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

20èmes journées Approches Formelles dans l'Assistance au Développement de Logiciels

AFADL, 17 Juin 2021

Published at IEEE Symposium on Security and Privacy 2020

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

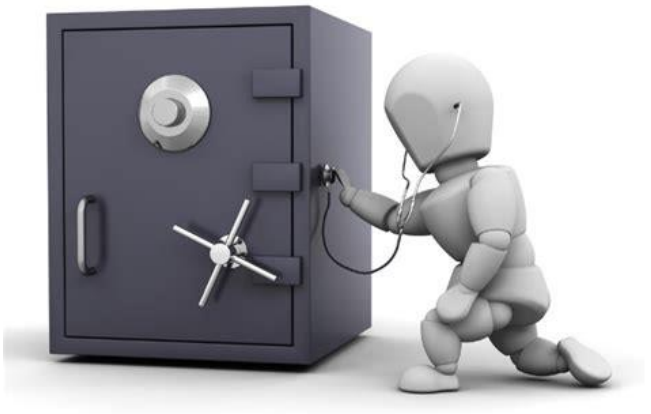
Sébastien Bardin
CEA, LIST, Université Paris-Saclay
France

Tamara Rezk
Inria
France

Context: Timing Attacks

Timing attacks: execution time of programs can leak secret information

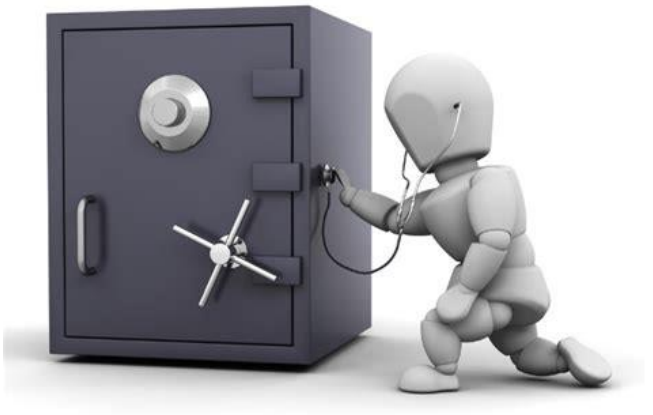
First timing attack in **1996** by Paul Kocher: full recovery of **RSA encryption key**

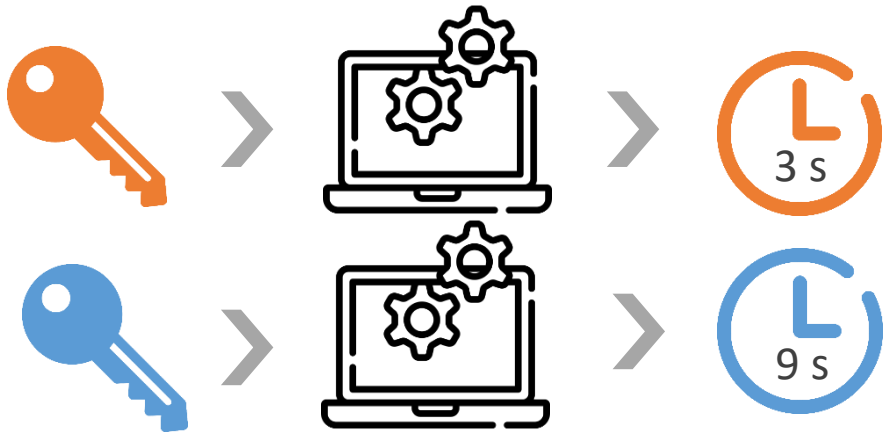


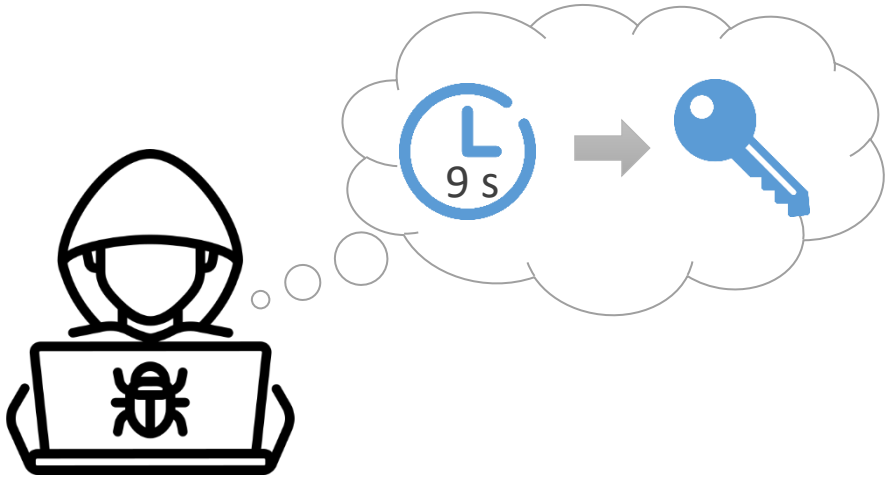
Context: Timing Attacks

Timing attacks: execution time of programs can leak secret information

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



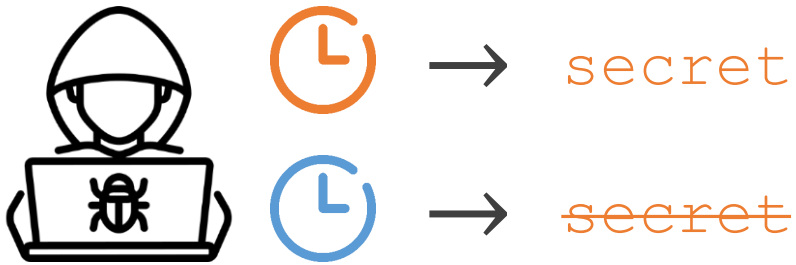




What can Influence the Execution Time?

Control Flow

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

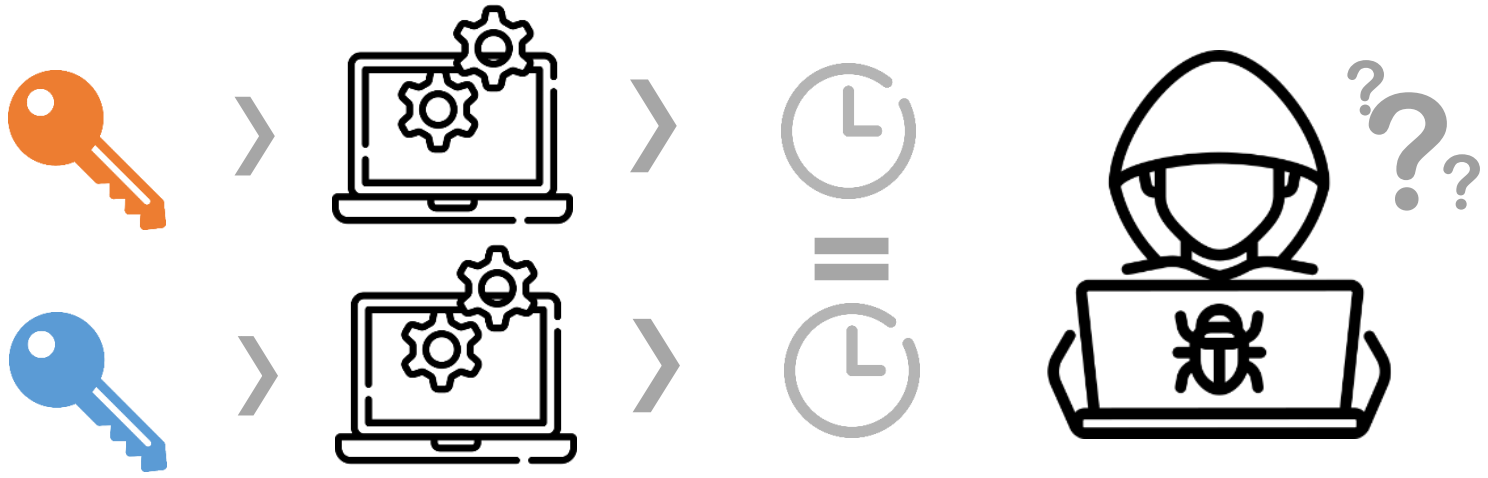


Memory Accesses

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

Protect Software with Constant-Time Programming

Constant-Time. Execution time is independent from secret input

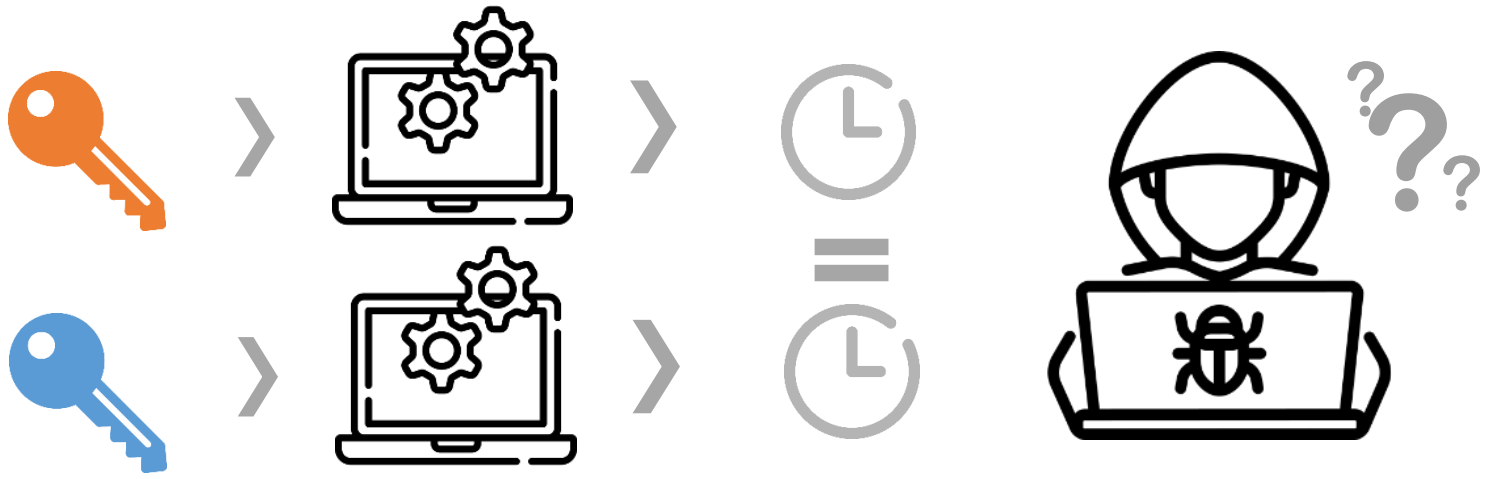


Protect Software with Constant-Time Programming

Constant-Time. Execution time is independent from secret input

→ Control-flow

→ Memory accesses

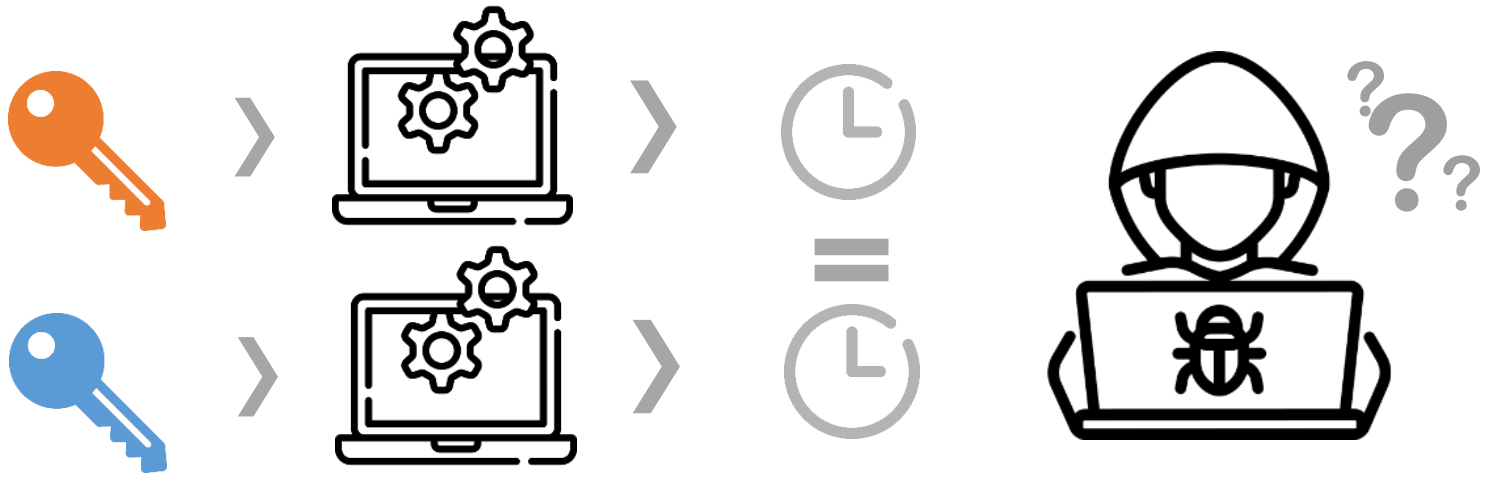


Protect Software with Constant-Time Programming

Constant-Time. Execution time is independent from secret input

→ Control-flow

→ Memory accesses



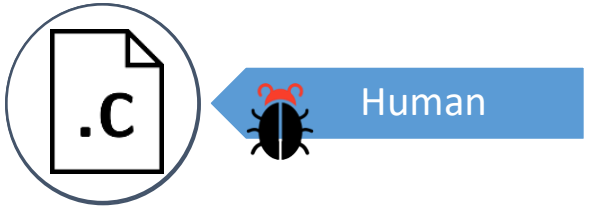
Property relating 2 execution traces (2-hypersafety)

Problem: Need Automated Verif.

Not easy to write CT code, avoid:

- Secret dependent control-flow
- Secret dependent memory accesses

Multiple failure points

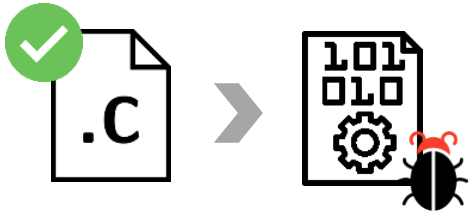


Problem: Need Automated Verif.

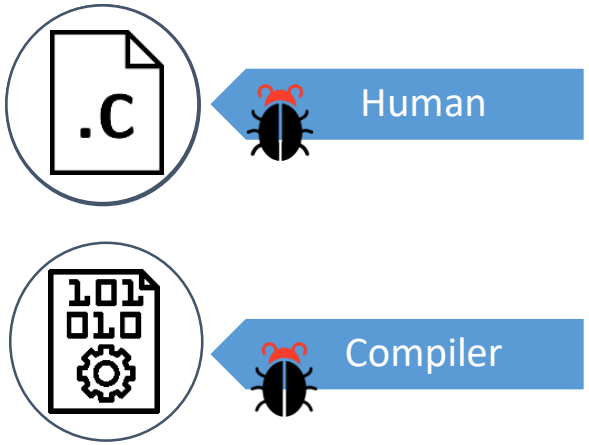
Not easy to write CT code, avoid:

- Secret dependent control-flow
- Secret dependent memory accesses

Compiler can introduce bugs [1]!



Multiple failure points



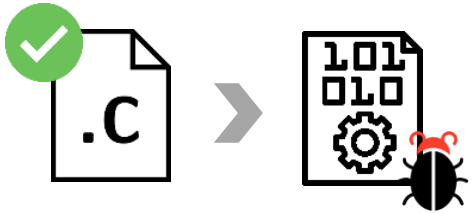
[1] “What you get is what you C”, Simon, Chisnall, and Anderson 2018

Problem: Need Automated Verif.

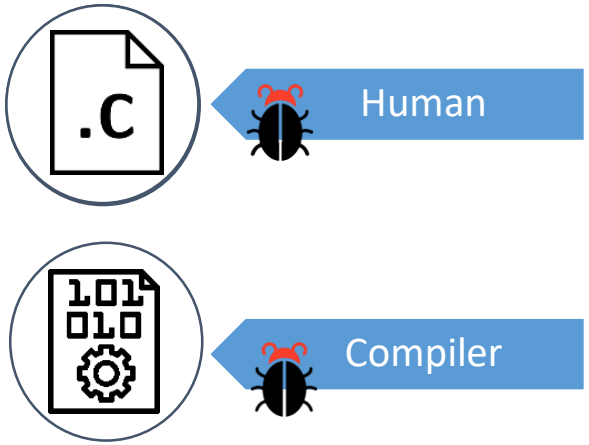
Not easy to write CT code, avoid:

- Secret dependent control-flow
- Secret dependent memory accesses

Compiler can introduce bugs [1]!



Multiple failure points



We need efficient automated verification tools!

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

Lots of verification tools for CT 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 CT violations ☹

+ Full proof of CT

[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] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "CacheD: Identifying cache-based timing channels in production software," in USENIX, 2017
[6] G. Doychev and B. Köpf, "Rigorous analysis of software countermeasures against cache attacks," in PLDI, 2017

Lots of verification tools for CT 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 CT violations ☹

+ Full proof of CT

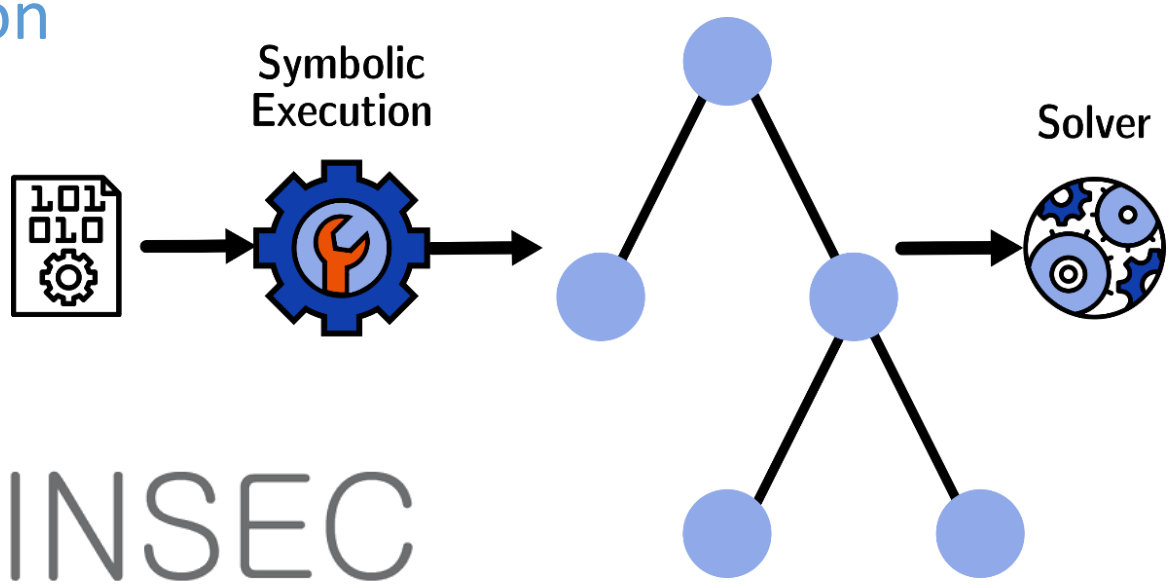
[1] J. Bacelar Almeida, M. Barbosa, J. S. Pinto, and B. Vieira, "Formal verification of side-channel countermeasures using self-composition," in Science of Computer Programming, 2013
[2] S. Blazy, D. Pichardie, and A. Trieu, "Verifying Constant-Time Implementations by Abstract Interpretation," in ESORICS, 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] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "CacheD: Identifying cache-based timing channels in production software," in USENIX, 2017
[6] G. Doychev and B. Köpf, "Rigorous analysis of software countermeasures against cache attacks," in PLDI, 2017

Bug-Finding & Bounded-Verification
Try Symbolic Execution

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

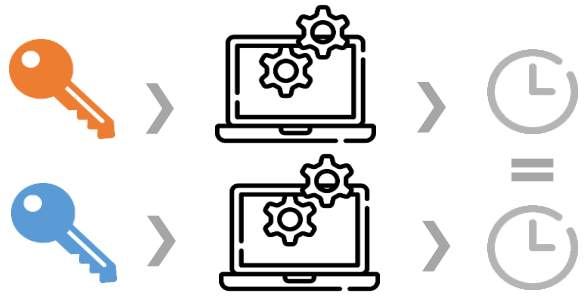
The KeY Project

BINSEC



Challenges for CT analysis

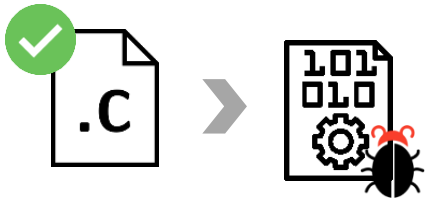
Property of 2 executions



→Model pairs of executions

Standard tools do not apply

Not necessarily preserved by compilers



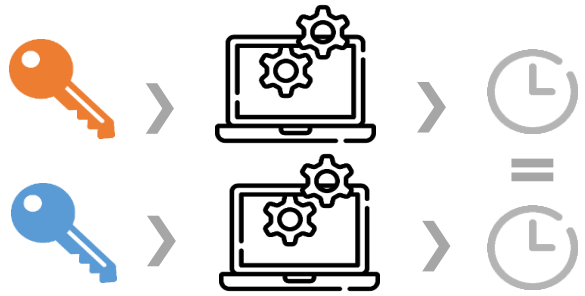
Compilation

→ Binary-analysis

Reason explicitly about memory

Challenges for CT analysis

Property of 2 executions



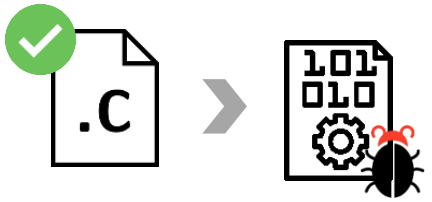
→ Efficiently model pairs of executions

Standard tools do not apply

RelSE

SE for pairs of traces with sharing

Not necessarily preserved by compilers



Compilation

→ Binary-analysis

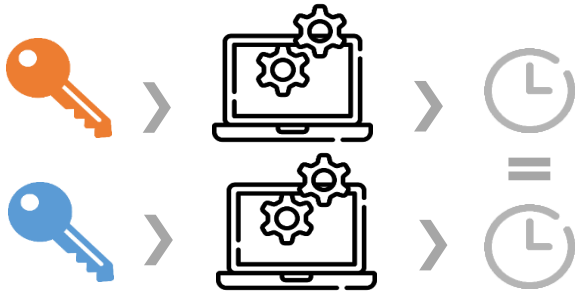
Reason explicitly about memory

Binary-level SE

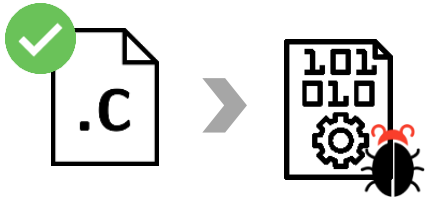
BINSEC

Challenges for CT analysis

Property of 2 executions



Not necessarily preserved by compilers



Compilation

→ Efficiently model pairs of executions

Standard tools do not apply

→ 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

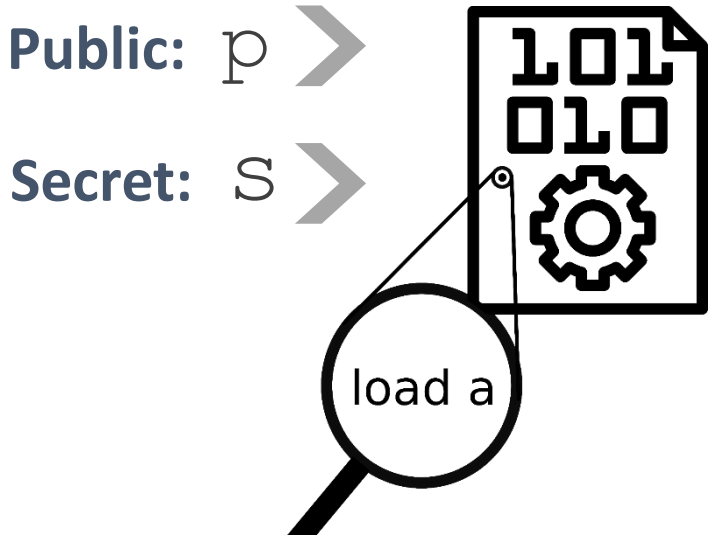
https://github.com/binsec/rel

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

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

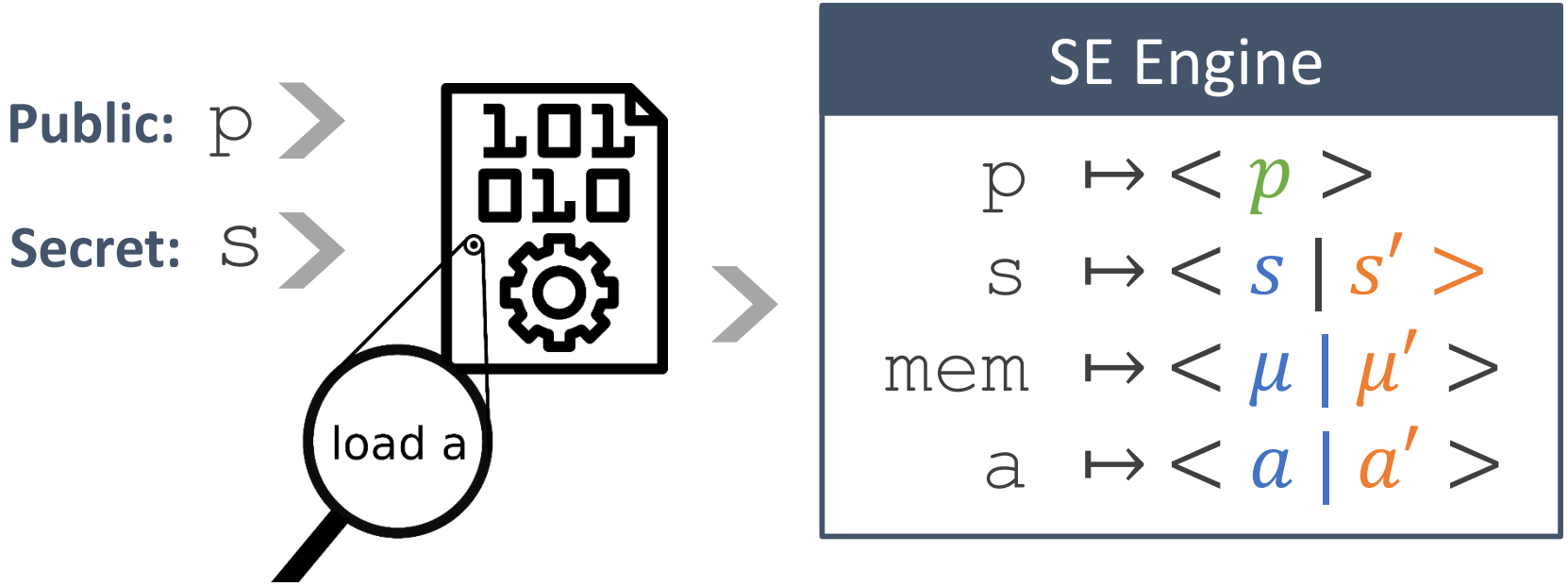
- Standard Approach: RelSE
- Our Approach: Binary-level RelSE

Relational Symbolic Execution [1,2]



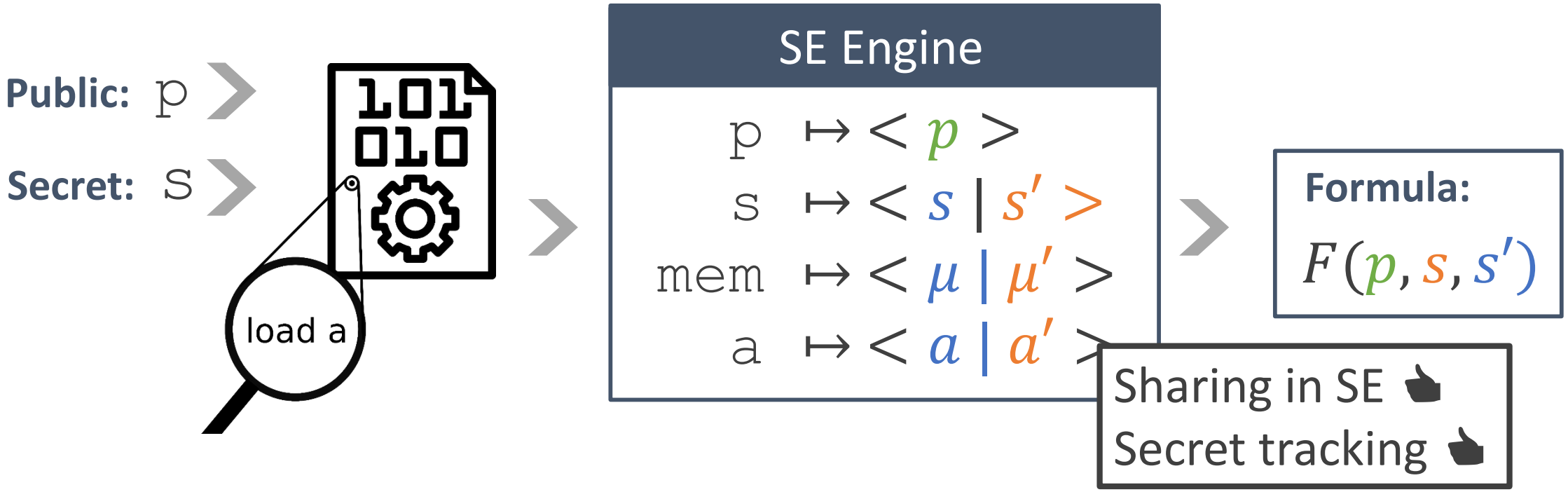
[1] “Shadow of a doubt”, Palikareva, Kuchta, and Cadar 2016
[2] “Relational Symbolic Execution”, Farina, Chong, and Gaboardi 2017

Relational Symbolic Execution [1,2]



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

Relational Symbolic Execution [1,2]

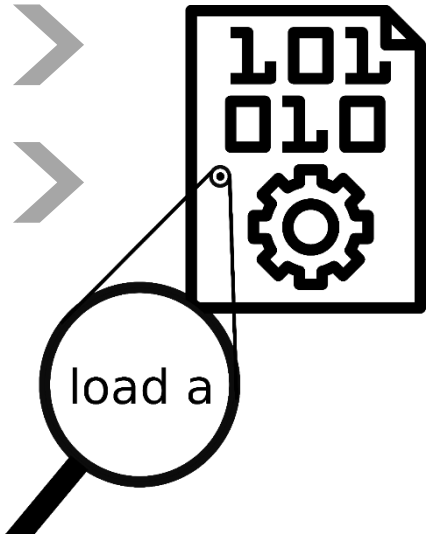


[1] “Shadow of a doubt”, Palikareva, Kuchta, and Cadar 2016
[2] “Relational Symbolic Execution”, Farina, Chong, and Gaboardi 2017

Relational Symbolic Execution [1,2]

Public: p

Secret: s



SE Engine

$$\begin{aligned} \texttt{p} &\mapsto < p > \\ \texttt{s} &\mapsto < s \mid s' > \\ \texttt{mem} &\mapsto < \mu \mid \mu' > \\ \texttt{a} &\mapsto < a \mid a' > \end{aligned}$$

Formula:

$$F(p, s, s')$$

Sharing in SE 👍

Secret tracking 👍

Question: Can a depend on secret s ?

Formula with sharing:

$$F(p, s, s') \wedge a \neq a'$$

Solver

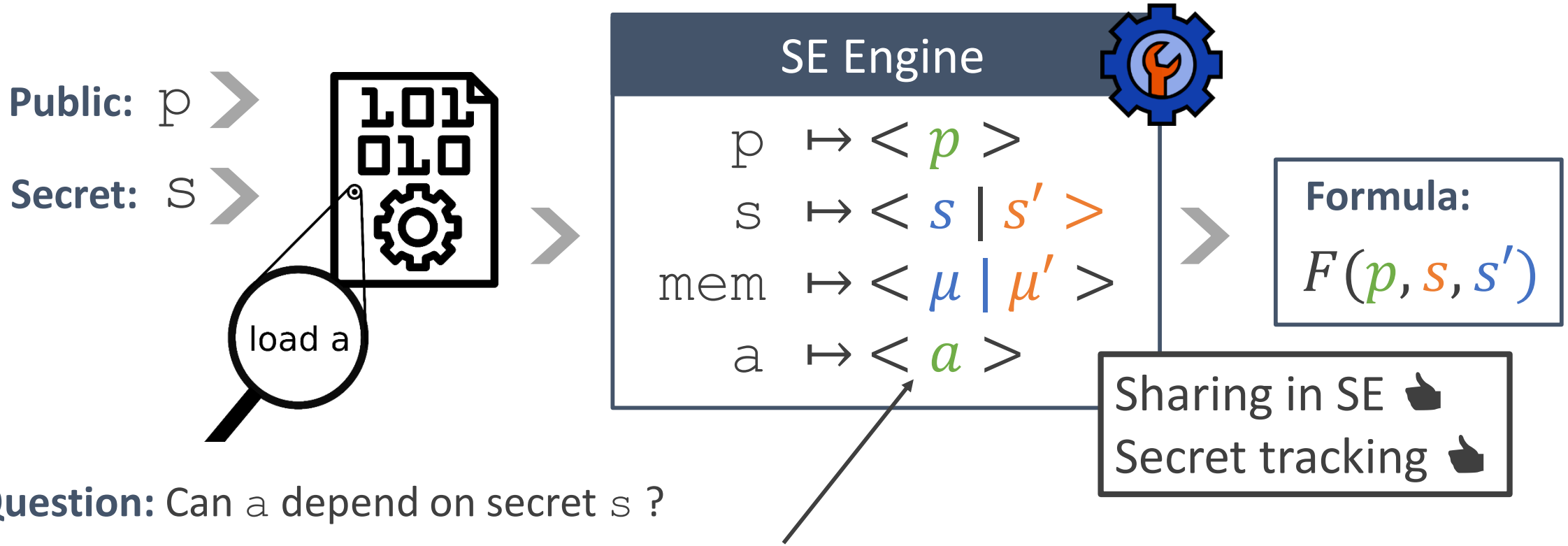
UNSAT

SAT 🐞

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

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

Relational Symbolic Execution [1,2]



Question: Can a depend on secret s ?

By definition, a does not depend on secrets

We spare a call to the solver !

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

Problem with RelSE at binary-level

Problem: Sharing fails at binary-level

- Memory is represented as a symbolic array $< \mu \mid \mu' >$
- Duplicated at the beginning of SE
- Duplicate all load operations

In our experiments, we show that standard RelSE does not scale on binary code

Our approach: Binary-level RelSE

FlyRow: on-the-fly read-over-write

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

→ Avoids resorting to duplicated memory

[1] “Arrays Made Simpler”, Farinier et al. 2018

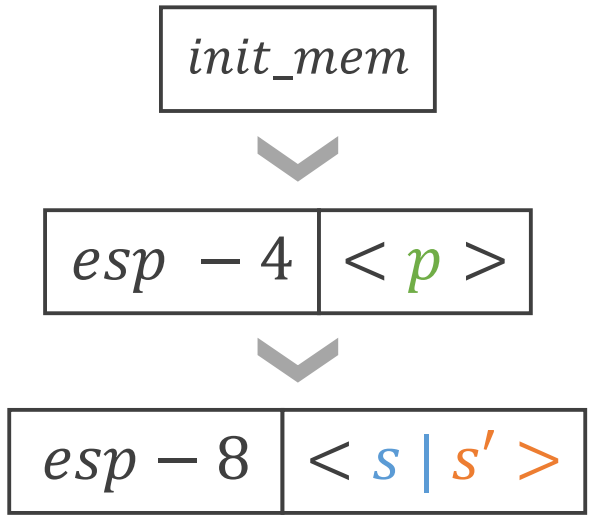
Our approach: Binary-level RelSE

FlyRow: on-the-fly read-over-write

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

→ Avoids resorting to duplicated memory

Memory as the history of stores.



[1] “Arrays Made Simpler”, Farinier et al. 2018

Our approach: Binary-level RelSE

FlyRow: on-the-fly read-over-write

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

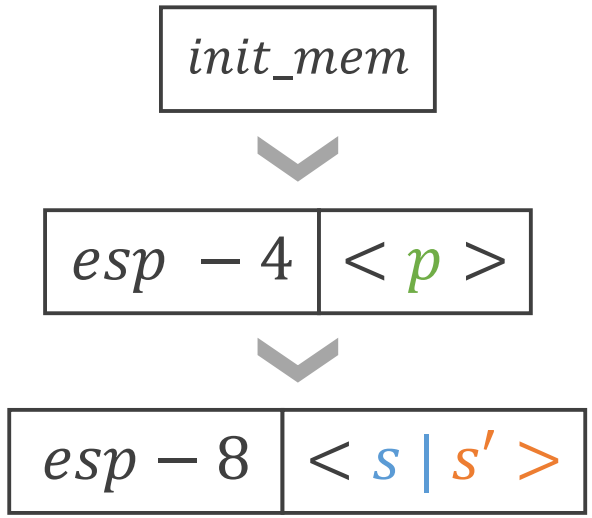
→ Avoids resorting to duplicated memory

Example.

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

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

Memory as the history of stores.



[1] "Arrays Made Simpler", Farinier et al. 2018

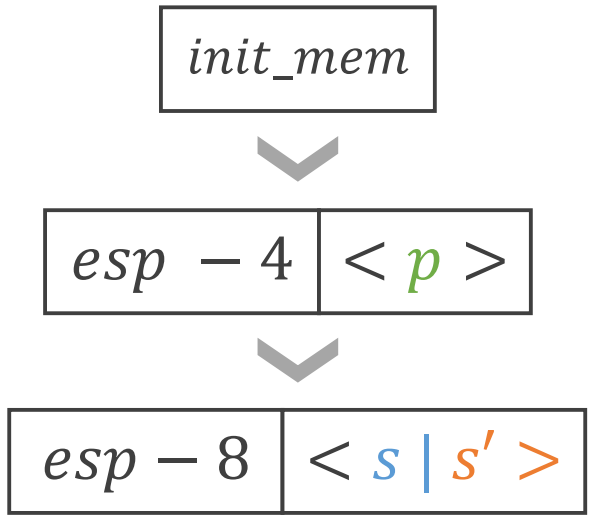
Our approach: Binary-level RelSE

FlyRow: on-the-fly read-over-write

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

→ Avoids resorting to duplicated memory

Memory as the history of stores.



Example.

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

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

+ simplifications for efficient syntactic disequality checks

[1] "Arrays Made Simpler", Farinier et al. 2018

Dedicated optimizations for CT

Untainting

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

- Better tracking secret dependencies
- Spare more queries

Fault-Packing

Pack queries along basic-blocks

- Reduces number of queries
- Useful for CT analysis (lots of queries)

Experimental evaluation

Experimental evaluation

https://github.com/binsec/rel

Benchmark

- Utility functions from OpenSSL & HACL*
- Cryptographic primitives from libsodium, BearSSL, OpenSSL, HACL*

Experiments

RQ1. Effective on real crypto?

→ 338 programs: 54M unrolled instr in 2h

RQ2. Comparison vs. RelSE

→ 700× faster

RQ3. Genericity

→ gcc/clang compilers & x86/ARM binaries

+ More in paper

RQ1: Effectiveness

	Programs	Static Instr.	Unrolled Instr.	Time	Success
Secure (Bounded-Verif)	**296**	64k	23M	**46min**	100%
Insecure (Bug-Finding)	**42**	6k	22k	**40min**	100%

- First automatic CT analysis of these programs at binary-level
- Can find vulnerabilities in binaries compiled from CT source
- Found **3 bugs** that **slipped through prior LLVM analysis**

RQ2: Comparison with 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)

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
- New results:
  - 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 paper

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

Conclusion

Conclusion

Binsec/Rel

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

https://github.com/binsec/rel

- Dedicated optimizations for RelSE at binary-level
  → Sharing for scaling
- Binsec/Rel, binary-level tool for constant-time analysis
- Verification of crypto libraries at binary-level + new bugs introduced by compilers out-of reach of LLVM verification

After Binsec/Rel

Detection of Spectre attacks

Binsec/Haunted

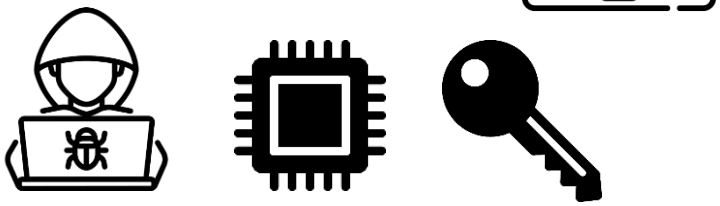
NDSS SYMPOSIUM/2021

https://github.com/binsec/haunted

New framework to verify secret-erasure (WIP)

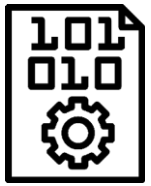
Credits

Icons made by Freepik from www.flaticon.com



Icons made by bqlqn from www.flaticon.com

Icons made by Becris from www.flaticon.com



Icons made by scrip from www.flaticon.com

From draw.io