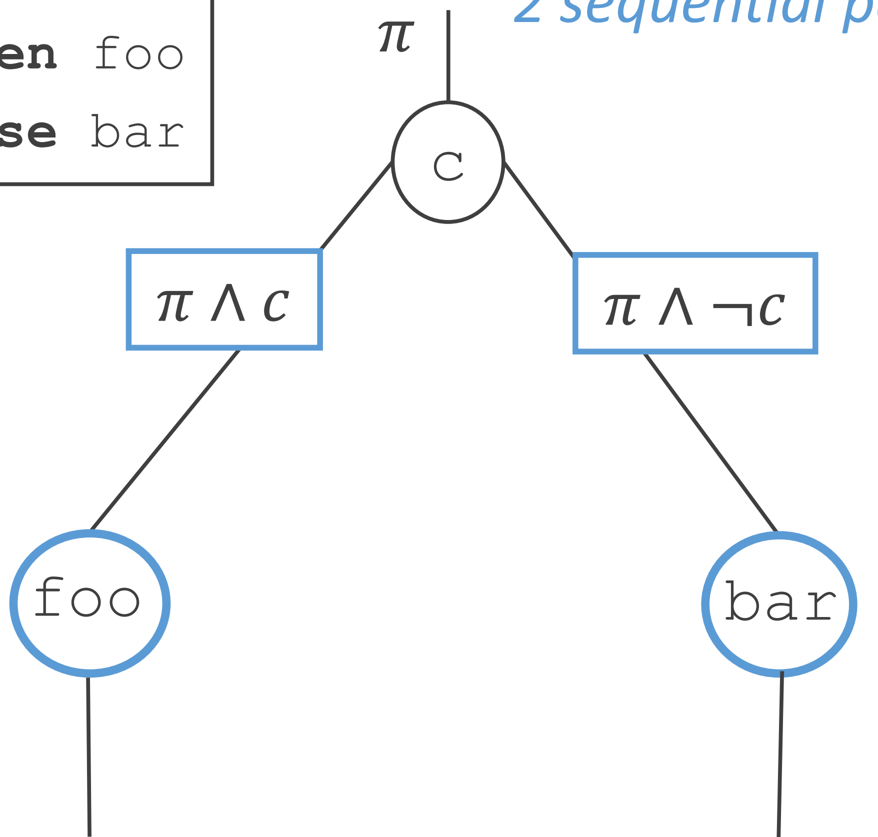
[1] G. Wang, et al “KLEESpectre: Detecting Information Leakage through Speculative Cache Atttacks via Symbolic Execution”, ACM Trans. Softw. Eng. Methodol., vol. 29, no. 3, 2020.
[2] S. Guo, Y. Chen, P. Li, Y. Cheng, H. Wang, M. Wu, and Z. Zuo, “SpecuSym: Speculative Symbolic Execution for Cache Timing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

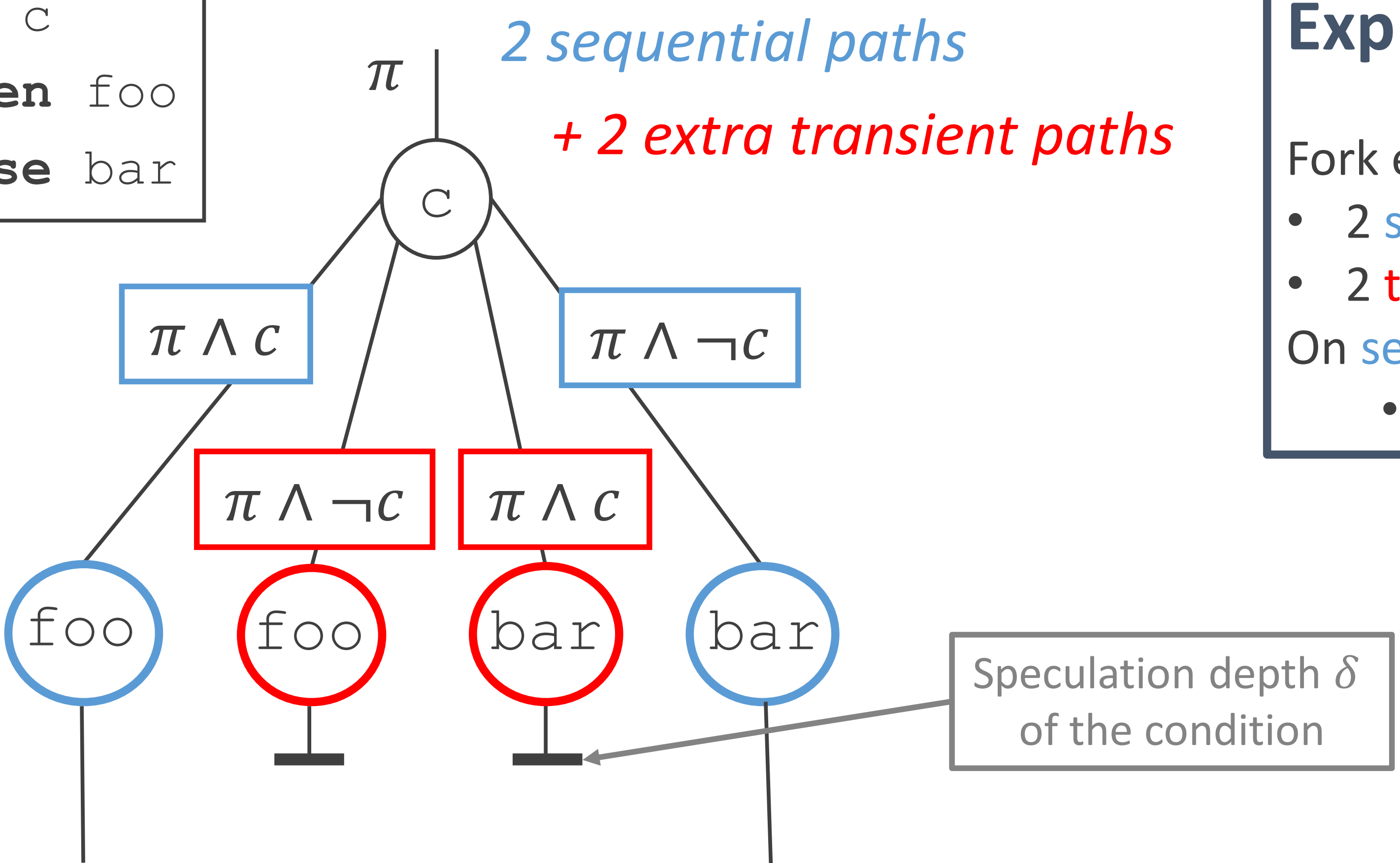


2 sequential paths

Explicit RelSE for Spectre PHT

Spectre-PHT. Conditional branches can be executed speculatively

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



Explicit RelSE.

Fork execution into 4 at conditionals:

- 2 sequential branches
- 2 transient branches

On sequential and transient branches:

- Verify no secret can leak.

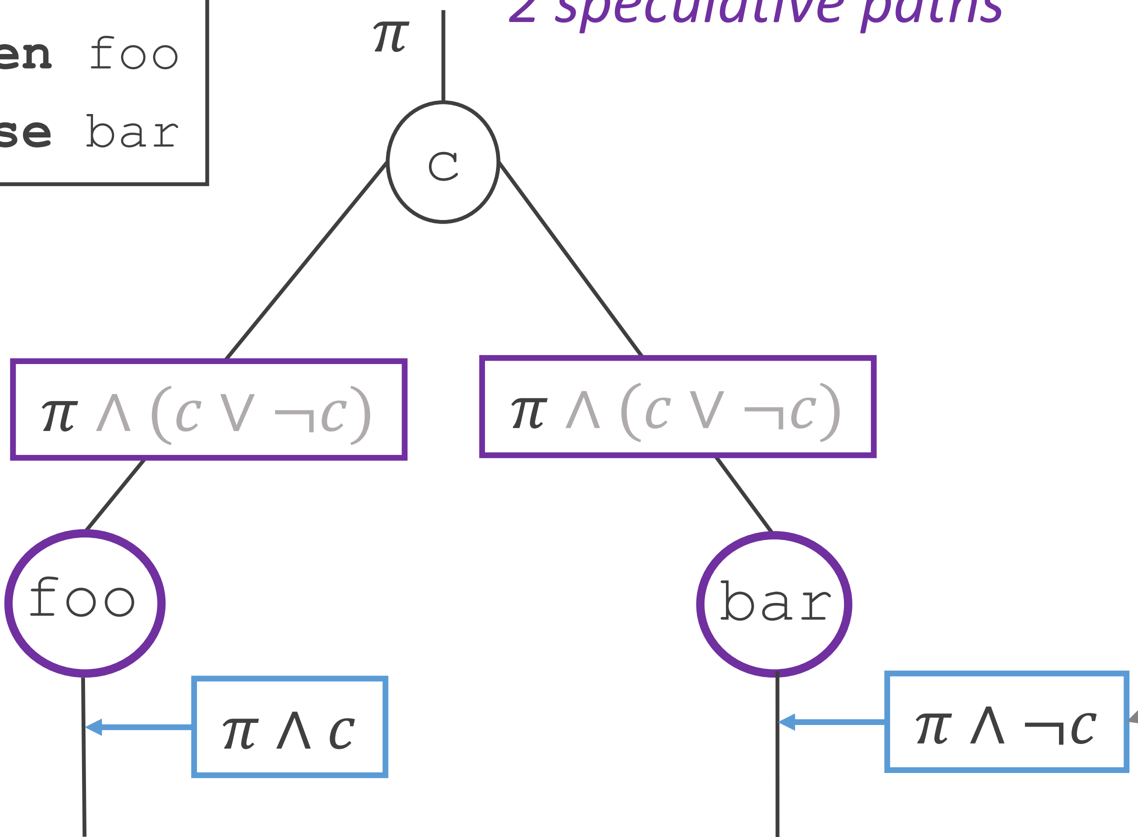
(e.g. KLEESpectre)

Haunted RelSE for Spectre PHT

Spectre-PHT. Conditional branches can be executed speculatively

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

2 speculative paths



Speculation depth δ of the condition

Haunted RelSE.

Fork execution into 2 speculative paths:

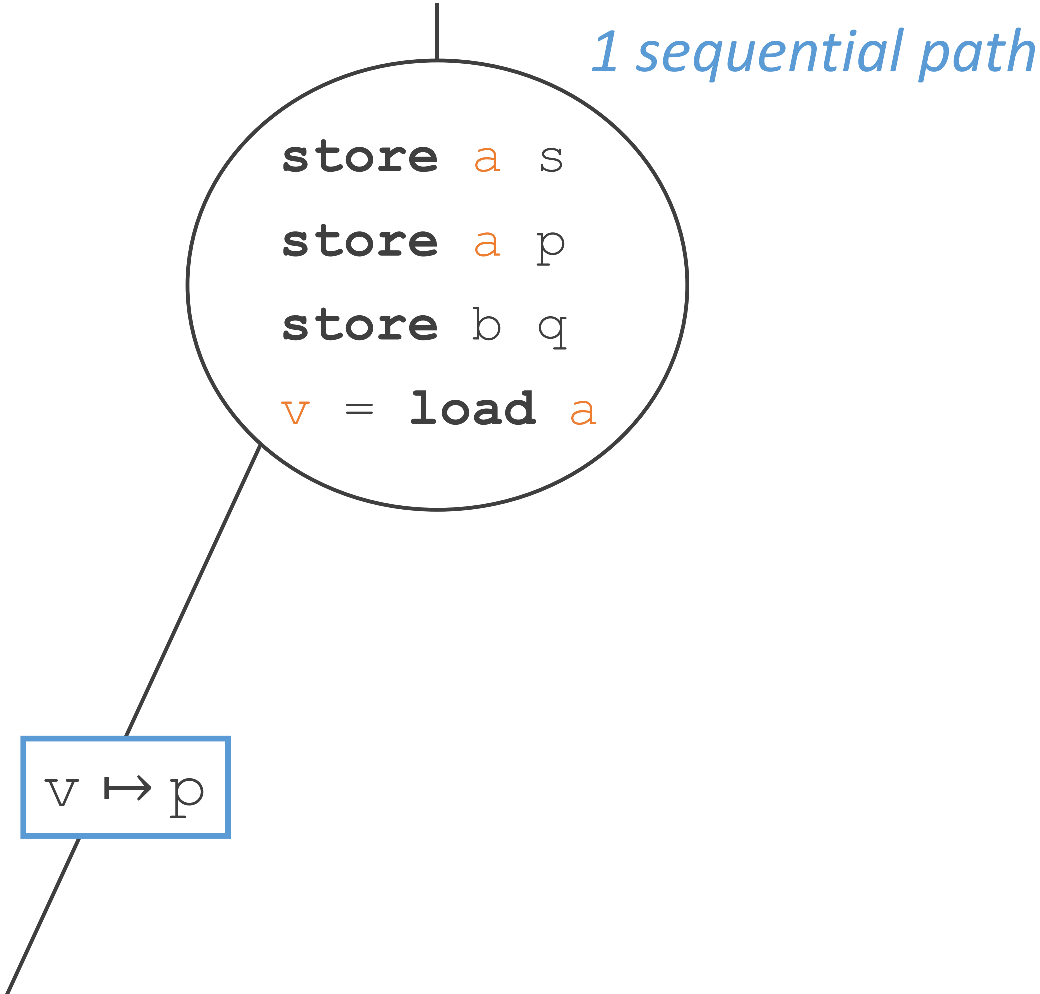
- speculative = sequential ∨ transient
- Add constraint to invalidate transient path

→ *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
```

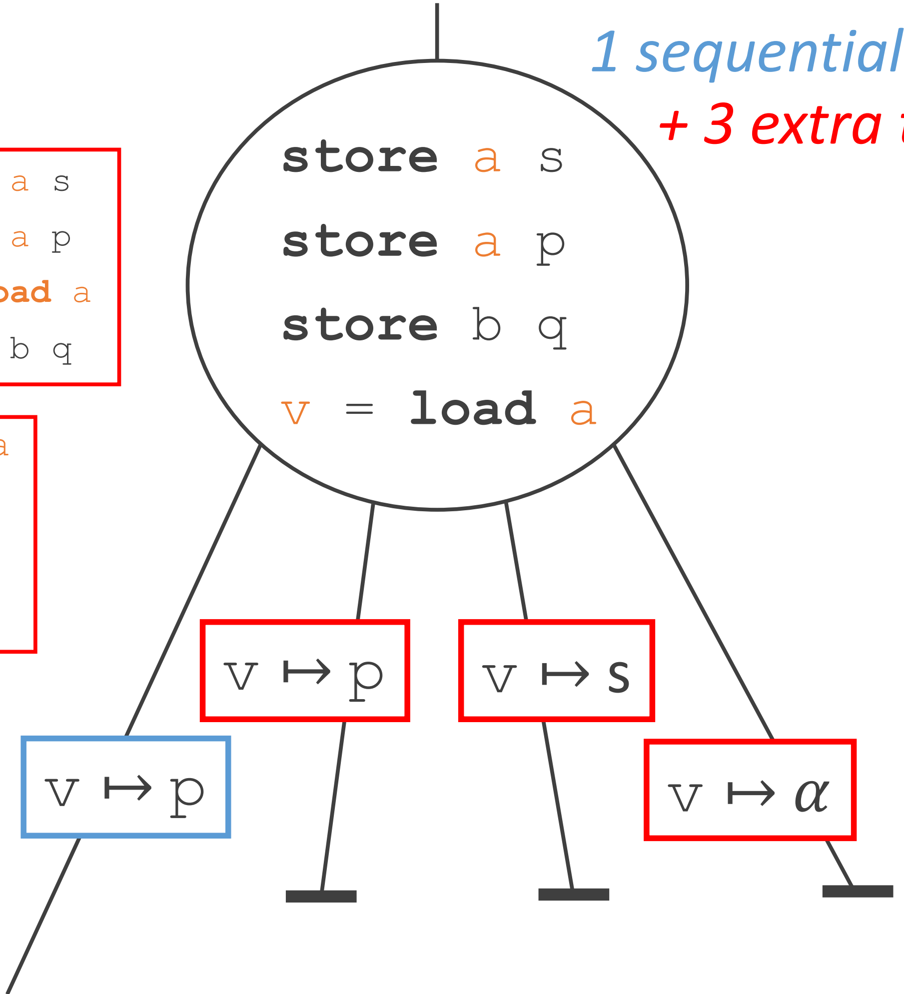
```
store a s
v = load a
store a p
store b q
```

```
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

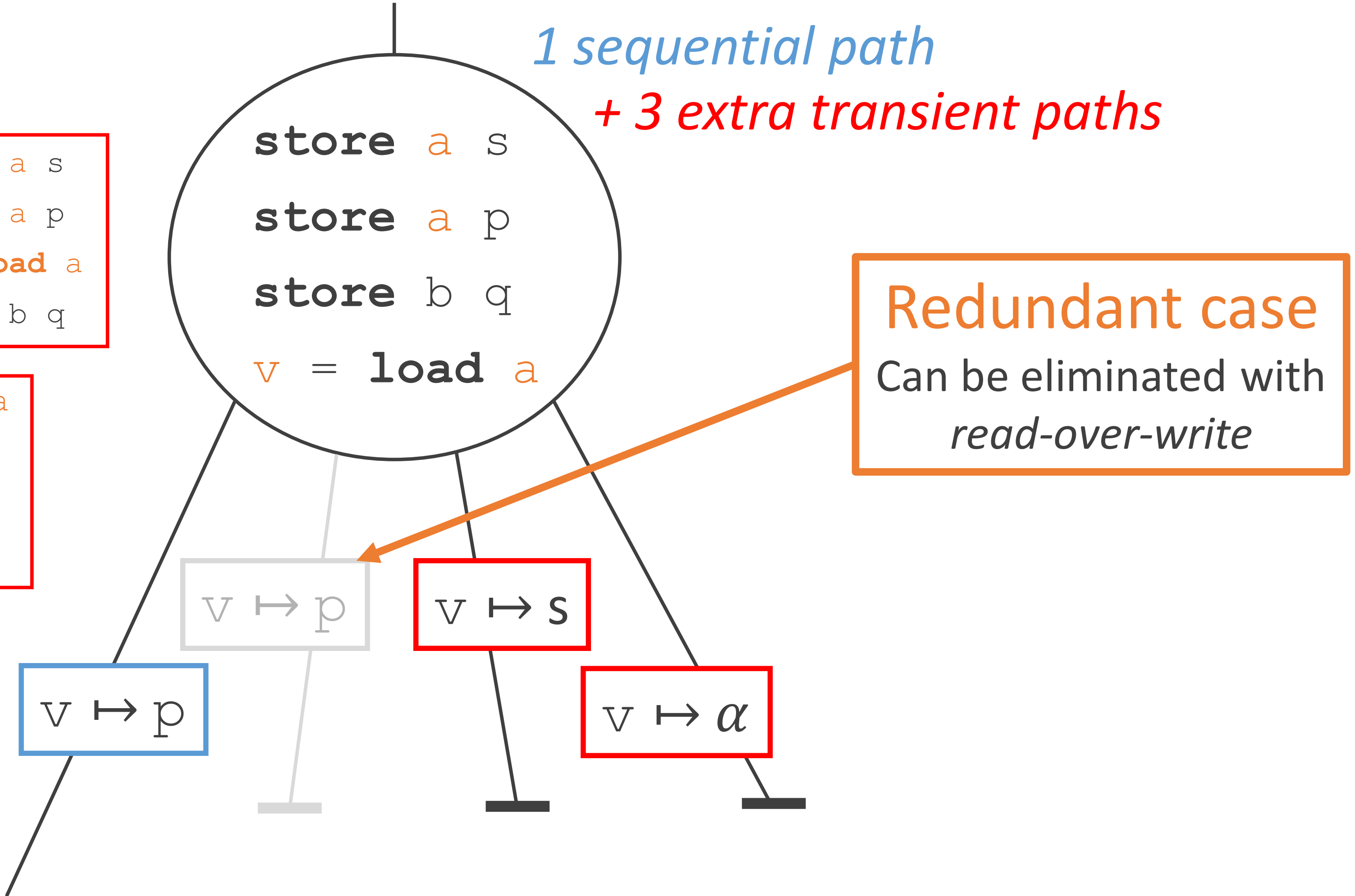
```
store a s
store a p
store b q
v = load a
```

```
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



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

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");`