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

- Efficient constant-time verification at binary-level (overview)
- Adaptation to detect Spectre attacks

Journée 2021 du GT "Méthodes Formelles pour la Sécurité"
March, 16th 2021
Binsec/Rel: Efficient constant-time verification at binary-level (overview)
**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
**Constant-Time.** Execution time is independent from secret input

Property relating 2 execution traces (2-hypersafety)
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 Verification Tools

Execution time is not easy to determine

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

Multiple failure points:
- Human
- Compiler
- Hardware
Problem: Need Automated Verification Tools

Execution time is not easy to determine

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

Not easy to write constant-time programs
We need efficient automated verification tools!
Challenges for CT analysis

Property of 2 executions

Not necessarily preserved by compilers

Compilation
Challenges for CT analysis

Property of 2 executions

→ Efficiently model **pairs of executions**

RelSE (SE for pairs of traces with sharing)

for Bug-Finding & Bounded-Verif

Not necessarily preserved by compilers

→ **Binary-analysis** (harder)

Compilation
Challenges for CT analysis

Property of 2 executions

→ Efficiently model pairs of executions

Not necessarily preserved by compilers

→ Binary-analysis (harder)

RelSE (SE for pairs of traces with sharing)

for Bug-Finding & Bounded-Verif

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**

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>New Tool</th>
<th>Application: crypto verif.</th>
</tr>
</thead>
</table>
| Dedicated optimizations for RelSE at binary-level: maximize sharing in memory (x700 speedup) | **BINSEC/REL** First efficient tool for *BF & BV* of CT at *binary-level* | From OpenSSL, BearSSL, libsodium
296 verified binaries
3 new bugs introduced by compilers from verified source |
Haunted RelSE: detect Spectre vulnerabilities
Spectre haunting our code

Spectre attacks (2018)

• Exploit speculative execution in processors
• Affect almost all processors
• Attackers can force misspeculations: 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

Exploits conditional branch predictor

```plaintext
if idx < size {
    v = tab[idx]
    leak(v)
}
```

- `idx` is attacker controlled
- `content of tab` is public
- `leak(v)` encodes `v` to cache

Regular execution

- Conditional bound check ensures `idx` is in bounds
- `v` contains public data
Spectre-PHT

Exploits conditional branch predictor

```cpp
if idx < size {
    v = tab[idx]
    leak(v)
}
```

- `idx` is attacker controlled
- `content of tab is public`
- `leak(v) encodes v to cache`

**Regular 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: Loads can speculatively bypass prior stores

Regular 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 \( a \neq b \)
- \( \text{leak}(v) \) encodes \( v \) to cache
Spectre-STL: Loads can speculatively bypass prior stores

Regular execution + Transient Executions

\[
\begin{align*}
\text{store } a &\quad s \\
\text{store } a &\quad p \\
\text{store } b &\quad q \\
v = \text{load } a \\
\text{leak}(v) \\
\text{leak}(p)
\end{align*}
\]
\[
\begin{align*}
\text{store } a &\quad s \\
\text{store } a &\quad p \\
v = \text{load } a \\
\text{store } b &\quad q \\
\text{leak}(v) \\
\text{leak}(p)
\end{align*}
\]

• where \( s \) is secret, \( p \) and \( q \) are public
• where \( a \neq b \)
• \( \text{leak}(v) \) encodes \( v \) to cache
Spectre-STL: Loads can speculatively bypass prior stores

Regular execution + Transient Executions

\[
\text{store } a \ s \\
\text{store } a \ p \\
\text{store } b \ q \\
v = \text{load } a \\
\text{leak}(v) \\
\text{leak}(p)
\]

\[
\text{store } a \ s \\
\text{store } a \ p \\
v = \text{load } a \\
\text{store } b \ q \\
\text{leak}(v) \\
\text{leak}(p)
\]

\[
\text{store } a \ s \\
v = \text{load } a \\
\text{store } a \ p \\
\text{store } b \ q \\
\text{leak}(v) \\
\text{leak}(s)
\]

- where \( s \) is secret, \( p \) and \( q \) are public
- where \( a \neq b \)
- \( \text{leak}(v) \) encodes \( v \) to cache
Spectre-STL: Loads can speculatively bypass prior stores

Regular execution + Transient Executions

\[
\begin{align*}
\text{store } a \ s & \quad \text{store } a \ s \\
\text{store } a \ p & \quad \text{store } a \ p \\
\text{store } b \ q & \quad \text{store } a \ p \\
\text{v} = \text{load } a & \quad \text{v} = \text{load } a \\
\text{leak(v)} & \quad \text{leak(v)} \\
\text{leak(p)} & \quad \text{leak(p)} \\
\text{leak(s)} & \quad \text{leak(s)} \\
\text{leak(init_mem[a])} & \quad \text{leak(init_mem[a])}
\end{align*}
\]

- where \( s \) is secret, \( p \) and \( q \) are public
- where \( a \neq b \)
- \( \text{leak(v)} \) encodes \( v \) to cache
Execution time is not easy to determine

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

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.)

<table>
<thead>
<tr>
<th>Semantics</th>
<th>Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular semantics</td>
<td>14</td>
</tr>
<tr>
<td>Speculative semantics (Spectre-STL)</td>
<td>37M</td>
</tr>
</tbody>
</table>
Goal: New verification tools for Spectre

**Goal.** We need new verification tools to detect Spectre attacks!

**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 😞

<table>
<thead>
<tr>
<th></th>
<th>Target</th>
<th>Spectre-PHT</th>
<th>Spectre-STL</th>
</tr>
</thead>
<tbody>
<tr>
<td>KLEESpectre [1]</td>
<td>LLVM</td>
<td>😊</td>
<td>-</td>
</tr>
<tr>
<td>SpecuSym [2]</td>
<td>LLVM</td>
<td>😊</td>
<td>-</td>
</tr>
<tr>
<td>FASS [3]</td>
<td>Binary</td>
<td>😞</td>
<td>-</td>
</tr>
<tr>
<td>Spectector [4]</td>
<td>Binary</td>
<td>😞</td>
<td>-</td>
</tr>
<tr>
<td>Pitchfork [5]</td>
<td>Binary</td>
<td>😊</td>
<td>😞</td>
</tr>
</tbody>
</table>

**Legend**

😊 Good perfs. on crypto
😊 Good on small programs
😢 Limited perfs. On crypto
😢 Limited to small programs

LLVM analysis might miss SCT violations 😞

---

## No efficient verification tools for Spectre?

<table>
<thead>
<tr>
<th>Target</th>
<th>Spectre-PHT</th>
<th>Spectre-STL</th>
</tr>
</thead>
<tbody>
<tr>
<td>KLEESpectre [1]</td>
<td>LLVM</td>
<td>☺</td>
</tr>
<tr>
<td>SpecuSym [2]</td>
<td>LLVM</td>
<td>☺</td>
</tr>
<tr>
<td>FASS [3]</td>
<td>Binary</td>
<td>☹</td>
</tr>
<tr>
<td>Spectector [4]</td>
<td>Binary</td>
<td>☹</td>
</tr>
<tr>
<td>Pitchfork [5]</td>
<td>Binary</td>
<td>☺, ☹</td>
</tr>
<tr>
<td>Binsec/Haunted</td>
<td>Binary</td>
<td>☺, ☹</td>
</tr>
</tbody>
</table>

**Legend**
- ☺: Good perfs. on crypto
- ☹: Good on small programs
- ☹: Limited perfs. On crypto
- ☹: Limited to small programs
- ☹: LLVM analysis might miss SCT violations

---

Contributions

Haunted RelSE optimization
• Model transient and regular behaviors at the same time
  • Spectre-PHT: pruning redundant paths
  • Spectre-STL: pruning + encoding to merge paths
• 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-world 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
Symbolic execution. An illustration.

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

2 regular paths

```
\pi \land c
\pi \land \neg c
```

foo

bar
Explicit RelSE for Spectre PHT

**Spectre-PHT.** Conditional branches can be executed speculatively

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

Fork execution into 4 at conditionals:
- 2 regular branches
- 2 transient branches (until max speculation depth)

On regular and transient branches:
- Verify no secret can leak.

(e.g. KLEESpectre)
Haunted RelSE for Spectre PHT

**Spectre-PHT.** Conditional branches can be executed speculatively

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

Fork execution into 2 speculative paths:
- `speculative = regular ∨ 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

\[
\text{store } a \to s \\
\text{store } a \to p \\
\text{store } b \to q \\
v = \text{load } a
\]

where \( a \neq b \)

1 regular path

\[
\text{store } a \to s \\
\text{store } a \to p \\
\text{store } b \to q \\
v = \text{load } a
\]
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`

where \( a \neq b \)

1 regular path
+ 3 extra transient paths

Explicit RelSE.

At load instructions: fork execution for each load/store interleaving.

\( \rightarrow \) Path explosion

(e.g. Pitchfork)
Explicit RelSE for Spectre-STL

**Spectre-STL.** Loads can speculatively bypass prior stores

\[
\begin{align*}
\text{store } a &\rightarrow s \\
\text{store } a &\rightarrow p \\
\text{store } b &\rightarrow q \\
v &\rightarrow \text{load } a \\
\text{store } a &\rightarrow s \\
\text{store } a &\rightarrow p \\
\text{store } b &\rightarrow q \\
v &\rightarrow \text{load } a \\
\end{align*}
\]

where \(a \neq b\)

1 regular path  
+ 3 extra transient paths

**Redundant case**  
Can be eliminated with read-over-write
Explicit RelSE for Spectre-STL

**Spectre-STL.** Loads can speculatively bypass prior stores

\[
\begin{align*}
\text{store} \ a \ s \\
\text{store} \ a \ p \\
\text{store} \ b \ q \\
v &= \text{load} \ a \\
\text{store} \ a \ s \\
\text{store} \ a \ p \\
v &= \text{load} \ a \\
\text{store} \ b \ q \\
\end{align*}
\]

where \( a \neq b \)

**Haunted RelSE.**
- Cut redundant cases
- Encode remaining ones in 1 path
  - symbolic \( \text{ite} \)
  - free booleans \( \beta_0, \beta_1 \)

\[
v \mapsto \text{ite} \ \beta_0 \ \text{then} \ \alpha \ \text{else} \ (\text{ite} \ \beta_1 \ \text{then} \ s \ \text{else} \ p)
\]

\[
\beta_0 = \text{false} \quad \beta_1 = \text{false}
\]
Experimental evaluation
Experimental evaluation

Binsec/Haunted.
Implementation of Haunted RelSE

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 ↗
Haunted vs. Explicit for Spectre-PHT

<table>
<thead>
<tr>
<th>Litmus tests (32 programs)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Paths</td>
</tr>
<tr>
<td>Explicit</td>
<td>1546</td>
</tr>
<tr>
<td>Haunted</td>
<td>370</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Libsodium &amp; OpenSSL (3 programs)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X86 Instr.</td>
</tr>
<tr>
<td>Explicit</td>
<td>2273</td>
</tr>
<tr>
<td>Haunted</td>
<td>8634</td>
</tr>
</tbody>
</table>

Tea and donna (10 programs). No difference between Explicit and Haunted ≈

Take away, Haunted RelSE vs Explicit RelSE.
• At worse: no overhead compared to Explicit ≈
• At best: faster, more coverage, less timeouts
### Haunted vs. Explicit for Spectre- STL

<table>
<thead>
<tr>
<th></th>
<th>Paths</th>
<th>X86 Ins.</th>
<th>Time</th>
<th>Timeouts</th>
<th>Bugs</th>
<th>Secure</th>
<th>Insecure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit</td>
<td>93M</td>
<td>2k</td>
<td>30h</td>
<td>15</td>
<td>22</td>
<td>3/4</td>
<td>13/23</td>
</tr>
<tr>
<td>Haunted</td>
<td>42</td>
<td>17k</td>
<td>24h</td>
<td>8</td>
<td>148</td>
<td>4/4</td>
<td>23/23</td>
</tr>
</tbody>
</table>

- Avoids paths explosion
- More unique instruction explored
- Faster

- Less timeouts
- More bugs found
- More programs proven secure / insecure

**Take away, Haunted RelSE vs Explicit RelSE.**

*Always wins!*
### Comparison Binsec/Haunted against Pitchfork & KLEESpectre (RQ3)

<table>
<thead>
<tr>
<th></th>
<th>Target</th>
<th>Programs</th>
<th>PHT</th>
<th>STL</th>
</tr>
</thead>
<tbody>
<tr>
<td>KLEESpectre</td>
<td>LLVM</td>
<td>Litmus tests, Tea &amp; donna</td>
<td>Explicit</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>☞ (≈240× slower)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>☀ (≈equivalent)</td>
<td></td>
</tr>
<tr>
<td>Pitchfork</td>
<td>Binary</td>
<td>Litmus tests, Tea &amp; donna</td>
<td>Optims</td>
<td>Explicit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>☀ (≈equivalent)</td>
<td>☞ 6/10 TO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>☞ (50× slower &amp; TO)</td>
<td>☞ TO</td>
</tr>
<tr>
<td>Binsec/Haunted</td>
<td>Binary</td>
<td>Litmus tests, Tea &amp; donna</td>
<td>Haunted</td>
<td>Haunted</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>☀</td>
<td>☀</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>☀</td>
<td>☀</td>
</tr>
</tbody>
</table>
Weakness of index-masking countermeasure
Weakness of Spectre-PHT countermeasure

**Index masking.** Add branchless bound checks

Program vulnerable to Spectre-PHT

```c
if (idx < size) { // size = 256
    v = tab[idx]
    leak(v)
}
```
Index masking. Add branchless bound checks

Index masking countermeasure

```c
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

```c
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

```c
register uint32_t ridx asm ("eax");
```
Wrap-up: detection of Spectre

- **Haunted RelSE** optimization
  - Model transient and regular 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

**Haunted**

https://github.com/binsec/haunted

- Haunted RelSE optimization for modelling speculative semantics
- **Binsec/Haunted**, binary-level tool to detect Spectre-PHT & STL
- New Spectre-**STL violations** with index masking and PIC