Symbolic Binary-Level Code Analysis for Security

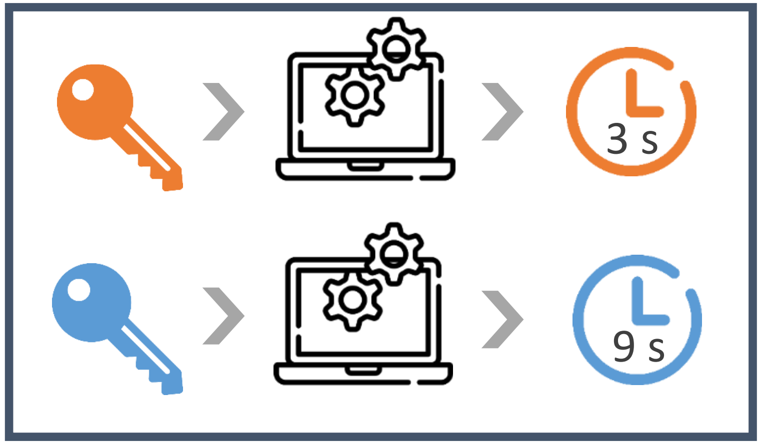
Application to the Detection of Microarchitectural Attacks
in Cryptographic Code

TEE talk October, 25th 2021
Lesly-Ann Daniel
CEA List and Université Côte d'Azur

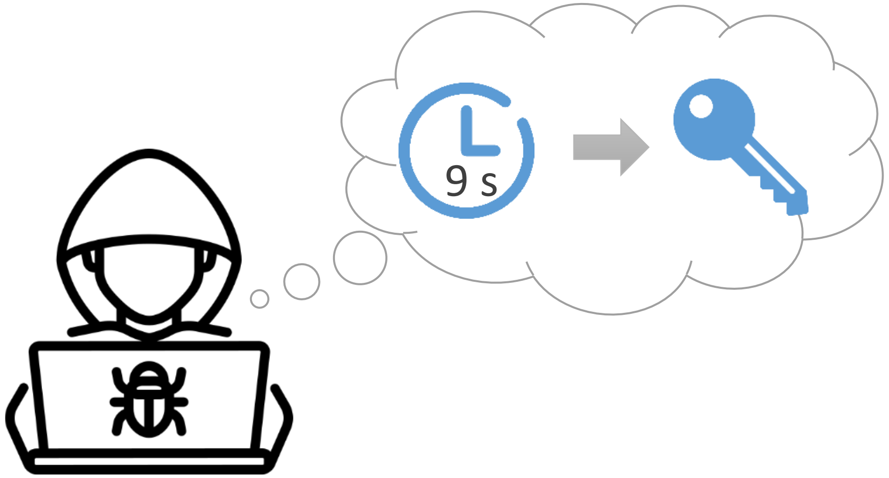
Supervised by:
- Sébastien Bardin, CEA List
- Tamara Rezk, INRIA

Timing and Microarchitectural Attacks

Timing and microarchitectural attacks:
Execution time / microarchitectural state can leak secret information manipulated by programs

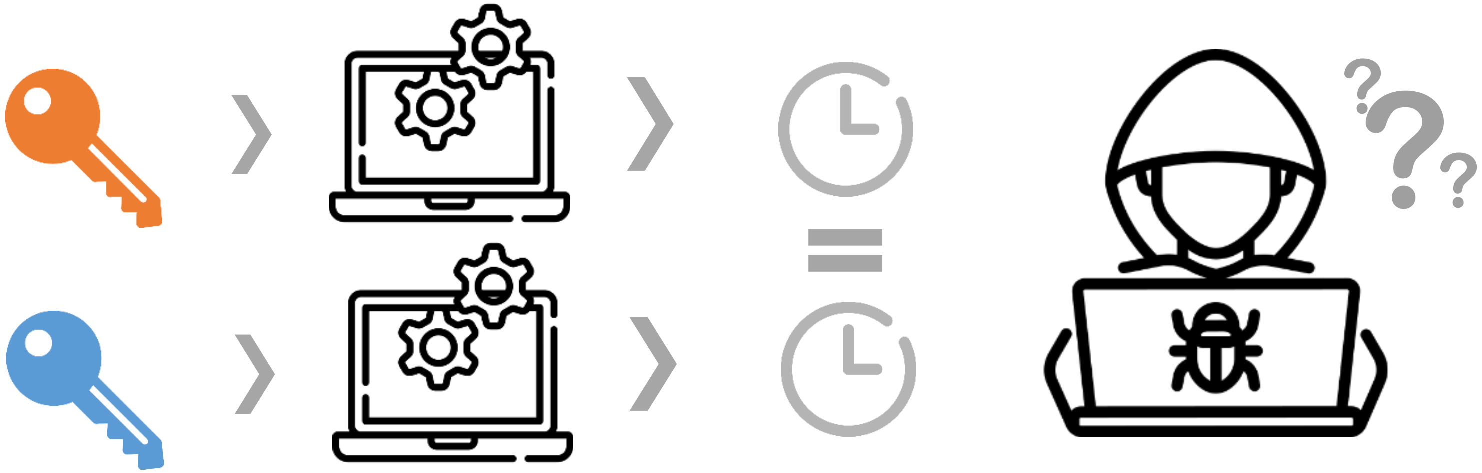


First timing attack in **1996** by Paul Kocher: full recovery of **RSA encryption key**



Protect software with Constant-Time programming

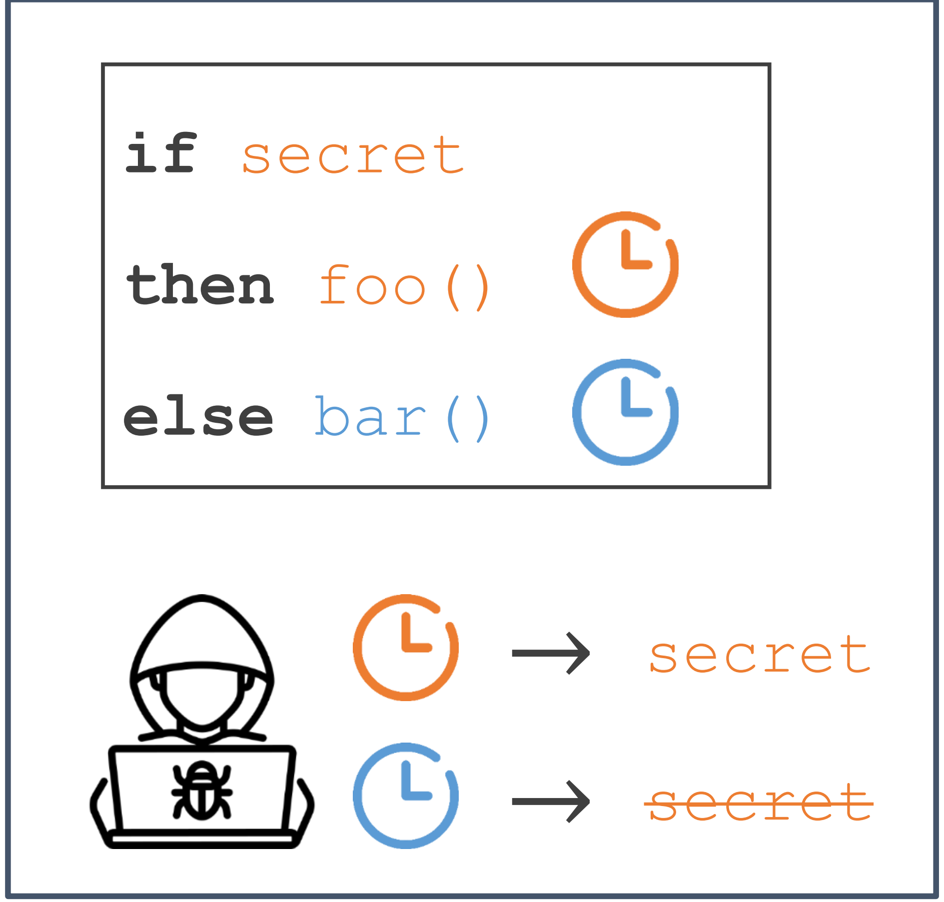
Constant-Time. Execution time / changes to microarchitectural state are independent from secret input



Already used in many cryptographic implementations

What can influence exec. time/microarchitecture?

Control Flow



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

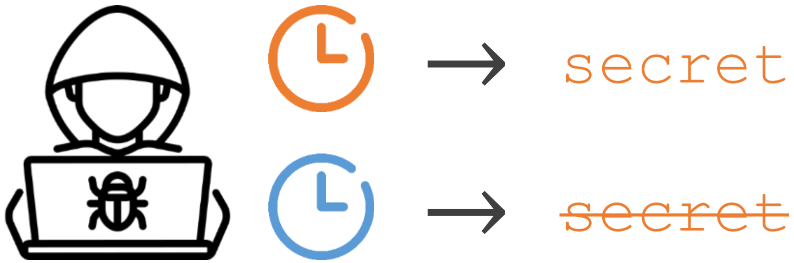
🕒 → secret

🕒 → ~~secret~~

What can influence exec. time/microarchitecture?

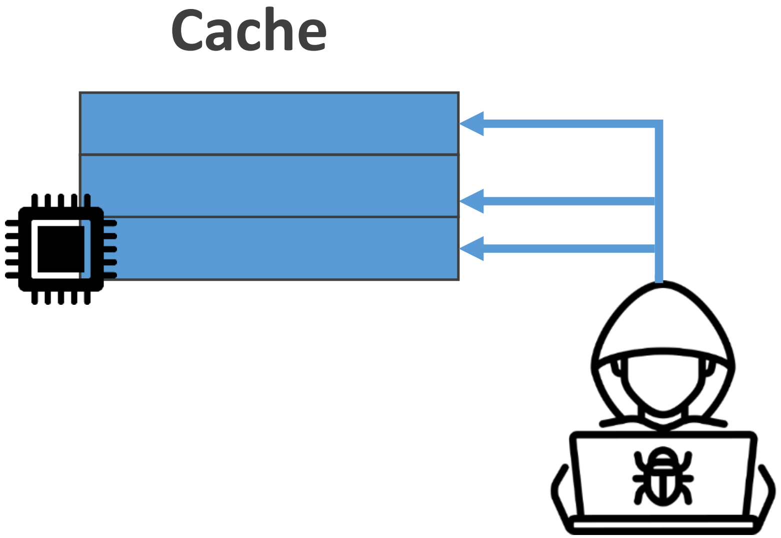
Control Flow

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



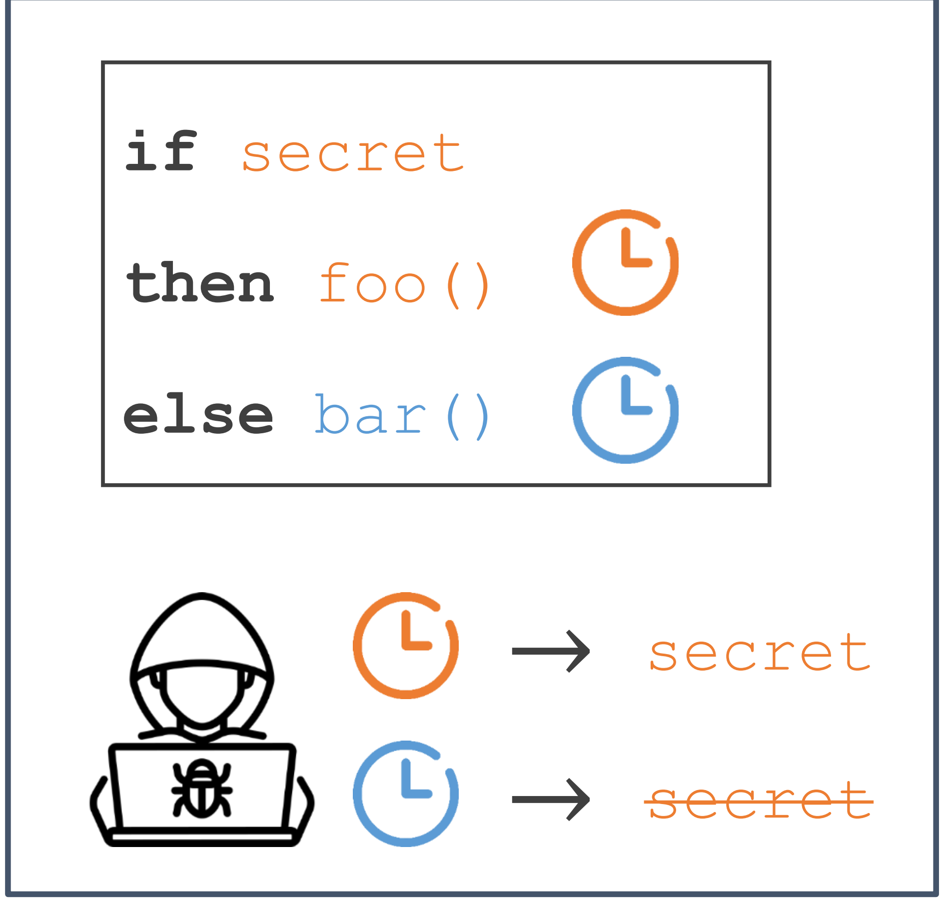
Memory Accesses

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



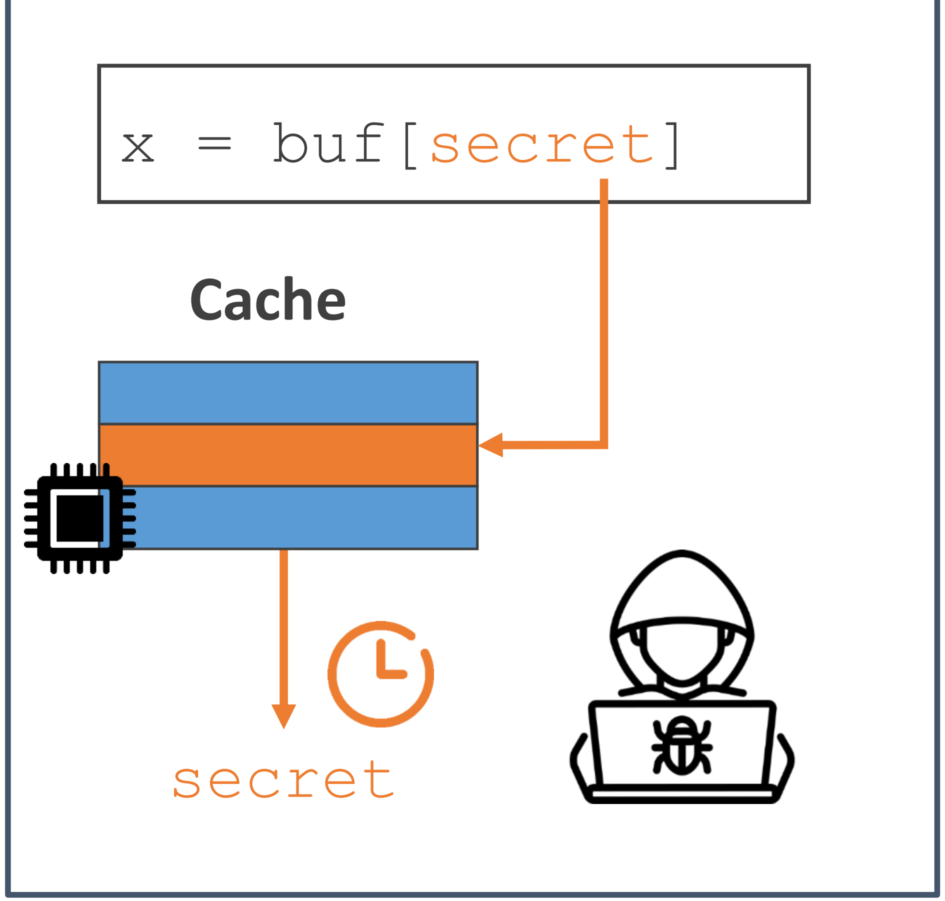
What can influence exec. time/microarchitecture?

Control Flow



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

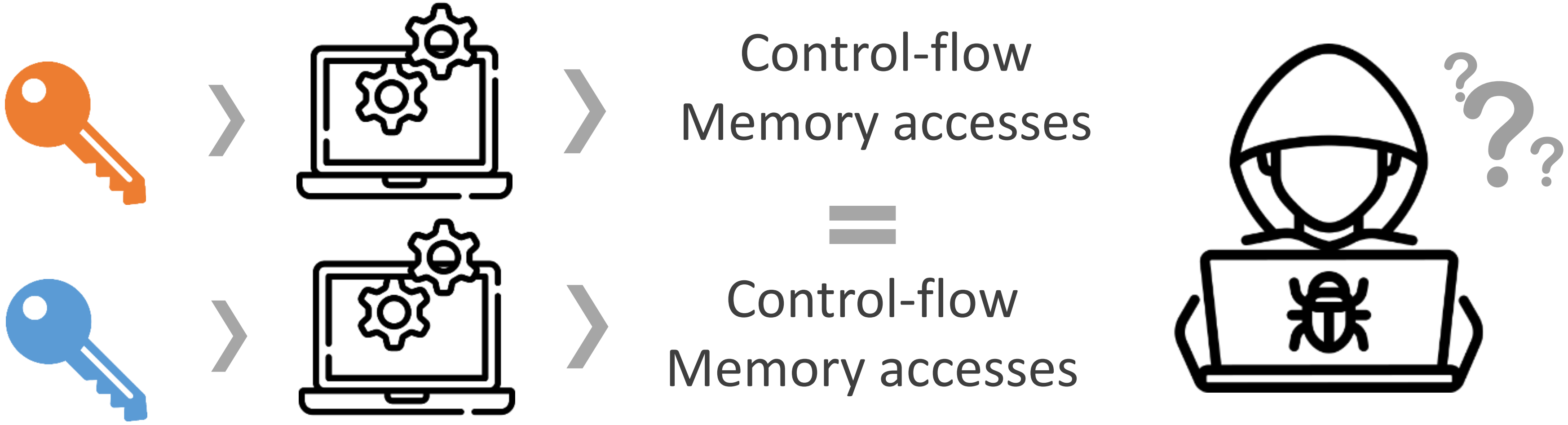
Memory Accesses



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

Protect software with Constant-Time programming

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



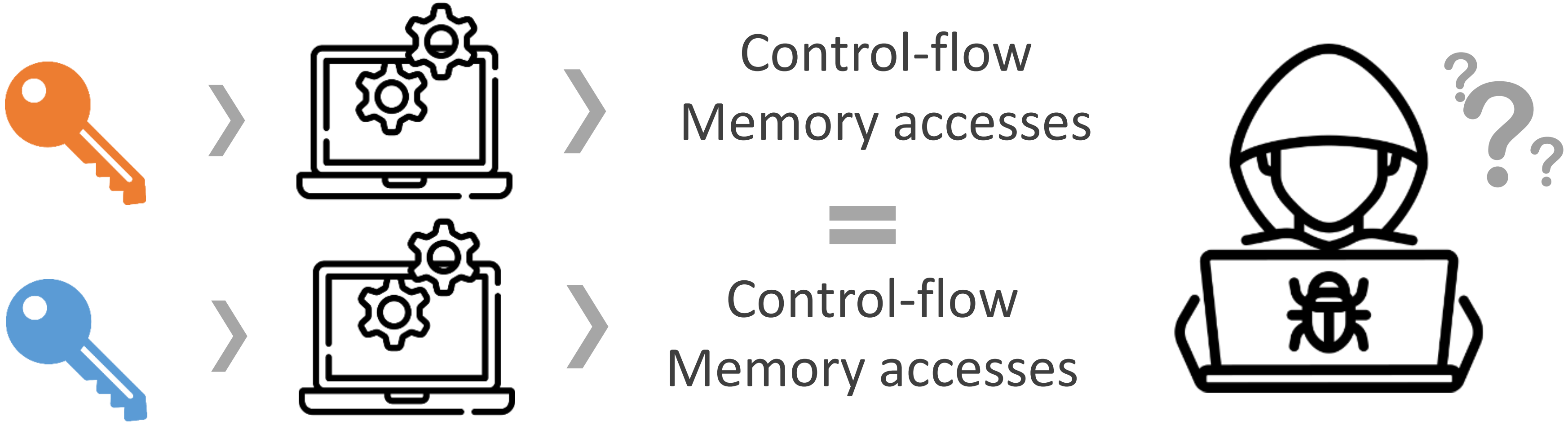
Control-flow
Memory accesses

=

Control-flow
Memory accesses

Protect software with Constant-Time programming

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



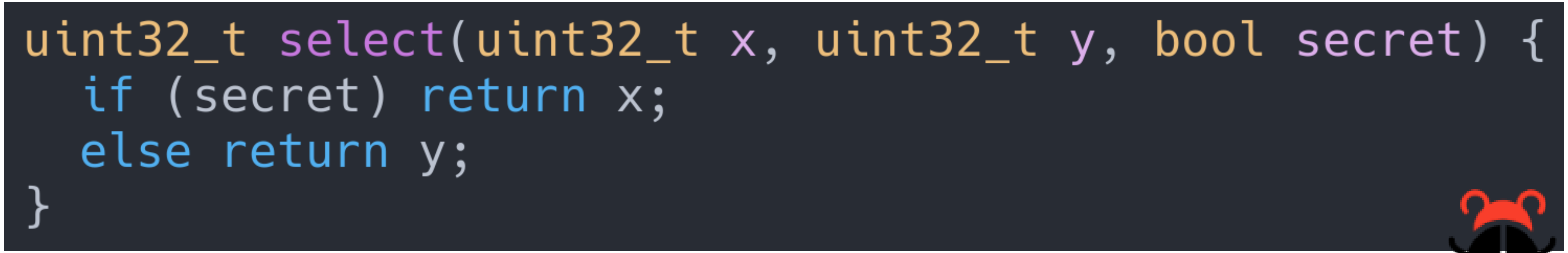
Control-flow
Memory accesses

=

Control-flow
Memory accesses

Property relating 2 execution traces (2-hypersafety)

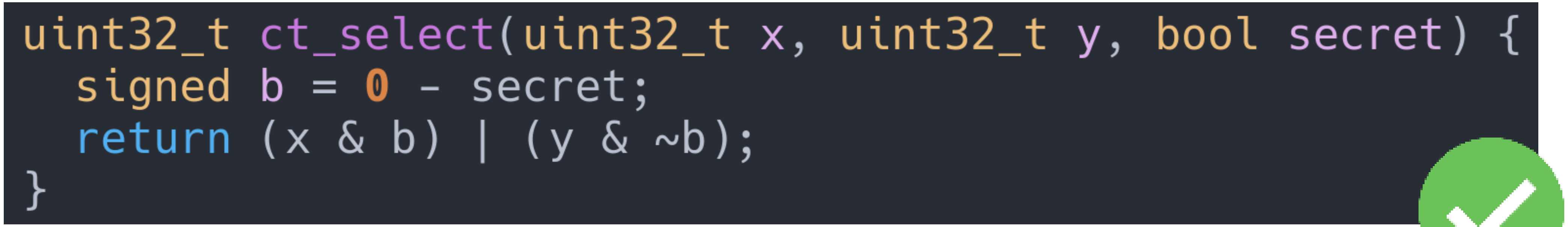
CT code is not easy to implement



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

🐞

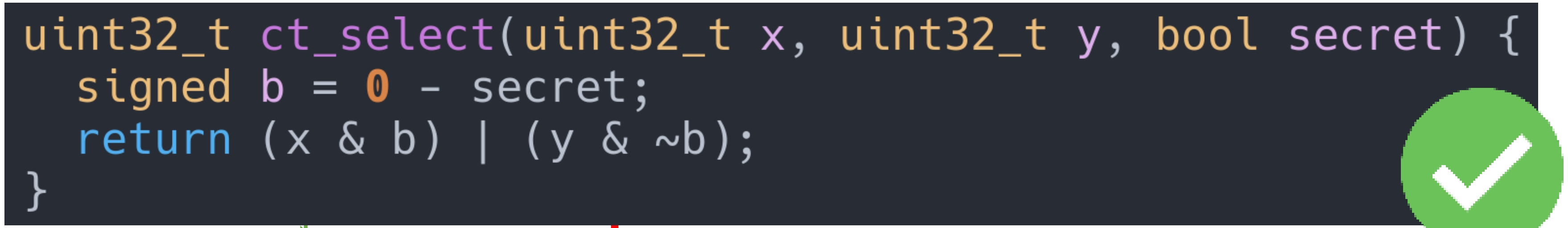
↓



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

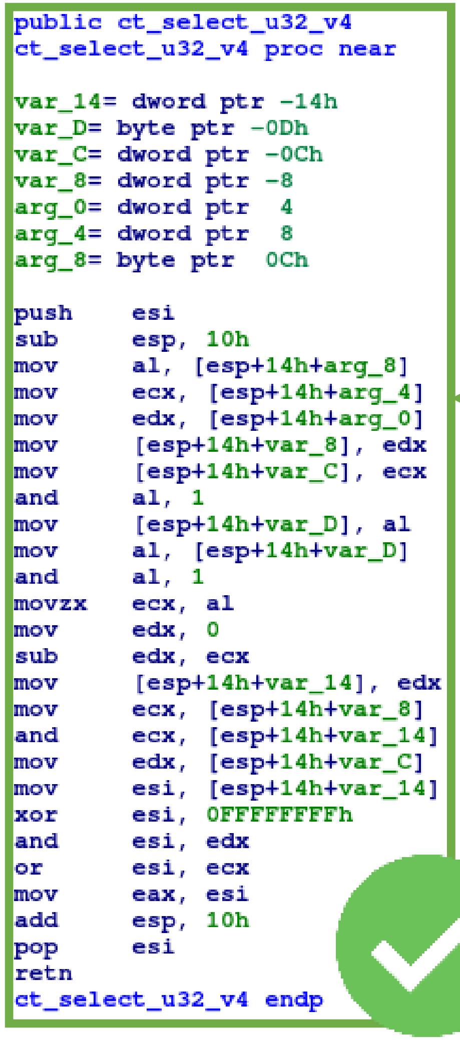
✓

Compilers can break CT!



```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

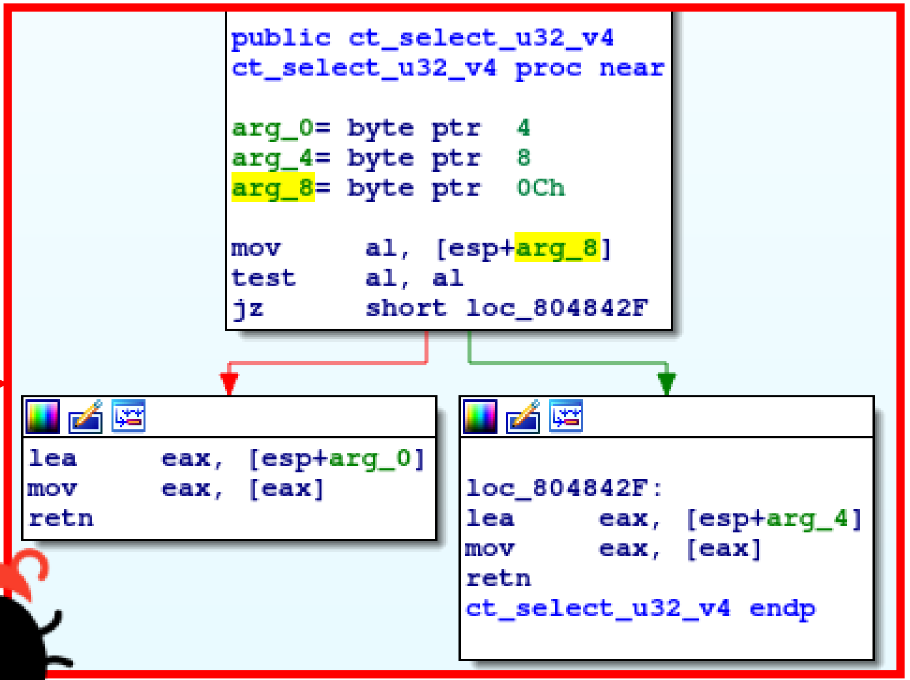


```
public ct_select_u32_v4
ct_select_u32_v4 proc near

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
```

Goal

Automated verification tools for constant-time (and more) at binary-level

Automated program verification

Verification tool

Bug-Finding

Verification

Ideally we would like our verification tool to:

- Reject all insecure programs
- Accept all 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

Ideally we would like our verification tool to:

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

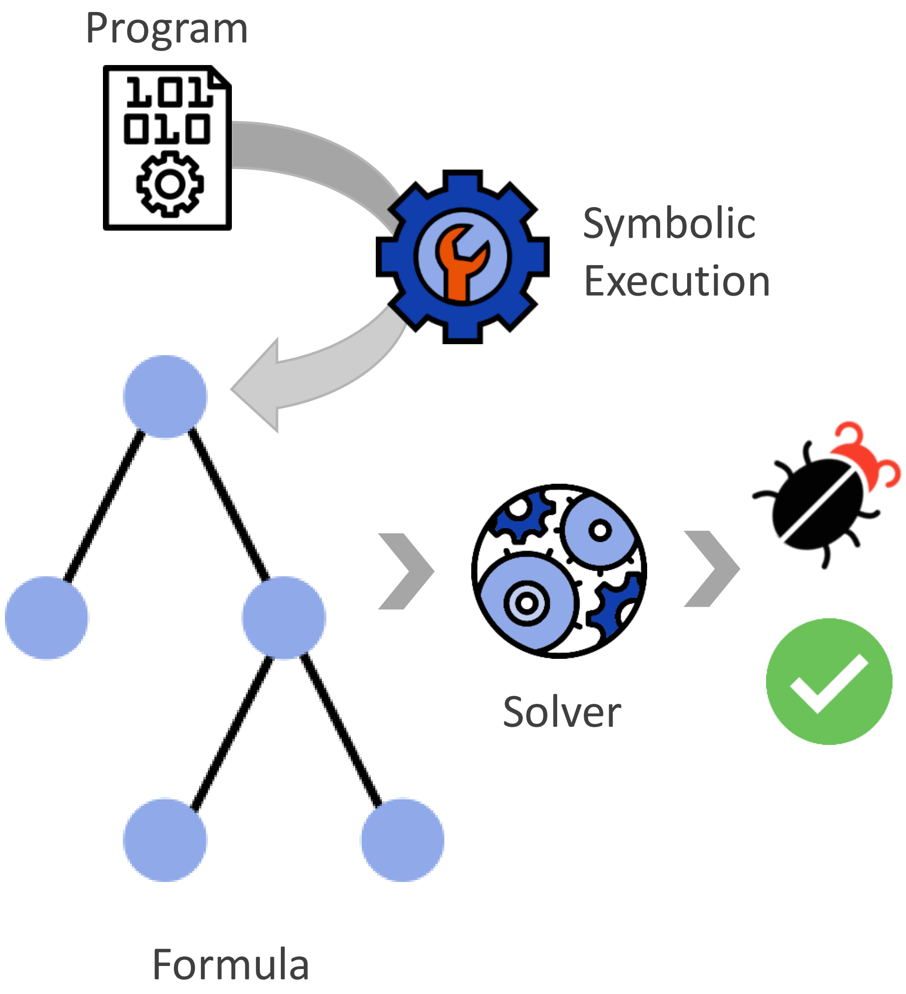
Convenient to have both because binary-level tools are difficult to use!

Bounded Verification & Bug-Finding?
Try Symbolic Execution

- Leading formal method for bug-finding
- Scales well on binary code
- Finds real bugs + reports counterexamples
- Can also do bounded-verification

The KeY Project

BINSEC



PART 1

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

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

+ Beyond constant-time (overview)

PART 2

NDSS
SYMPOSIUM/2021

Haunted RelSE: detect Spectre vulnerabilities

PART 1

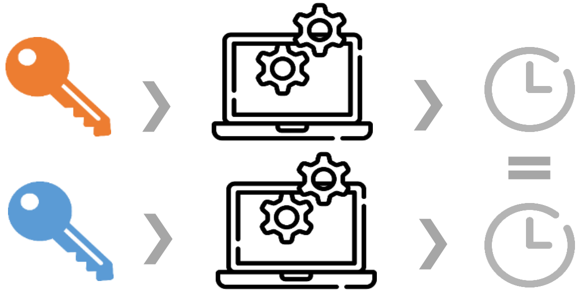
Binsec/Rel:
Efficient constant-time verification at binary-level

MAY 18-20, 2020

41st IEEE Symposium on Security and Privacy

Challenges of CT analysis

Property of 2 executions



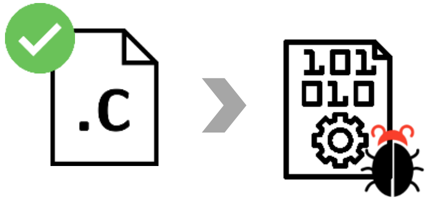
→ Efficiently model pairs of executions

Standard SE do not apply

RelSE

SE for pairs of traces with sharing

Not necessarily preserved by compilers



Compilation

→ Binary-analysis

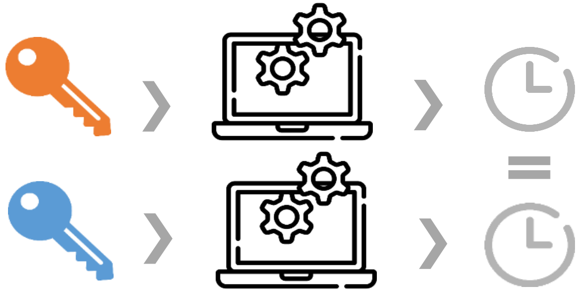
Reason explicitly about memory

Binary-level SE

BINSEC

Challenges of CT analysis

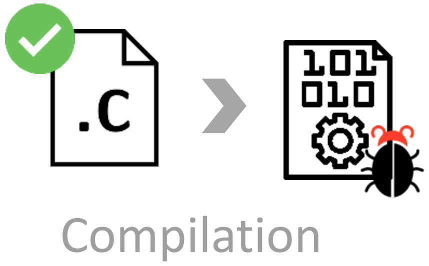
Property of 2 executions



→ Efficiently model pairs of executions

Standard SE do not apply

Not necessarily preserved by compilers



→ Binary-analysis

Reason explicitly about memory

RelSE
SE for pairs of traces with sharing

\+

Binary-level SE
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

Dedicated optimizations for RelSE at binary-level: maximize sharing in memory (x700 speedup)

New Tool

BINSEC/REL
First efficient tool for BV&BF of CT at *binary-level*
+ formal proofs

Application: crypto verif.

From OpenSSL, BearSSL, libsodium
296 verified binaries
3 new bugs introduced by compilers from verified source
Out of reach of LLVM verification tools

RQ3: Preservation of CT 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

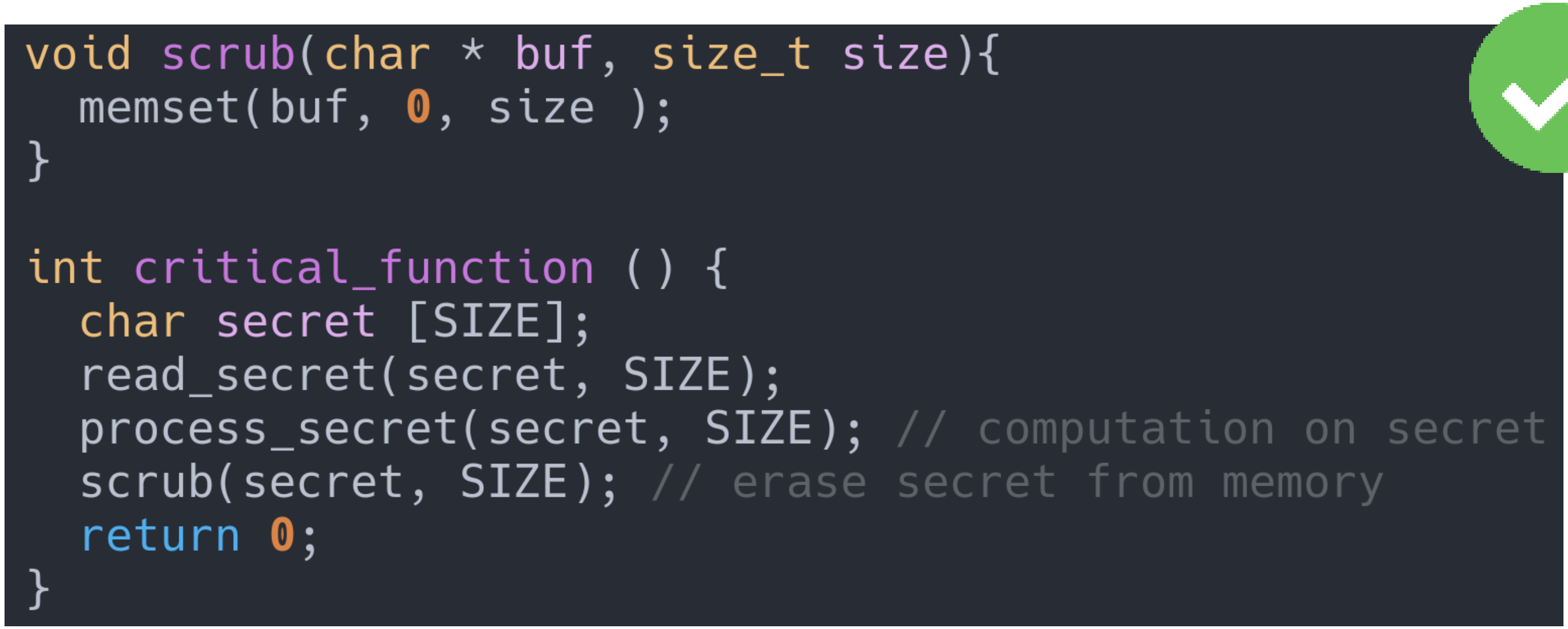
Total
408 binaries

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

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

Beyond Constant-Time

Secret-erasure

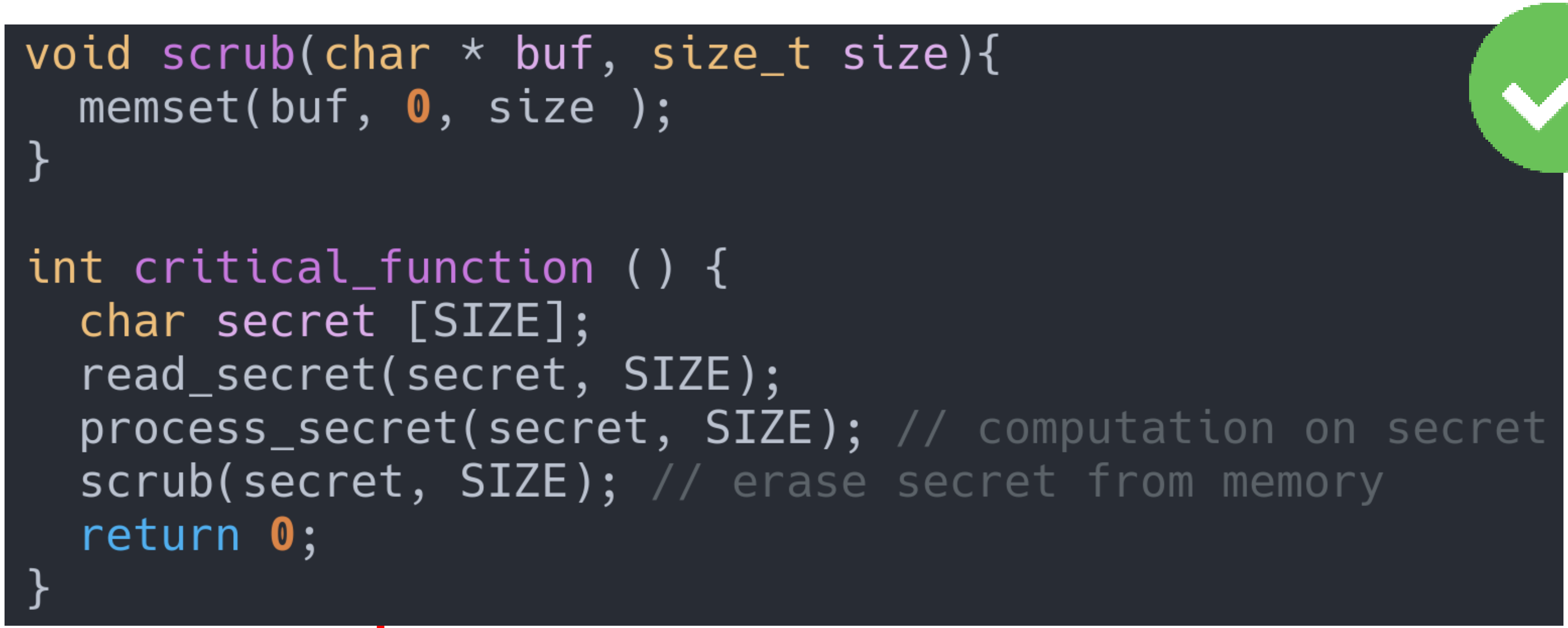


```
void scrub(char * buf, size_t size){
  memset(buf, 0, size );
}

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;
}
```

✓

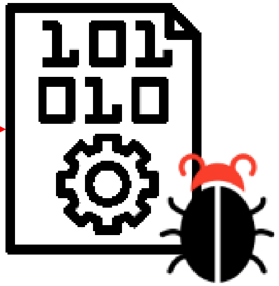
Secret-erasure



```
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

Conclusion

Conclusion

Binsec/Rel

https://github.com/binsec/rel

- Dedicated optimizations for RelSE at binary-level

  → *Sharing for scaling*

- Binsec/Rel, binary-level tool for analyzing constant-time & secret-erasure

  → *For bug-finding & bounded-verif*

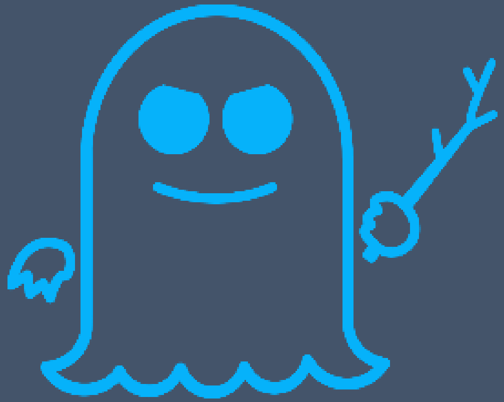
- Verification of crypto primitives at binary-level

  → *new bugs introduced by compilers out-of reach of LLVM verification*

PART 2

NDSS
SYMPOSIUM/2021

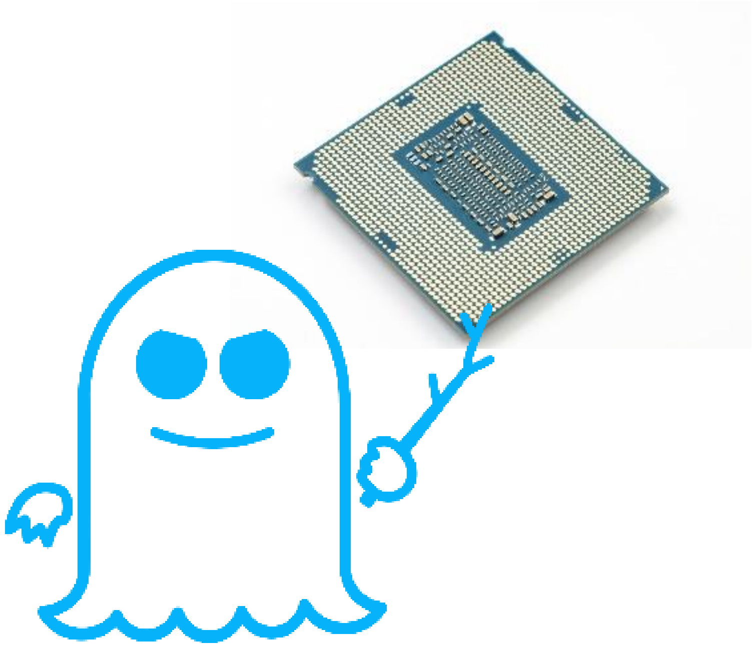
Haunted RelSE: detect Spectre vulnerabilities



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

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` $\neq$ `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)
```

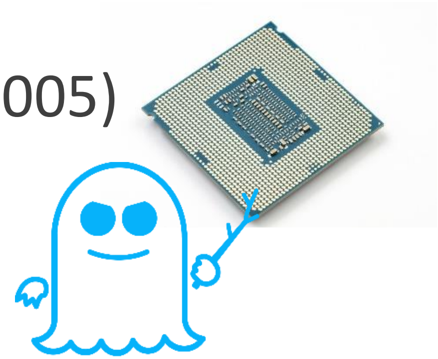
leak(init_mem[a])

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

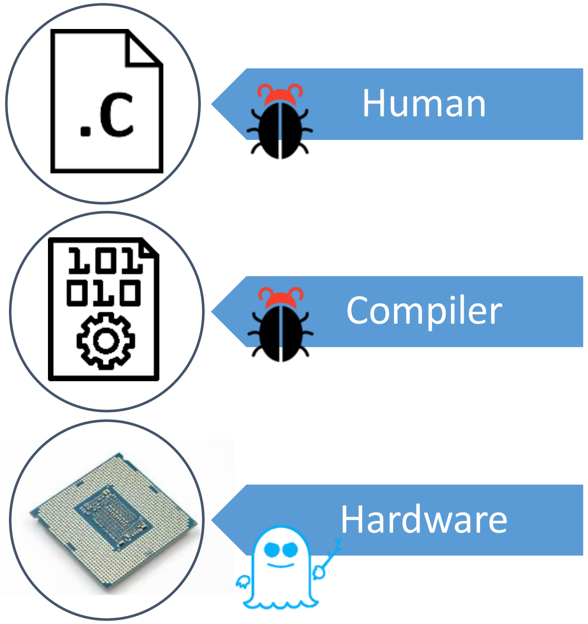
Constant-time verification & Spectre attacks

Execution time is not easy to determine

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



Multiple failure points



Human

Compiler

Hardware

Not easy to write constant-time programs

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

Detect Spectre attacks ?

Challenging !

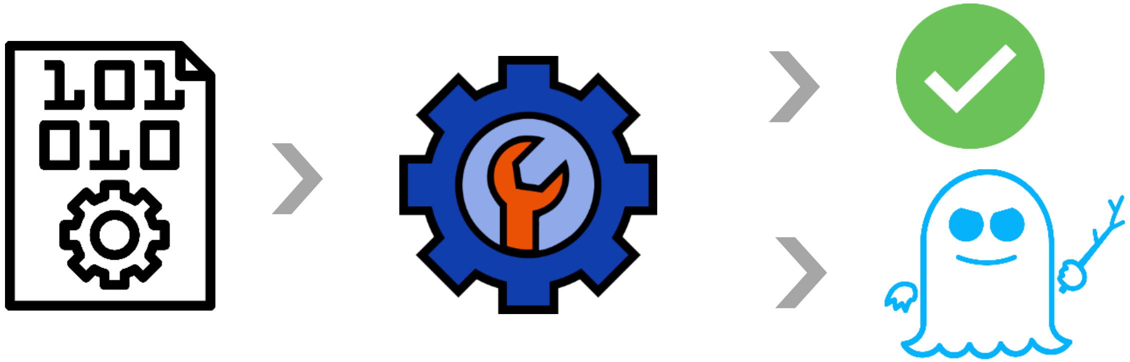
- Counter-intuitive semantics
- Path explosion:
  - **Spectre-STL**: all possible load/store interleavings !
- Needs to hold at binary-level

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

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

Goal: New verification tools for Spectre

Goal. We need new verification tools to detect Spectre vulnerabilities !



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 sequential behaviors at the same time
- 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

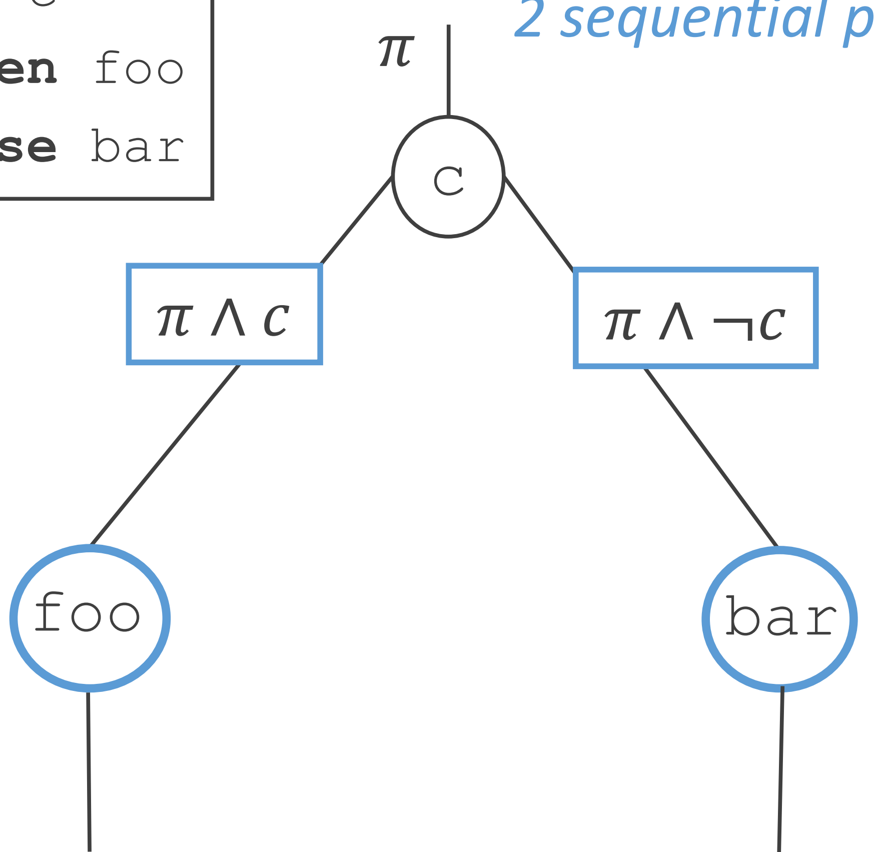
- Index-masking (countermeasure against Spectre-PHT) + proven mitigations
- Code introduced for Position-Independent-Code [in the paper]

Haunted RelSE for Spectre-PHT

Background: Symbolic Execution

Symbolic execution. An illustration.

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

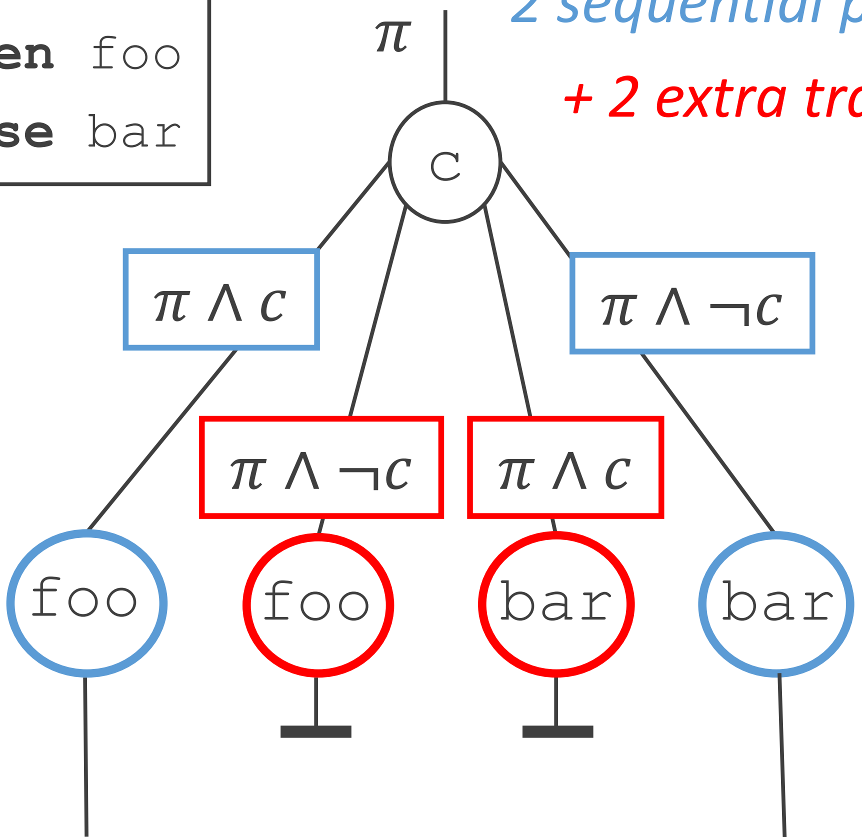


2 sequential paths

Explicit RelSE for Spectre PHT

Spectre-PHT. Conditional branches can be executed speculatively

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



2 sequential paths

+ 2 extra transient paths

Explicit RelSE.

Fork execution into 4 at conditionals:

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

On sequential and transient branches:

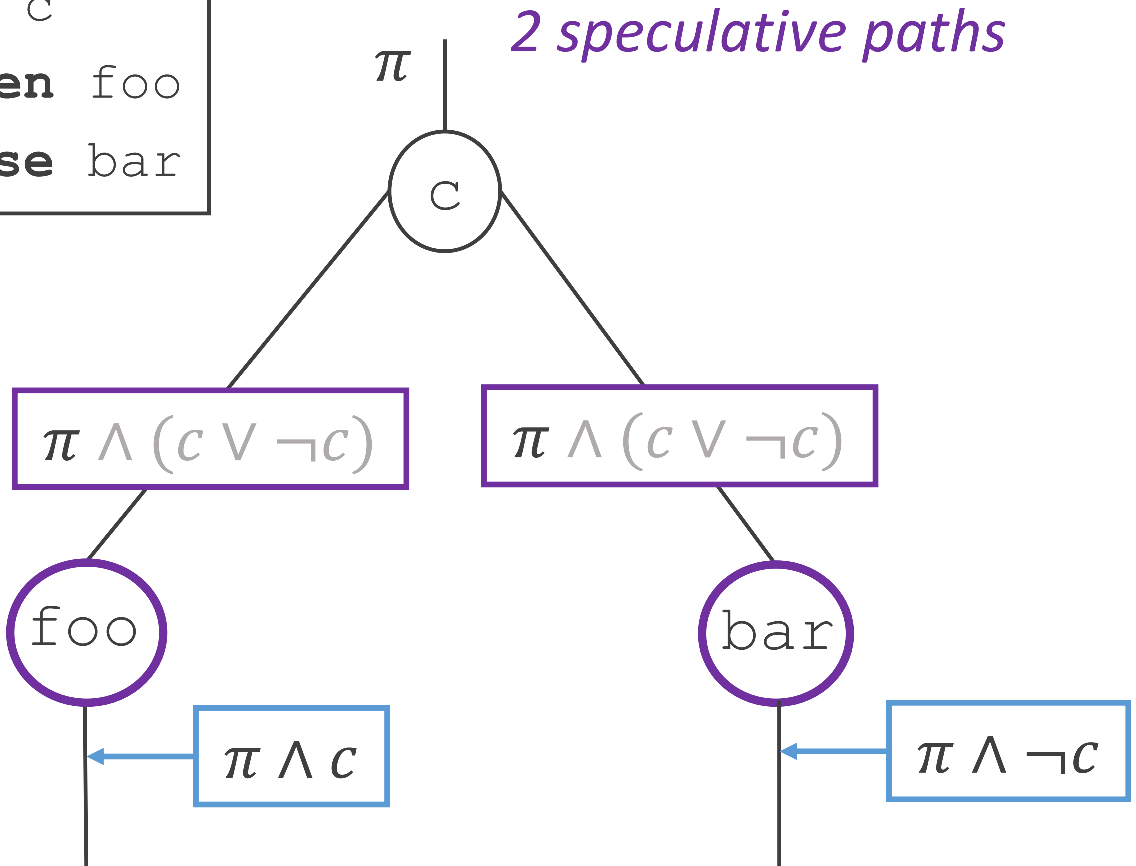
  - Verify no secret can leak.

(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

Haunted RelSE.

Fork execution into 2 speculative paths:

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

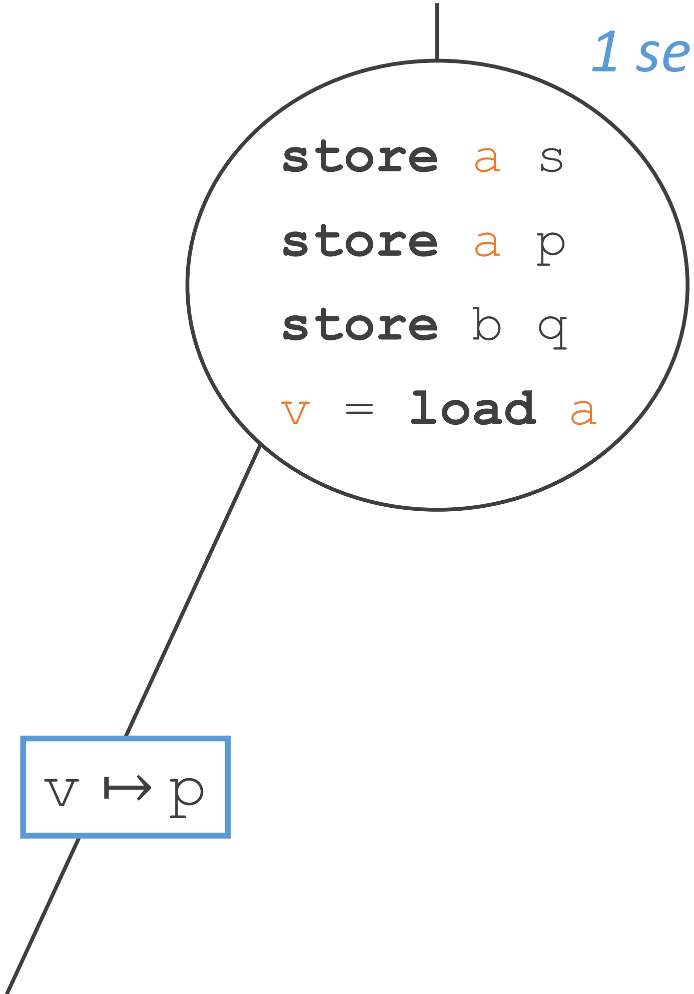
→ *can spare two paths at conditionals*

Haunted RelSE for Spectre-STL

Explicit RelSE for Spectre-STL

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

where $a \neq b$



1 sequential path

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

$v \mapsto p$

Explicit RelSE for Spectre-STL

Spectre-STL. Loads can speculatively bypass prior stores

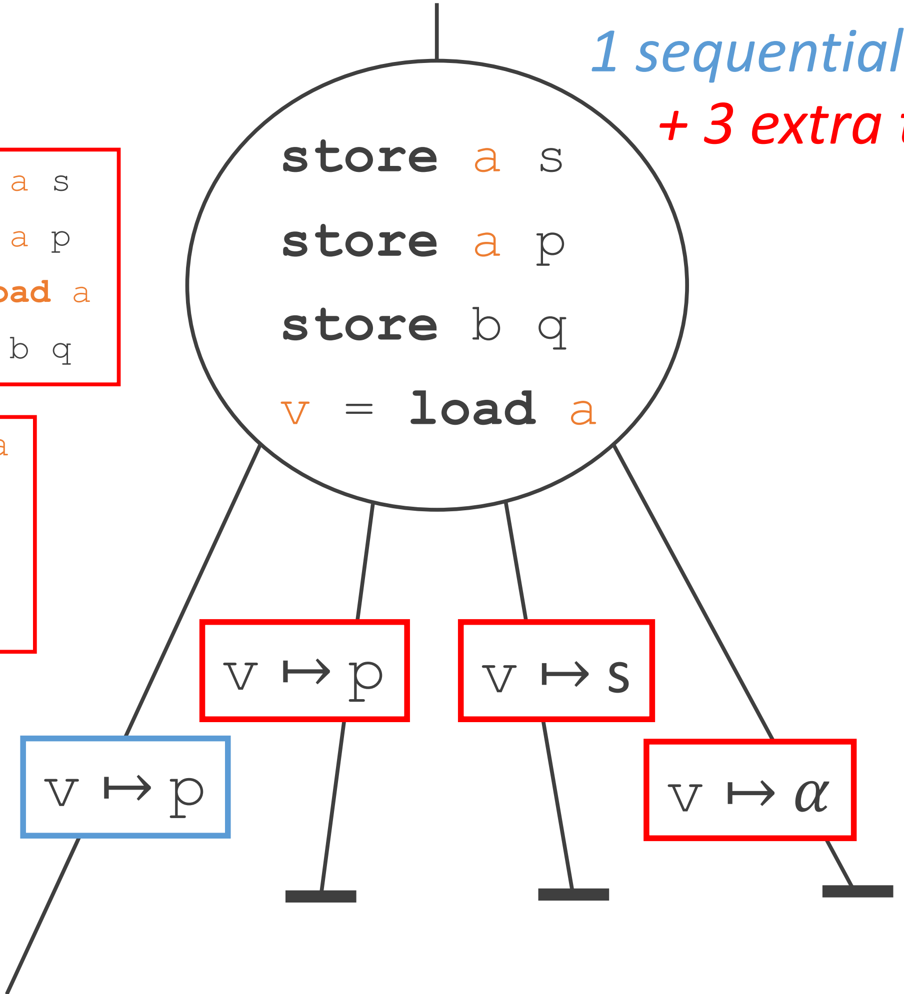
```
store a s
store a p
store b q
v = load a
```

```
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)

Explicit RelSE for Spectre-STL

Spectre-STL. Loads can speculatively bypass prior stores

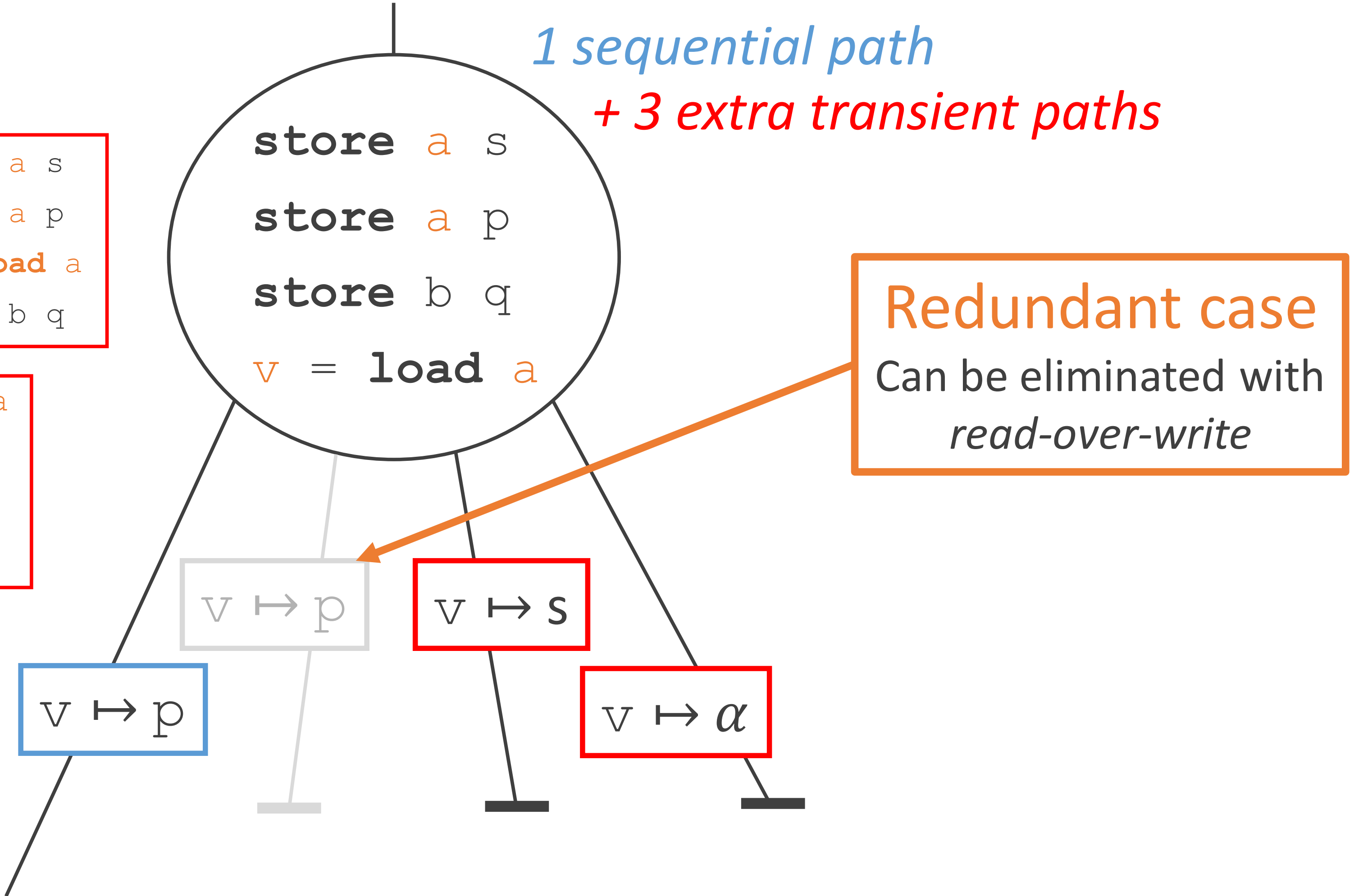
```
store a s
store a p
store b q
v = load a
```

```
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

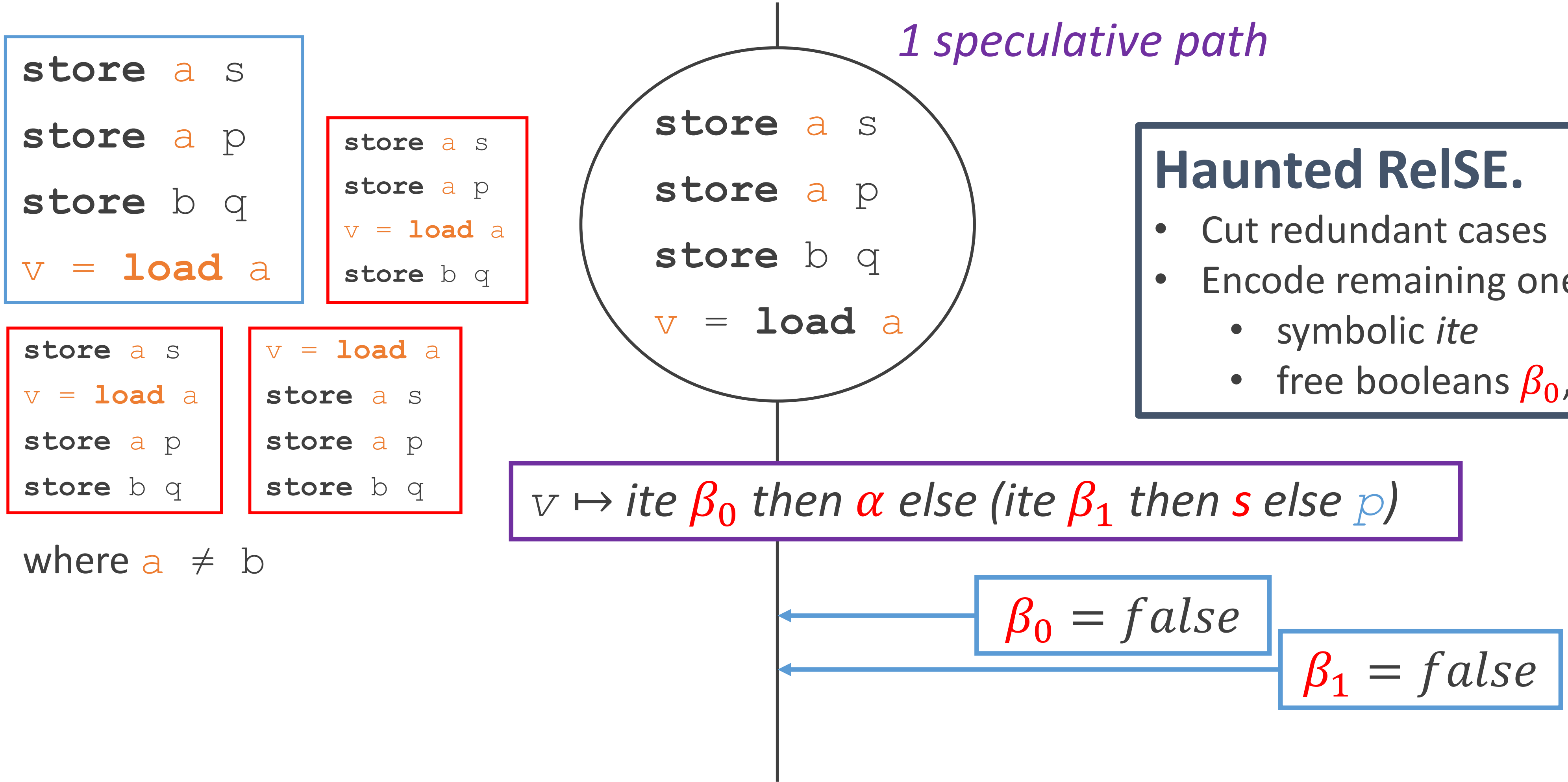
```
store a s
store a p
store b q
v = load a
```

Redundant case

Can be eliminated with *read-over-write*

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
```

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

$\beta_0 = false$

$\beta_1 = false$

Haunted RelSE.

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

Experimental evaluation

Binsec/Haunted.
Implementation of Haunted RelSE

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 ↗*

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 sequential 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

- 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

Binsec/Haunted

https://github.com/binsec/haunted

- 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

Follow-up ?

Extend framework to check property preservation by compilers

- Analysis of other countermeasures (lfence, speculative load hardening)
- Spectre RSB/BTB + analysis of countermeasures

Exploitability

- Less conservative SCT definition: `load ebp-4` cannot bypass `store ebp-4`
- Cache model

Backup

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 = current location

eax = global pointer

edx = global variable

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

current location pushed on stack at call

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