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()

Memory Accesses

\[ x = \text{buf}[\text{secret}] \]

Secret-dependent control-flow can leak secret

Secret-dependent memory access can leak secret
**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]

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**

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**.
Symbolic Execution (SE)

- Leading formal method for bug-finding
- Finds real bugs & reports counterexamples
- Can also perform bounded-verification
- Scales well on binary code
Proposal: Adapt symbolic execution for constant-time

<table>
<thead>
<tr>
<th>Binary-Level</th>
<th>Bug Finding</th>
<th>Bound. Verif.</th>
<th>Scalability</th>
</tr>
</thead>
</table>

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

Contributions

Dedicated optims for constant-time analysis at binary-level
With formal definitions & proofs

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

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

\[ F(P, S) \land F(P', S') \land P = P' \land \alpha \neq \alpha' \]

Solver

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

Better Approach: Relational Symbolic Execution [1,2]

Public: \( P \)
Secret: \( S \)

Relational SE

\[
\begin{align*}
P & \mapsto \langle P \rangle \\
S & \mapsto \langle S | S' \rangle \\
\text{mem} & \mapsto \langle \mu | \mu' \rangle \\
a & \mapsto \langle \alpha | \alpha' \rangle
\end{align*}
\]

Formula

\[ F(P, S, S') \]

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

Formula \( \rightarrow \) two executions

Solver

\[ F(P, S, S') \land \alpha \neq \alpha' \]

Better Approach: Relational Symbolic Execution [1,2]

Public: \( P \rightarrow \) 10101010
Secret: \( S \rightarrow \) gear

load a

Relational SE

- \( P \mapsto \langle P \rangle \)
- \( S \mapsto \langle S \mid S' \rangle \)
- \( \text{mem} \mapsto \langle \mu \mid \mu' \rangle \)
- \( a \mapsto \langle \alpha \rangle \)

Formula

\[ F(P, S, S') \]

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

No

\textit{Spared solver call}

Better Approach: Relational Symbolic Execution [1,2]

**Problem: sharing fails at binary-level**

- Memory is represented as a symbolic array variable $\langle \mu | \mu' \rangle$
- 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

---

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

→ Avoids resorting to the duplicated memory

Example

“load esp-4” returns $\langle \lambda \rangle$ instead of $\langle select \mu (esp - 4) | select \mu (esp - 4) \rangle$

+ simplifications in the paper

### Untainting

Use solver responses to transform \( \langle \alpha \ | \ \alpha' \rangle \) 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)
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*

<table>
<thead>
<tr>
<th></th>
<th>#Prog</th>
<th>#Instr</th>
<th>#Instr_unrol</th>
<th>Time</th>
<th>Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure (BV)</td>
<td>296</td>
<td>64k</td>
<td>23M</td>
<td>53min</td>
<td>100%</td>
</tr>
<tr>
<td>Insecure (BF)</td>
<td>42</td>
<td>6k</td>
<td>31k</td>
<td>69min</td>
<td>100%</td>
</tr>
</tbody>
</table>

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

### Scalability: Comparison with RelSE (RQ2)

<table>
<thead>
<tr>
<th></th>
<th>#I</th>
<th>#I/s</th>
<th>Time</th>
<th>▼</th>
<th>✓</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>RelSE</td>
<td>320k</td>
<td>5.4</td>
<td>16h30</td>
<td>14</td>
<td>283</td>
<td>42</td>
</tr>
<tr>
<td>Binsec/Rel</td>
<td>22.8M</td>
<td><strong>3861</strong></td>
<td><strong>1h38</strong></td>
<td><strong>0</strong></td>
<td>296</td>
<td>42</td>
</tr>
</tbody>
</table>

*Total on 338 cryptographic samples (secure & insecure)*

*Timeout set to 1h*

700× faster than RelSE

No ▼, even on large programs (e.g. donna)
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 -00 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*

Conclusion

Efficient Bug-Finding & Bounded-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-00 and clang backend)
• automate manual study on compilers
Thank You for your Attention