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

2nd International KLEE Workshop on Symbolic Execution,
10-11 June, 2021

Published at IEEE Symposium on Security and Privacy 2020

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

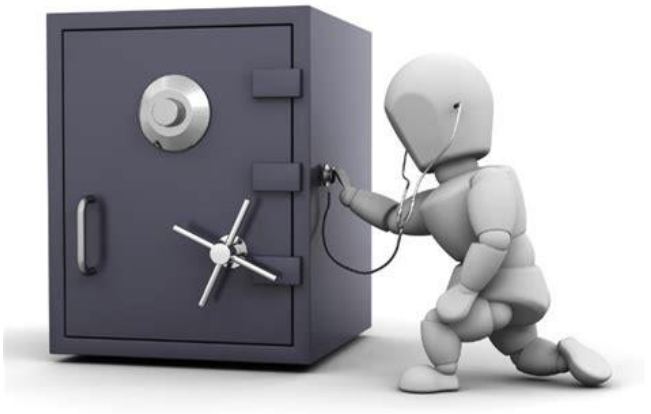
Sébastien Bardin
CEA, LIST, Université Paris-Saclay
France

Tamara Rezk
Inria
France

Context: Timing Attacks

Timing attacks: execution time of programs can leak secret information

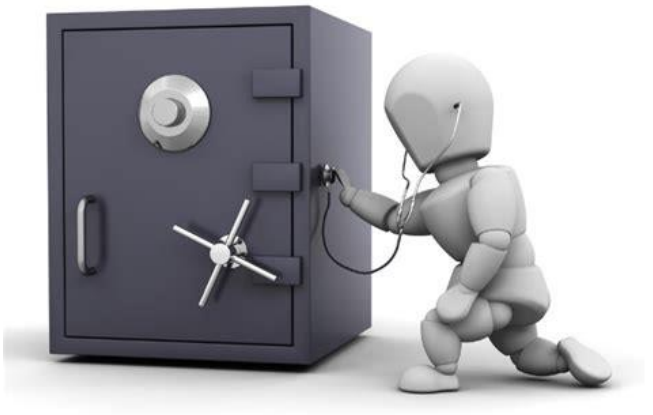
First timing attack in **1996** by Paul Kocher: full recovery of **RSA encryption key**

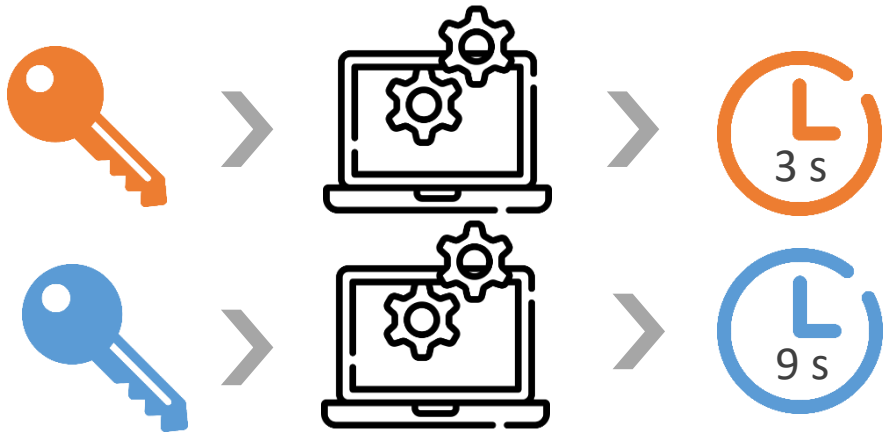


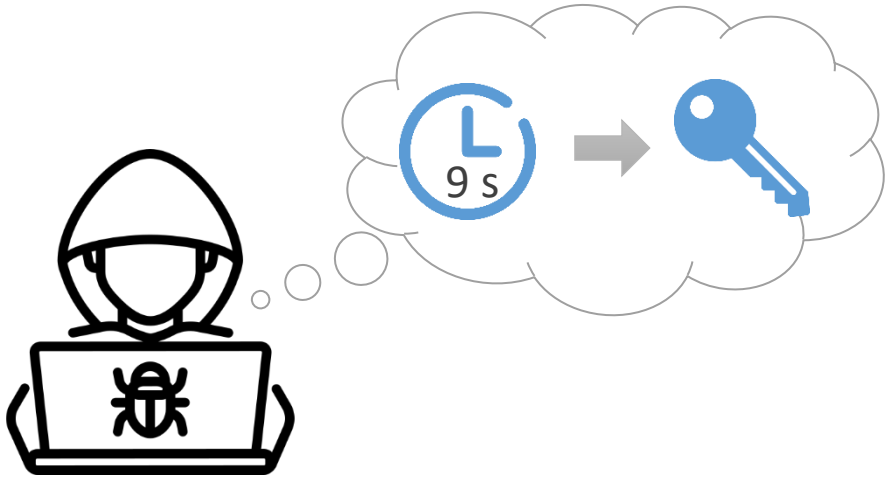
Context: Timing Attacks

Timing attacks: execution time of programs can leak secret information

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

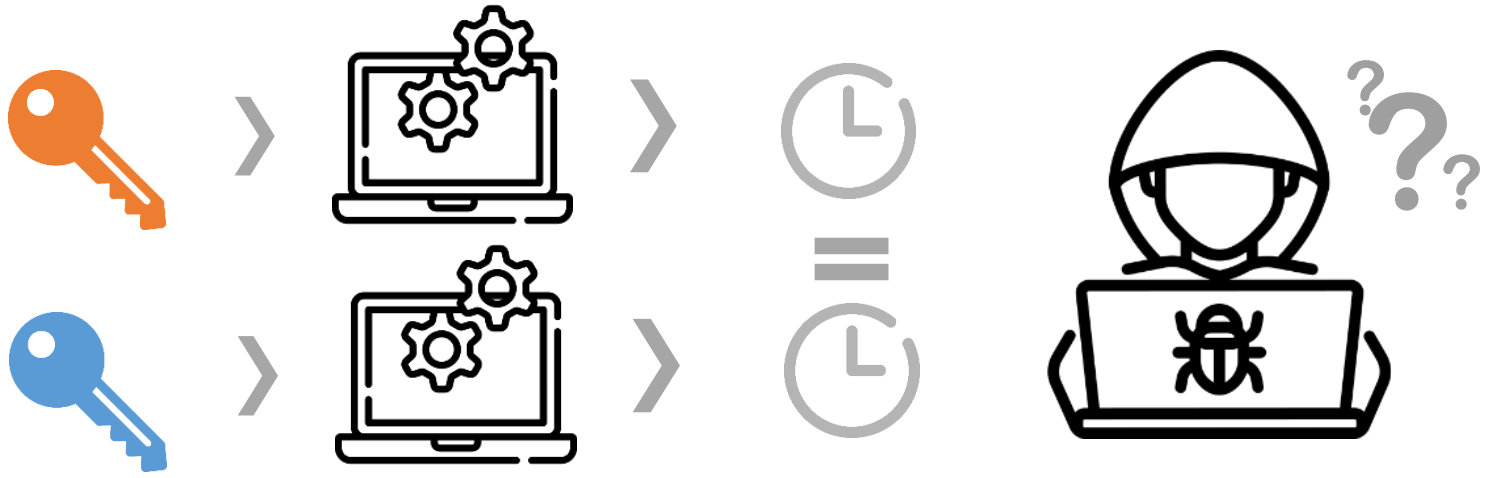






Protect Software with Constant-Time Programming

Constant-Time. Execution time is independent from secret input

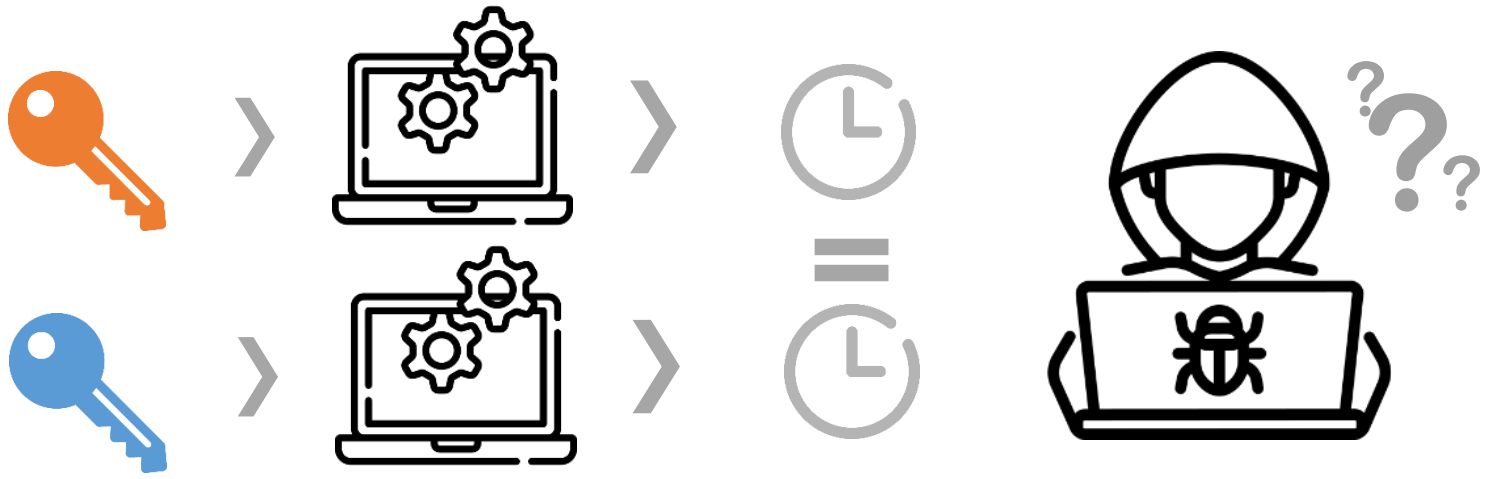


Protect Software with Constant-Time Programming

Constant-Time. Execution time is independent from secret input

→ Control-flow

→ Memory accesses

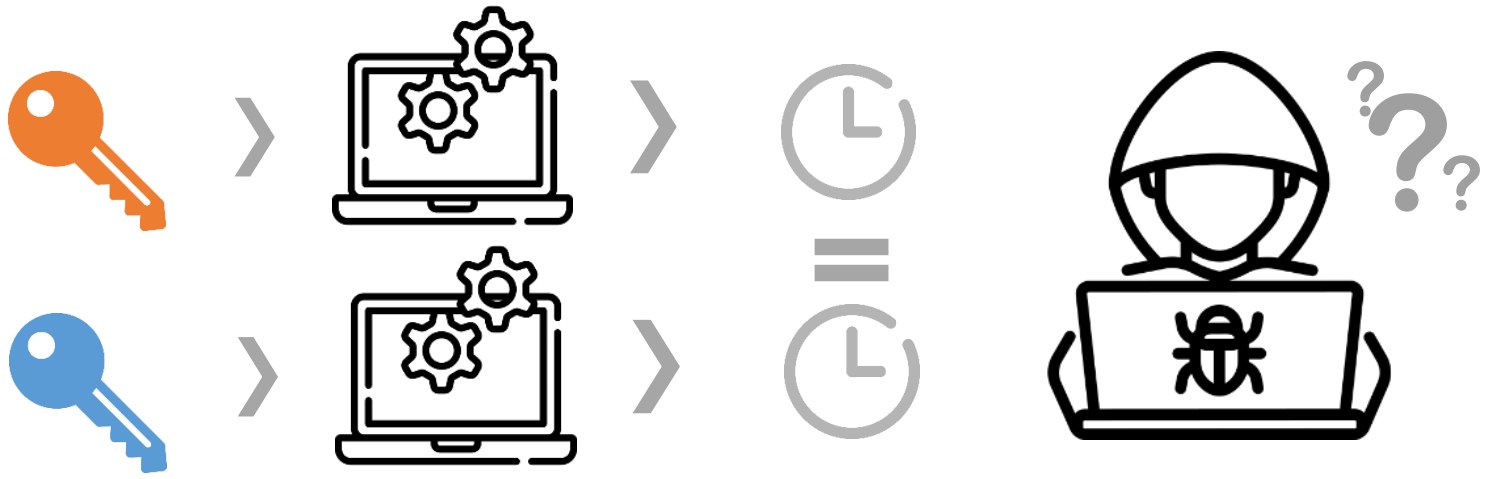


Protect Software with Constant-Time Programming

Constant-Time. Execution time is independent from secret input

→ Control-flow

→ Memory accesses



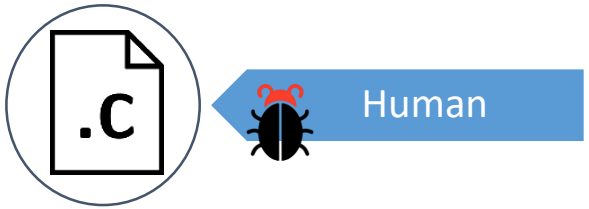
Property relating 2 execution traces (2-hypersafety)

Problem: Need Automated Verif.

Execution time is not easy to determine

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

Multiple failure points

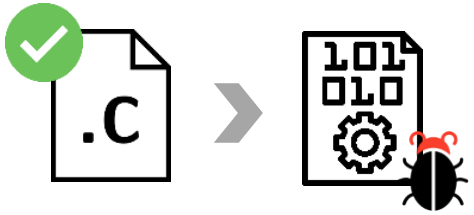


Problem: Need Automated Verif.

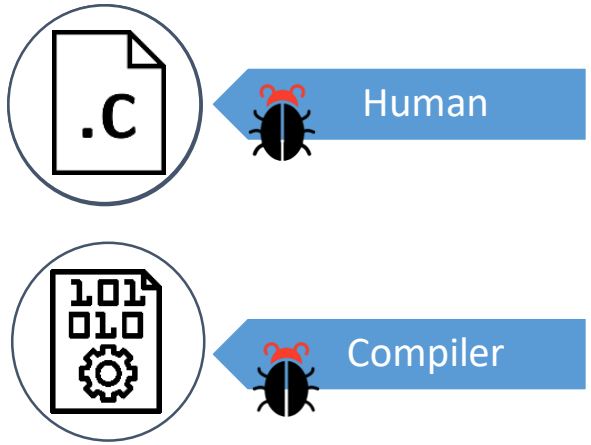
Execution time is not easy to determine

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

Compiler can introduce bugs [1]!



Multiple failure points



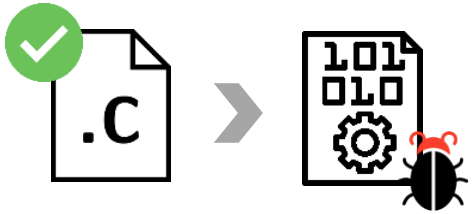
[1] “What you get is what you C”, Simon, Chisnall, and Anderson 2018

Problem: Need Automated Verif.

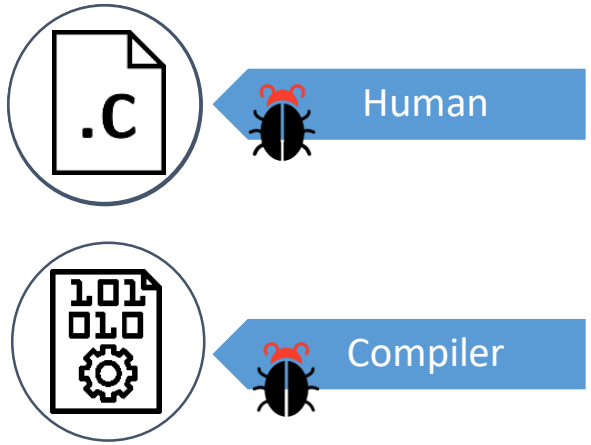
Execution time is not easy to determine

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

Compiler can introduce bugs [1]!



Multiple failure points



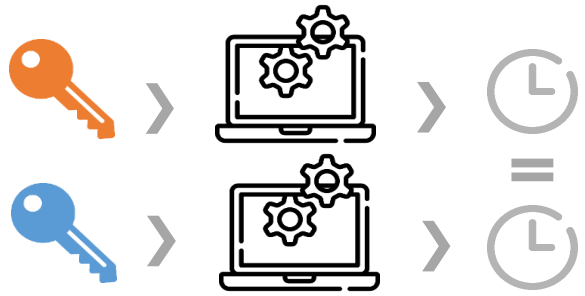
Not easy to write constant-time programs

We need efficient automated verification tools!

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

Challenges for CT analysis

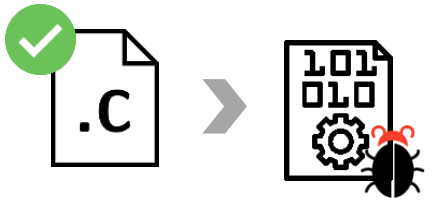
Property of 2 executions



→ Efficiently model pairs of executions

Standard tools do not apply

Not necessarily preserved by compilers



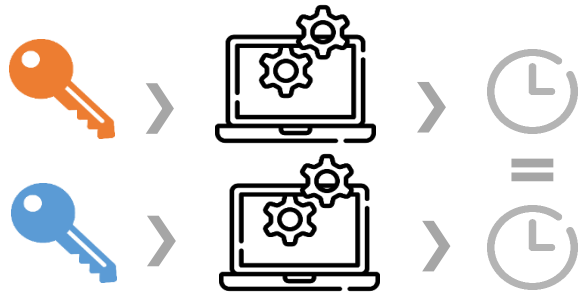
Compilation

→ Binary-analysis

Reason explicitly about memory

Challenges for CT analysis

Property of 2 executions



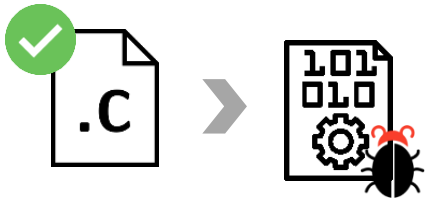
→ Efficiently model pairs of executions

Standard tools do not apply

RelSE

SE for pairs of traces with sharing

Not necessarily preserved by compilers



Compilation

→ Binary-analysis

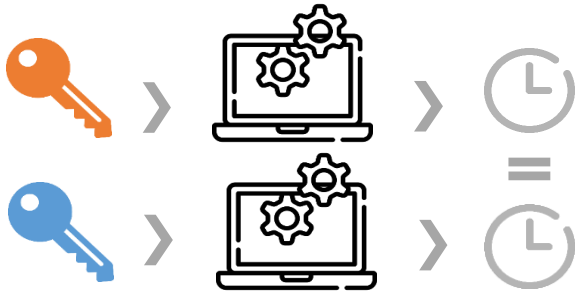
Reason explicitly about memory

Binary-level SE

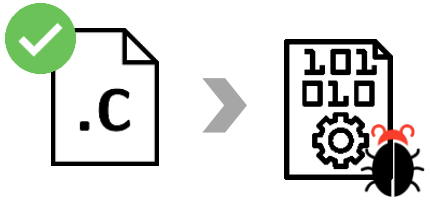
BINSEC

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

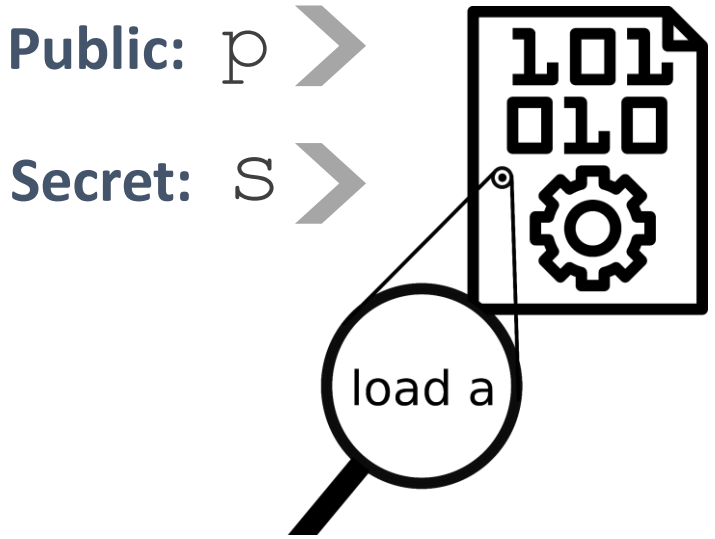
https://github.com/binsec/rel

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*

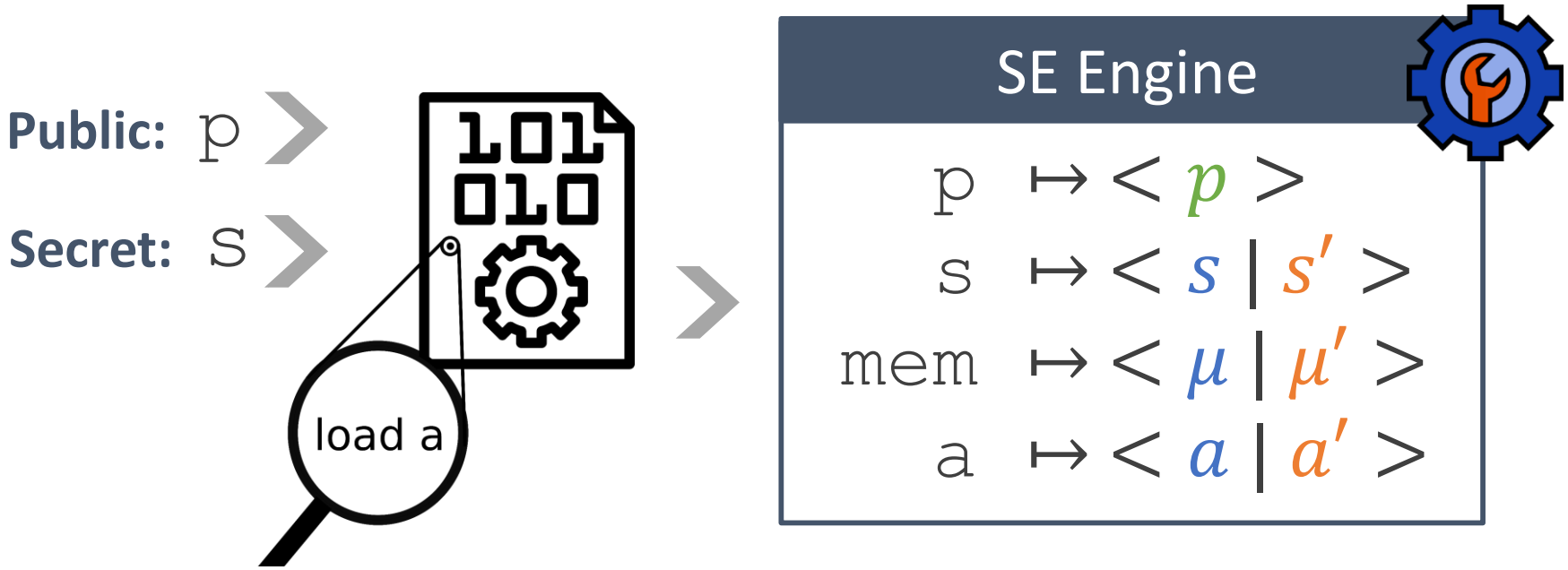
- Relational Symbolic Execution (RelSE)
- Our Approach: Binary-level RelSE

Relational Symbolic Execution [1,2]



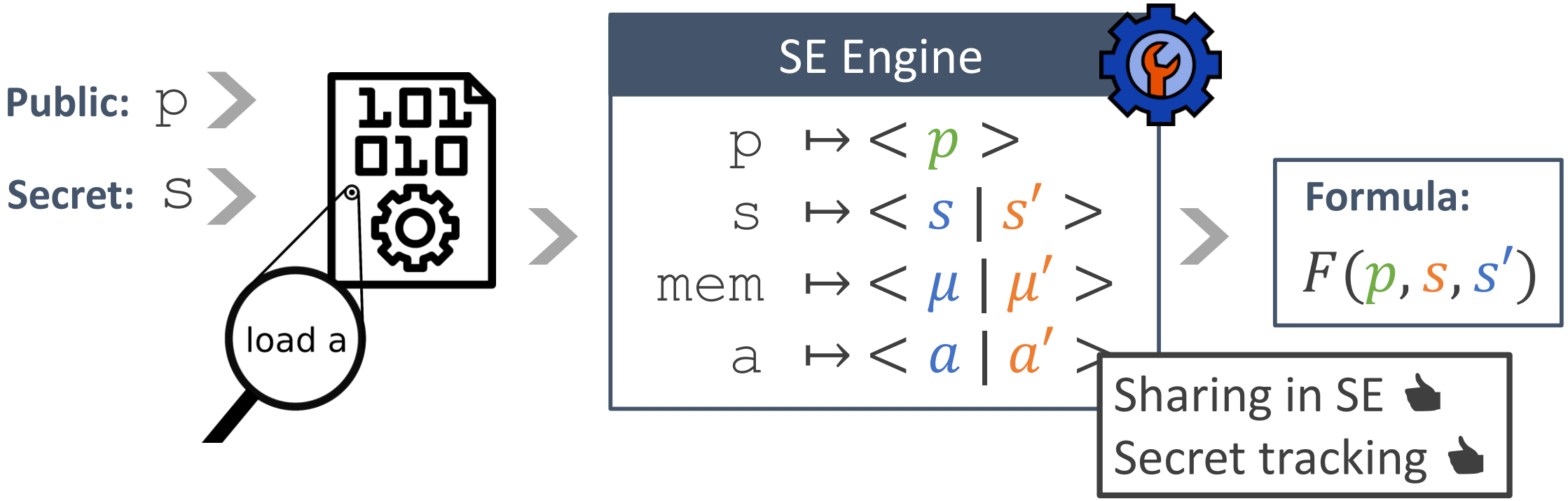
[1] “Shadow of a doubt”, Palikareva, Kuchta, and Cadar 2016
[2] “Relational Symbolic Execution”, Farina, Chong, and Gaboardi 2017

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]

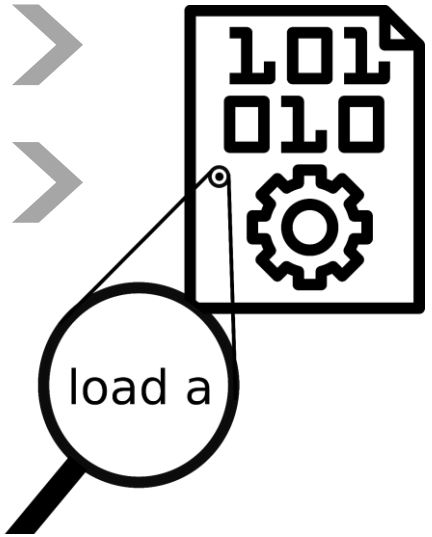


[1] “Shadow of a doubt”, Palikareva, Kuchta, and Cadar 2016
[2] “Relational Symbolic Execution”, Farina, Chong, and Gaboardi 2017

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

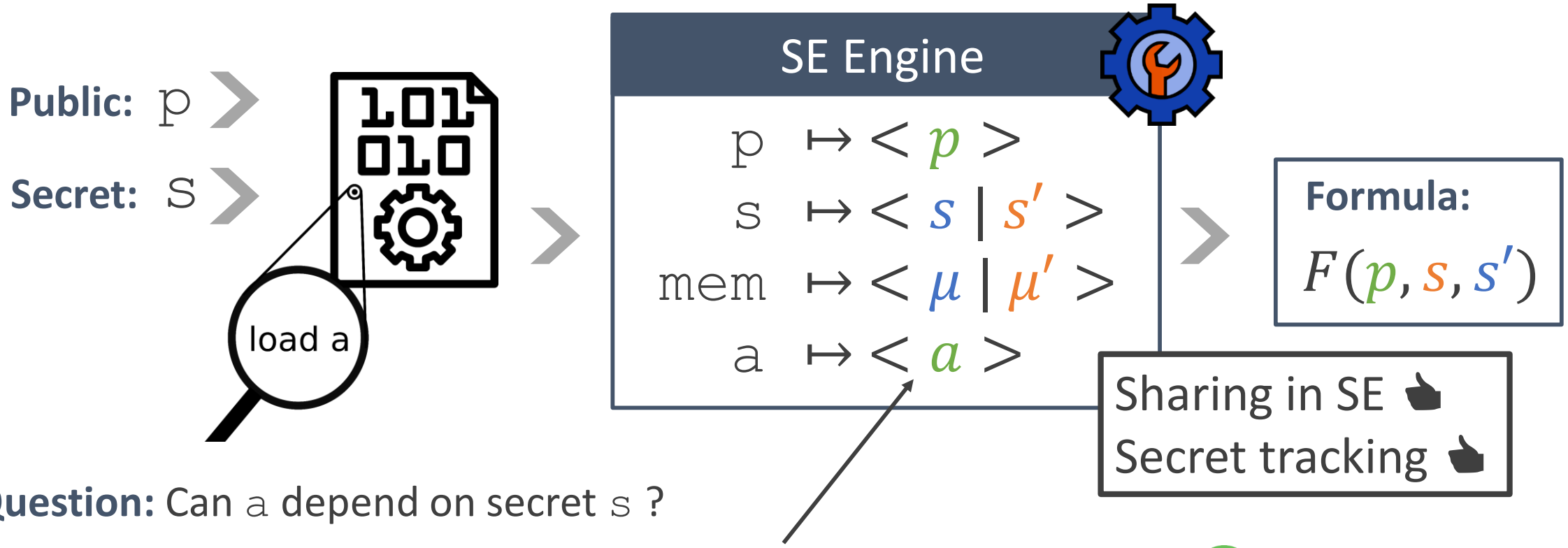
UNSAT

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

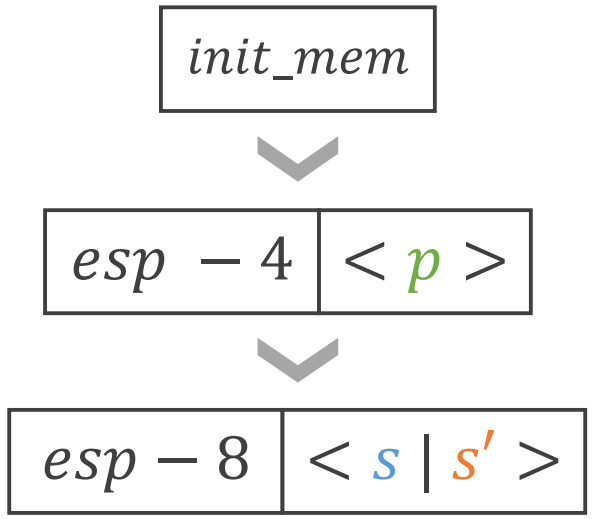
Our approach: Binary-level RelSE

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

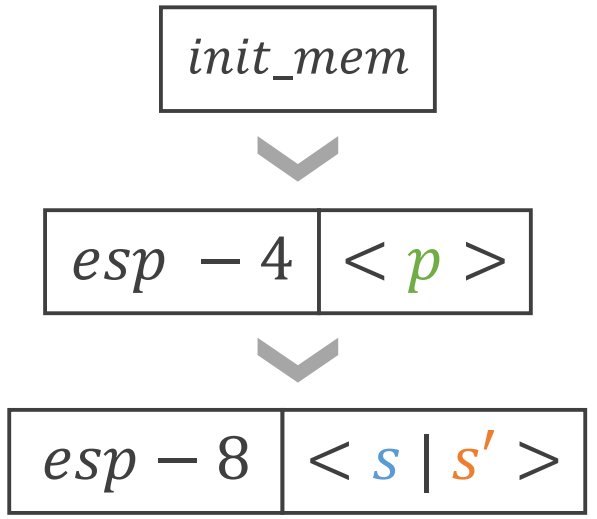
→ Avoids resorting to duplicated memory

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

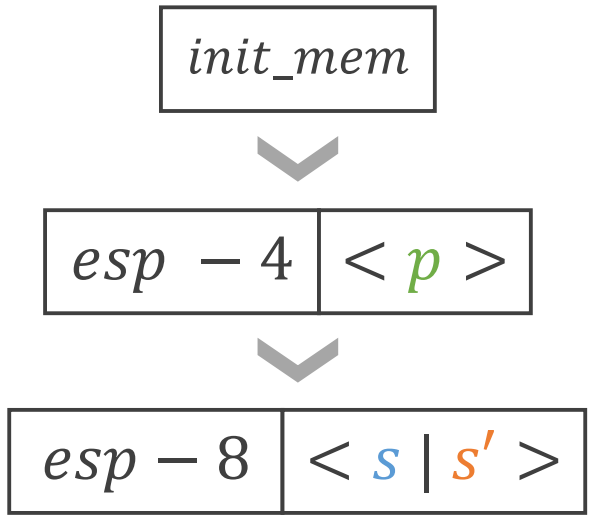
Our approach: Binary-level RelSE

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

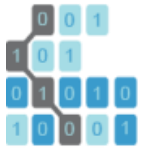
$< select\ \mu\ (esp - 4) \mid select\ \mu'(esp - 4) >$

+ simplifications for efficient syntactic disequality checks

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

Experimental evaluation

Experimental evaluation



Binsec/Rel

https://github.com/binsec/rel

Experiments

RQ1. Effective on real crypto?

→ 338 programs: 54M unrolled instr in 2h

RQ2. Comparison vs. RelSE

→ 700× faster

+ More in paper

Benchmark

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

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**

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

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

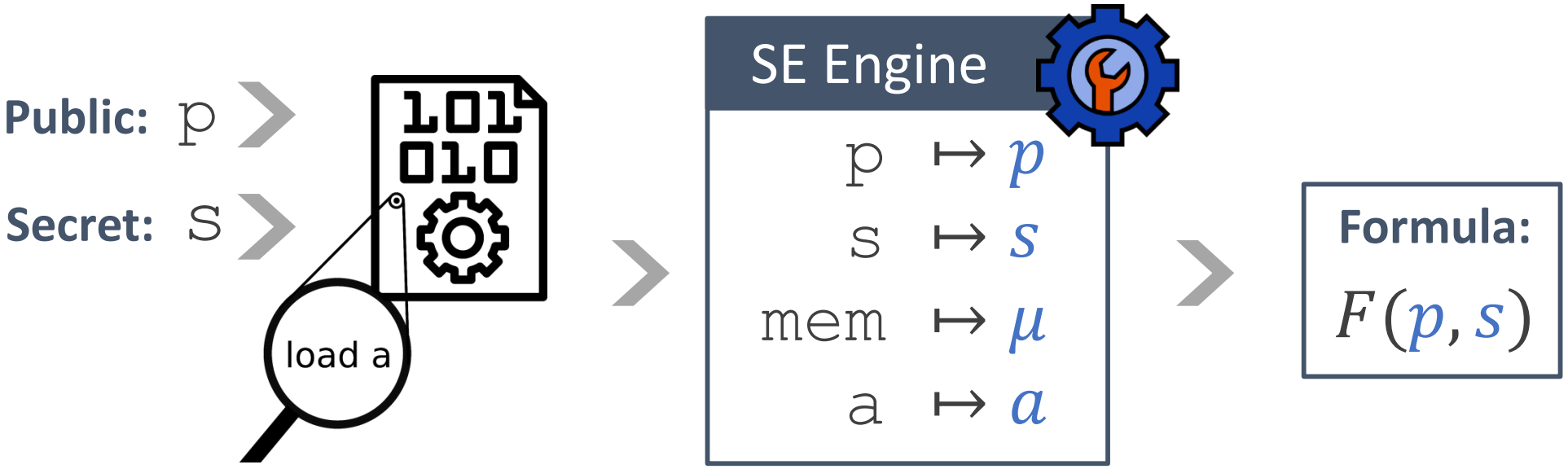
https://github.com/binsec/haunted

New framework to verify secret-erasure (WIP)

I'm also looking for a postdoc for next year ☺ !

- Standard Approach: Self-Composition
- Better Approach: RelSE
- Our Approach: Binary-level RelSE

Standard Approach: Self-Composition [1,2]



Question: Can a depend on secret s ?

Self-composed formula:

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

UNSAT

SAT

[1] "Verifying information flow properties of firmware using symbolic execution", Subramanyan et al. 2016
[2] "CaSym: Cache aware symbolic execution for side channel detection and mitigation", Brotzman et al. 2019

Standard Approach: Self-Composition

Limitations of self-composition:

High number of insecurity queries: conditional + memory access

Why?

- No sharing between two executions
- Does not keep track of secret-dependencies

SE for constant-time via self-composition does not scale

+ we show it in our experiments