Binsec/RelSE

Efficient Constant-Time Analysis of Binary-Level Code with Relational Symbolic Execution

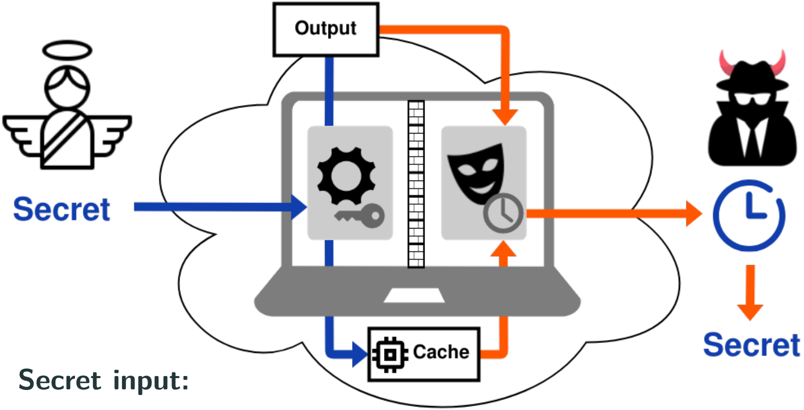
Lesly-Ann Daniel
CEA LIST, France

Sébastien Bardin
CEA LIST, France

Tamara Rezk
INRIA, France

IEEE Symposium on Security and Privacy
May 19, 2020

Problem: Protecting Secrets against Timing Attacks



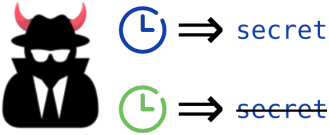
Secret input:
sensitive data,
cryptographic key, etc.

Attacker's goal:
Recover info on secret

What Can Influence the Execution Time?

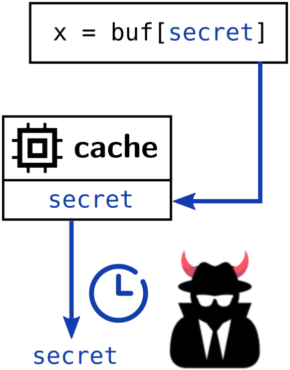
Control Flow

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



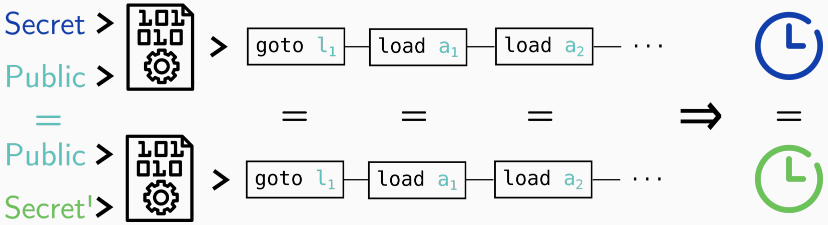
Secret-dependent control-flow
can leak secret

Memory Accesses



Secret-dependent memory access
can leak secret

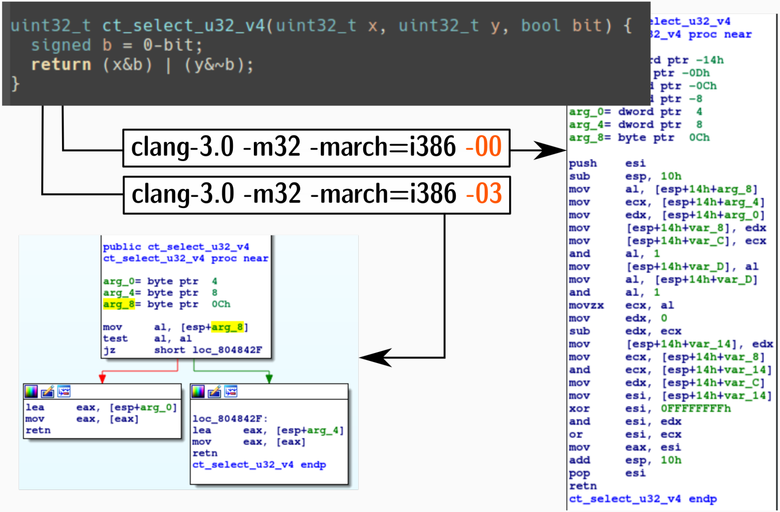
Solution: Constant-Time Programming (CT)



Definition: Two executions with the same *public* input must have the same control flow and memory accesses regardless of the value of the *secrets*.

Programming discipline to protect against timing attacks

Constant-Time is Generally not Preserved by Compilers [1]



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

The Need for Automatic Analysis

Constant-time is important to protect against timing attacks
but writing constant-time code is tricky

Constant-time is generally not preserved by compiler [2]

- Binary-level is harder than higher-level analysis (C, llvm)
- Explicit representation of memory

Need efficient binary-level reasoning

Constant-time is about pairs of executions (2-hypersafety)

- Standard tools do not directly apply

Need dedicated tools that scale for analyzing pairs of traces

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

Lots of Verification Tools for Constant-Time

- For high level code:
  - Source code [Bacelar Almeida et al. 2013], [Blazy, Pichardie, and Trieu 2017]
  - LLVM code [Almeida et al. 2016], [Brotzman et al. 2019]
- For binary code:
  - Sacrifice bounded-verification [Wang et al. 2017], [Subramanyan et al. 2016]
  - Sacrifice bug-finding [Doychev and Köpf 2017]

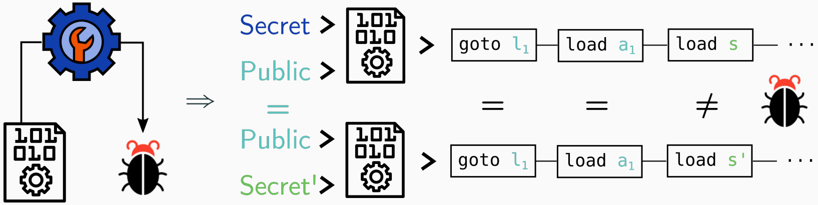
Lots of Verification Tools for Constant-Time

- For high level code:
  - Source code [Bacelar Almeida et al. 2013], [Blazy, Pichardie, and Trieu 2017]
  - LLVM code [Almeida et al. 2016], [Brotzman et al. 2019]
- For binary code:
  - Sacrifice bounded-verification [Wang et al. 2017], [Subramanyan et al. 2016]
  - Sacrifice bug-finding [Doychev and Köpf 2017]

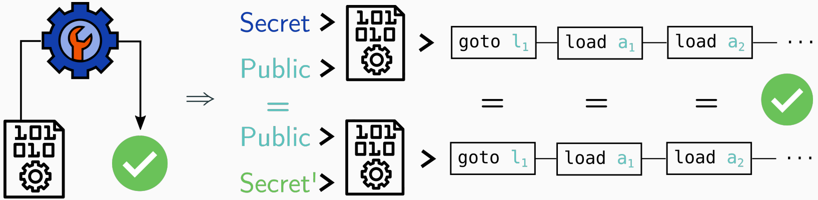
Our goal: *Design efficient tool to analyze constant-time at binary-level for bounded-verification and bug-finding*

Definition: Bug-Finding & Bounded-Verif for Constant-Time

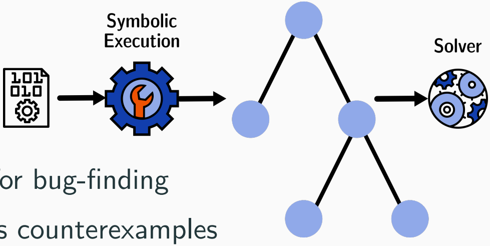
Bug-Finding (BF): a bug found in the analysis is a real bug.



Bounded-Verification (BV): when no bugs are found in the analysis then there is no bug in the program up to a certain bound.



Bug-Finding and Bounded-Verification? Try Symbolic Execution



Symbolic Execution (SE)

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

BINSEC

The KeY Project

Adapt SE for Constant-Time: Technical Key Insights

Proposal: Adapt symbolic execution for constant-time			
Binary-Level	Bug Finding	Bound. Verif.	Scalability

Build on Relational SE [1,2]

Execute two programs in the same symbolic execution instance

We show that it does not scale at binary-level

New: Binary-Level RelSE

Dedicated optimizations for binary-level and constant-time analysis

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

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

Contributions

Dedicated optims for constant-time analysis at binary-level

With formal definitions & proofs

Binsec/Rel: First efficient BF & BV tool for CT

700× speedup compared to standard RelSE

Large Scale Experiments (338 cryptographic binaries)

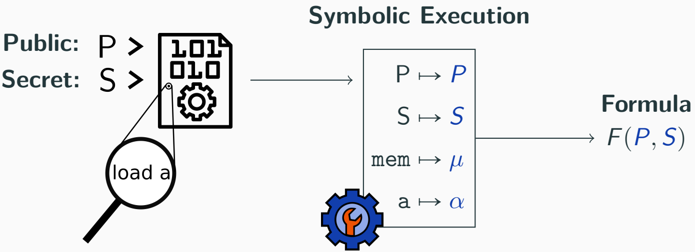
- *New proofs* on binary previously done on C/LLVM/F*
- Replay of known bugs (e.g. Lucky13)

Extension of study on preservation of CT by compilers [4]

Discover *new bugs* introduced by `gcc -O0` and `clang` backend passes, out of reach of previous tools for LLVM

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

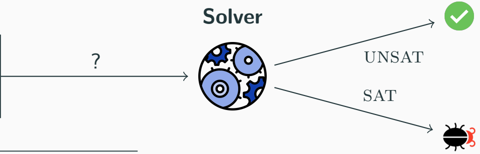
Standard Approach (e.g. [1,2]): Symbolic Execution for Constant-Time via Self-Composition



Question: Can “a” leak w.r.t. constant-time policy?

Self-Composed Formula
→ two executions

$$F(P,S) \wedge F(P',S') \wedge P = P' \wedge \alpha \neq \alpha'$$



[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 (e.g. [1,2]): Symbolic Execution for Constant-Time via Self-Composition

Limitation of self-composition:

High number of insecurity queries: conditional + memory access

Why?

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

Symbolic-execution for constant-time via self-composition does not scale

We show it in our experiments

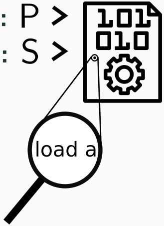
[1] "Verifying information flow properties of firmware using symbolic execution", Subramanyan et al. 2016

[2] "CaSym: Cache aware symbolic execution for side channel detection and mitigation", Brotzman et al. 2019

Better Approach: Relational Symbolic Execution [1,2]

Public: P >

Secret: S >



Relational SE

$\mathsf{P} \mapsto \langle P \rangle$

$\mathsf{S} \mapsto \langle S \mid S' \rangle$

$\mathtt{mem} \mapsto \langle \mu \mid \mu' \rangle$

$\mathsf{a} \mapsto \langle \alpha \mid \alpha' \rangle$

Formula

$F(P, S, S')$

- Sharing
- Secret-tracking

Question: Can “a” leak w.r.t. constant-time policy?

Formula → two executions

$$\boxed{F(P, S, S') \wedge \alpha \neq \alpha'} \xrightarrow{?} \textbf{Solver}$$

UNSAT

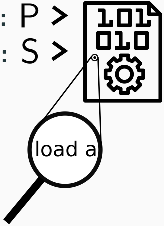
SAT

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

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

Better Approach: Relational Symbolic Execution [1,2]

Public: P >
Secret: S >



Relational SE

$\mathtt{P} \mapsto \langle P \rangle$

$\mathtt{S} \mapsto \langle S \mid S' \rangle$

$\mathtt{mem} \mapsto \langle \mu \mid \mu' \rangle$

$\mathtt{a} \mapsto \langle \alpha \rangle$

Formula

$F(P, S, S')$

- Sharing
- Secret-tracking

Question: Can "a" leak w.r.t. constant-time policy?

No $\longrightarrow$ ✓

Spared solver call

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

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

Better Approach: Relational Symbolic Execution [1,2]

Problem: sharing fails at binary-level

- Memory is represented as a symbolic array variable $\langle \mu \mid \mu' \rangle$
- Duplicated at the beginning of RelSE
- Duplicate all the load operations

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

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

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

Our Idea: Dedicated Simplifications for Binary-Level RelSE

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

- Build on *read-over-write* [1]
- Relational expressions in the memory
- Simplify load operations on-the-fly

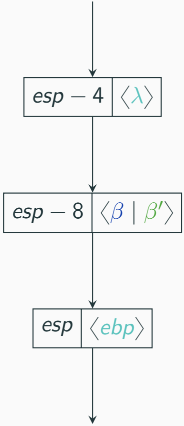
$\rightarrow$ Avoids resorting to the duplicated memory

Example

"`load esp-4`" returns $\langle \lambda \rangle$ instead of
$\langle \mathit{select}\ \mu\ (\mathit{esp} - 4) \mid \mathit{select}\ \mu\ (\mathit{esp} - 4) \rangle$

$+$ *simplifications in the paper*

Memory as the history of stores



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

Dedicated Optimizations for Constant-Time Analysis

Untainting

Use solver responses to transform $\langle \alpha \mid \alpha' \rangle$ to $\langle \alpha \rangle$.

- Track secret-dependencies more precisely
- Spare insecurity queries

Fault-Packing

Pack insecurity queries along a basic-block

- Reduces number of queries
- Useful for constant-time (lot of insecurity queries)

Binsec/Rel: Experimental Evaluation

RQ1 **Effectiveness**: BINSEC/REL for bounded-verif. & bug-finding of constant-time on real-world crypto. binaries?

RQ2 **Comparison vs. Std Approaches** BINSEC/REL vs. RelSE?

RQ3 **Genericity**: several architectures / compilers?

RQ4 **Impact of Simplifications** *FlyRow*, *Untainting*, *Fault-Packing*?

RQ5 **Comparison vs. Std SE**: BINSEC/REL vs. Std SE & *FlyRow* with SE?

Effectiveness for Bounded-Verif & Bug-Finding (RQ1)

338 samples of cryptographic binaries taken from [1,2,3]

- *utility functions* from OpenSSL & HACL*
- *cryptographic primitives*: tea, donna, salsa20, chacha20, etc
- *libraries*: libsodium, BearSSL, OpenSSL, HACL*

	#Prog	#Instr	#Instr$_{unrol}$	Time	Success
Secure (BV)	296	64k	**23M**	**53min**	100%
Insecure (BF)	42	6k	31k	**69min**	100%

First automatic CT-analysis at binary level
Can find vulnerabilities in binaries compiled from CT sources
Found 3 bugs that slipped through prior analysis

[1] "Verifying Constant-Time Implementations.", Almeida et al. 2016

[2] "Verifying Constant-Time Implementations by Abstract Interpretation", Blazy, Pichardie, and Trieu 2017

[3] "HACL*", Zinzindohoué et al. 2017

Scalability: Comparison with RelSE (RQ2)

	#I	#I/s	Time	⌛	✓	✗
RelSE	320k	5.4	16h30	14	283	42
BINSEC/REL	22.8M	**3861**	**1h38**	**0**	296	42

Total on 338 cryptographic samples (secure & insecure)

Timeout set to 1h

700× faster than RelSE

No ⌛, even on large programs (e.g. `donna`)

Effect of Compiler Optimizations on Constant-Time (RQ1/RQ3)

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` compiler + `clang v7.1`, ARM binaries
- **New Results**
  - `gcc -O0` can introduce violations in programs but as optimization level increases, it tends to remove violations (contrary to `clang`)
  - `clang` backend passes introduce violations in programs deemed secure by CT-verification tools for LLVM
  - *More in paper*

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

Conclusion

Efficient Bug-Finding & Bounde-Verification for Constant-Time at Binary-Level

Bug-Finding ✓ & **Bounded-Verif.** ✓

no over-approx. & no under-approx.

Sharing for **Scaling**

- Relational SE
- Dedicated optimizations

Binary-level

- No source code needed
- Do not rely on compiler

Experiments on 338 crypto binaries

- new proofs at binary level
- new bugs (`gcc-O0` and `clang` backend)
- automate manual study on compilers

BINSEC RelSE

Thank You for your Attention