Vivienne: Relational Verification of Cryptographic Implementations in WebAssembly

Rodothea Myrsini Tsoupidi, Musard Balliu, and Benoit Baudry
KTH Royal Institute of Technology

#IEEESecDev
https://secdev.ieee.org/2021
WebAssembly

- **Low-level** language for the Web that runs on the **Web browser** or **standalone**.
- Designed for **portability**, improved **scalability**, and **security**.
- WebAssembly programs can be vulnerable to **timing attacks**.
Timing attacks

- Observe **differences in execution time** of programs to infer **secrets** (e.g. crypto keys)
- Timing attacks against **cryptographic implementations** of RSA, AES, and more

Example

```c
int tls1_cbc_remove_padding(const SSL *s,
SSL3_RECORD *rec, unsigned bs,
unsigned mac_size) {
    int ii, i, j;
    int l = rec->length;
    ii = i = rec->data[l-1]; /* padding_length */
    i++;
    ...
    for (j=(int)(l-i); j<(int)l; j++)
        if (rec->data[j] != ii) /* Incorrect padding */
            return -1;
    ...
}
```
Timing attacks

- Observe **differences in execution time** of programs to infer **secrets** (e.g. crypto keys)
- Timing attacks against **cryptographic implementations** of RSA, AES, and more

**Example**

```c
int tls1_cbc_remove_padding(const SSL *s,
    SSL3_RECORD *rec, unsigned bs,
    unsigned mac_size) {
    int ii, i, j;
    int l = rec->length;
    ii = i = rec->data[l-1]; /* padding_length */
    i++;
    ...
    for (j=(int)(l-i); j<(int)l; j++)
        if (rec->data[j] != ii) /* Incorrect padding */
            return -1;
    ...
}
```
Timing attacks

- Observe **differences in execution time** of programs to infer **secrets** (e.g. crypto keys)
- Timing attacks against **cryptographic implementations** of RSA, AES, and more

**Example**

```c
int tls1_cbc_remove_padding(const SSL *s,
  SSL3_RECORD *rec, unsigned bs,
  unsigned mac_size) {
  int ii, i, j;
  int l = rec->length;
  ii = i = rec->data[l-1]; /* padding_length */
  i++;
  ...
  for (j=(int)(l-i); j<(int)l; j++)
    if (rec->data[j] != ii) /* Incorrect padding */
      return -1;
  ...
}
```

**Branching** on secret values is insecure
Timing attacks

- Observe differences in execution time of programs to infer secrets (e.g. crypto keys)
- Timing attacks against cryptographic implementations of RSA, AES, and more

Example

```c
int tls1_cbc_remove_padding(const SSL *s,
SSL3_RECORD *rec, unsigned bs,
unsigned mac_size) {
int ii, i, j;
int l = rec->length;
ii = i = rec->data[l-1]; /* padding_length */
i++;
...
for (j=(int)(l-i); j<(int)l; j++)
    if (rec->data[j] != ii) /* Incorrect padding */
        return -1;
...
}
```

- Branching on secret values is insecure
Timing attacks

- Observe **differences in execution time** of programs to infer **secrets** (e.g. crypto keys)
- Timing attacks against **cryptographic implementations** of RSA, AES, and more

Example

```
int tls1_cbc_remove_padding(const SSL *s,
SSL3_RECORD *rec, unsigned bs,
unsigned mac_size) {
    int ii, i, j;
    int l = rec->length;
    ii = rec->data[l-1]; /* padding_length */
    i++;
    ...
    for (j=(int)(l-i); j<(int)l; j++)
        if (rec->data[j] != ii) /* Incorrect padding */
            return -1;
    ...
}
```

- **Branching** on secret values is insecure
- **Indexing memory** with secret values is insecure
Timing attacks

- Observe **differences in execution time** of programs to infer **secrets** (e.g. crypto keys)
- Timing attacks against **cryptographic implementations** of RSA, AES, and more

**Example**

```c
int tls1_cbc_remove_padding(const SSL *s,
   SSL3_RECORD *rec, unsigned bs,
   unsigned mac_size) {
  int ii, i, j;
  int l = rec->length;
  /* padding_length */
  ii = i = rec->data[l-1];
  i++;
  ...
  for (j=(int)(l-i); j<(int)l; j++)
    if (rec->data[j] != ii) /* Incorrect padding */
      return -1;
  ...
}
```

- **Branching** on secret values is insecure
- **Indexing memory** with secret values is insecure
Timing attacks

- Observe **differences in execution time** of programs to infer **secrets** (e.g. crypto keys)
- Timing attacks against **cryptographic implementations** of RSA, AES, and more

**Example**

```c
int tls1_cbc_remove_padding(const SSL *s,
   SSL3_RECORD *rec, unsigned bs,
   unsigned mac_size) {
  int ii, i, j;
  int l = rec->length;
  ii = i = rec->data[l-1]; /* padding_length */
  i++;
  ...
  for (j=(int)(l-i); j<(int)l; j++)
    if (rec->data[j] != ii) /* Incorrect padding */
      return -1;
  ...
}
```

- **Branching** on secret values is insecure
- **Indexing memory** with secret values is insecure
Constant time

Constant-time Programming

- A **programming discipline** to eliminate timing vulnerabilities.
- Given a **security policy** it prevents execution-time dependencies on **secret** values.
  - Convert **if** statements and **memory** operations to **constant-time equivalent** (e.g. using logical operations)
  - BearSSL, Libsodium, HACL*, and TweetNaCl implement **constant-time** cryptographic algorithms

Goal

Verify that **WebAssembly programs** respect the **constant-time policy** for cryptographic implementations
Constant-time verification

Constant-time Verification Approaches in WebAssembly

- **Ct-wasm**\(^1\) - Type-based approach
  - Considers the whole **memory** as **secret**
  - Requires type **annotation** of the program

Relational Symbolic Execution (RelSE)

- **RelSE for constant-time**\(^2,3\): Symbolic execution of two identical versions of the program with equal initial **public** values.
  - The **execution path** does not depend on **secret values**.
  - The **index of memory operations** does not depend on **secret values**.

---

---

Relational Symbolic Execution in WebAssembly

Vivienne: Relational Symbolic Execution in WebAssembly

Vivienne: Relational Verification of Cryptographic Implementations in WebAssembly

R. M. Tsoupidi, M. Balliu, B. Baudry (KTH)
Vivienne: Relational Symbolic Execution in WebAssembly

Vivienne - Relational Symbolic Execution (RelSE)

```
1    ...
2    local.get 6 ;; get l
3    local.get 1 ;; get (j - l)
4    i32.add
5    i32.load8_u ;; load data[j] from mem
6    ...
7    br_if 2 (;@1;) ;; break if j >= l
8    br 0 (;@3;) ;; continue to loop
```
Vivienne - Relational Symbolic Execution (RelSE)

1...2
3 local.get 6 ;; get l
4 local.get 1 ;; get (j - l)
5 i32.add
6 i32.load8_u ;; load data[j] from mem
7 ...
8 br_if 2 (;@1;) ;; break if j >= l
9 br 0 (;@3;) ;; continue to loop

\[ \phi \equiv \text{lv}_6 + \text{lv}_1 \neq \text{lv}_6 + \text{lv}'_1 \]

If \( \phi \) is sat then we have a vulnerability, if \( \phi \) is unsat, there is no vulnerability.

\( \mathcal{O}_\text{v} \) : if \( \text{v}_{\text{top}} \neq 0 \) then break else continue

\( \text{v}_{\text{top}} \) does not depend on secrets!

\( \text{lv}_6 = \text{lv}'_6 : \text{public} \)
Vivienne - Relational Symbolic Execution (RelSE)

1. ...
2. local.get 6 ;; get l
3. local.get 1 ;; get (j - l)
4. i32.add
5. i32.load8_u ;; load data[j] from mem
6. ...
7. br_if 2 (;@1;) ;; break if j >= l
8. br 0 (;@3;) ;; continue to loop

\[\text{vs} :: \langle lv_6 \rangle\]
\[\text{vs} :: \langle lv_6 \rangle :: \langle lv_1, lv'_1 \rangle\]
\[lv_6 = lv'_6 : \text{public}\]
\[lv_1 \neq lv'_1 : \text{secret}\]
Vivienne - Relational Symbolic Execution (RelSE)

```assembly
... 
local.get 6 ;; get l 
local.get 1 ;; get (j - l) 
'i32.add' ;; load data[j] from mem 
... 
br_if 2 (;@1;) ;; break if j >= l 
br 0 (;@3;) ;; continue to loop 
```

```
\[ \nu_1 \]
\[ \vdots \]
\[ \nu_6 \]
\[ \vdots \]

\[ \nu_6 = \nu_6' : \text{public} \]
\[ \nu_1 \neq \nu_1' : \text{secret} \]
```

---

We generate formula:

\[ \varphi \equiv \nu_6 + \nu_1 \neq \nu_6 + \nu_1' \]

If \( \varphi \) is sat then we have a vulnerability, if \( \varphi \) is unsat, there is no vulnerability.

\( \nu_6 = \nu_6' \) : public

\( \nu_1 \neq \nu_1' \) : secret
Vivienne - Relational Symbolic Execution (RelSE)

```
1  ...
2  local.get 6 ;; get l
3  local.get 1 ;; get (j - l)
4  i32.add
5  i32.load8_u ;; load data[j] from mem
6  ...
7  br_if 2 (;@1;) ;; break if j >= l
8  br 0 (;@3;) ;; continue to loop
```

\[
\text{vs} :: \langle \text{l}_6 \rangle \\
\text{vs} :: \langle \text{l}_6 \rangle :: \langle \text{l}_1, \text{l}_1' \rangle \\
\text{vs} :: \text{add}(\langle \text{l}_6 \rangle, \langle \text{l}_1, \text{l}_1' \rangle) \\
\text{vs} :: \text{mem}[\text{add}(\langle \text{l}_6 \rangle, \langle \text{l}_1, \text{l}_1' \rangle)]
\]

\(\text{l}_6 = \text{l}_6' : \text{public}\)  \\
\(\text{l}_1 \neq \text{l}_1' : \text{secret}\)
Vivienne: Relational Symbolic Execution in WebAssembly

Vivienne - Relational Symbolic Execution (RelSE)

```plaintext
1  ... 
2  local.get 6 ;; get l
3  local.get 1 ;; get (j - l)
4  i32.add
5  i32.load8_u ;; load data[j] from mem
6  ... 
7  br_if 2 (;@1;) ;; break if j >= l
8  br 0 (;@3;) ;; continue to loop
```

Need to prove that the index does not depend on secrets!

\[ \phi \equiv lv_6 + lv_1 \neq lv_6 + lv'_1 \]

If \( \phi \) is sat then we have a vulnerability, if \( \phi \) is unsat, there is no vulnerability.

\( \phi \) is sat if:

\[ \begin{align*}
\text{vs} &:: \langle lv_6 \rangle \\
\text{vs} &:: \langle lv_6 \rangle :: \langle lv_1, lv'_1 \rangle \\
\text{vs} &:: add(\langle lv_6 \rangle, \langle lv_1, lv'_1 \rangle) \\
\text{vs} &:: mem[add(\langle lv_6 \rangle, \langle lv_1, lv'_1 \rangle)]
\end{align*} \]

\( lv_6 = lv'_6 \) : public

\( lv_1 \neq lv'_1 \) : secret
Vivienne: Relational Symbolic Execution in WebAssembly

Vivienne - Relational Symbolic Execution (RelSE)

1 ... 
2 local.get 6 ;; get l 
3 local.get 1 ;; get (j - l) 
4 i32.add 
5 i32.load8_u ;; load data[j] from mem 
6 ... 
7 br_if 2 (;@1;) ;; break if j >= l 
8 br 0 (;@3;) ;; continue to loop 

Need to prove that the index does not depend on secrets!

vs :: ⟨lv6⟩ 
vs :: ⟨lv6⟩ :: ⟨lv1, lv1′⟩ 
vs :: add(⟨lv6⟩, ⟨lv1, lv1′⟩) 
vtop = mem[add(⟨lv6⟩, ⟨lv1, lv1′⟩)]

We generate formula:
\[ \phi \equiv lv6 + lv1 \neq lv6 + lv1′ \]

\( lv6 = lv6′ \) : public
\( lv1 \neq lv1′ \) : secret
Vivienne - Relational Symbolic Execution (RelSE)

```
1  ...  
2  local.get 6 ;; get l  
3  local.get 1 ;; get (j - l)  
4  i32.add  
5  i32.load8_u ;; load data[j] from mem  
6  ...  
7  br_if 2 (;@1;) ;; break if j >= l  
8  br 0 (;@3;) ;; continue to loop  
```

We generate formula:
\[ \phi \equiv lv_1 + lv_6 \neq lv_6 + lv_1 \]

If \( \phi \) is \text{sat} then we have a \text{vulnerability},
if \( \phi \) is \text{unsat}, there is \text{no vulnerability}.

Need to prove that the \text{index} does not depend on \text{secrets}!

\[ lv_6 = lv_6' : \text{public} \]
\[ lv_1 \neq lv_1' : \text{secret} \]
Vivienne - Relational Symbolic Execution (RelSE)

Need to prove that the index does not depend on secrets!

\[ \phi \equiv \text{lv}_6 + \text{lv}_1 \neq \text{lv}_6' + \text{lv}_1' \]

If \( \phi \) is sat then we have a vulnerability, if \( \phi \) is unsat, there is no vulnerability.

\( \text{lv}_6 = \text{lv}_6' : \text{public} \)
\( \text{lv}_1 \neq \text{lv}_1' : \text{secret} \)
Vivienne - Relational Symbolic Execution (RelSE)

```wasm
local . get 6 ;; get l
local . get 1 ;; get (j - l)
i32 . add
i32 . load8_u ;; load data [j] from mem

br_if 2 (;@1;) ;; break if j >= l
br 0 (;@3;) ;; continue to loop
```

Need to prove that the index does not depend on secrets!

We generate formula:
\[ \phi \equiv lv_6 + lv_1 \neq lv_6 + lv_1 \]

If \( \phi \) is sat then we have a vulnerability, if \( \phi \) is unsat, there is no vulnerability.

\[ \text{お話} \text{vs} :: v_{top}, \text{if} v_{top} \neq 0 \text{ then break else continue} \]

Need to prove that \( v_{top} \) does not depend on secrets!

R. M. Tsoupidi, M. Balliu, B. Baudry (KTH)
Vivienne - Relational Symbolic Execution (RelSE)

```assembly
1 ... 
2 local.get 6 ;; get l  
3 local.get 1 ;; get (j - l) 
4 i32.add 
5 i32.load8_u ;; load data[j] from mem 
6 ... 
7 br_if 2 (;@1;) ;; break if j >= l 
8 br 0 (;@3;) ;; continue to loop 
```

Need to prove that the index does not depend on secrets!

We generate formula: \( \phi \equiv lv_6 + lv_1 \neq lv_6 + lv'_1 \)

If \( \phi \) is sat then we have a vulnerability, if \( \phi \) is unsat, there is no vulnerability.

\( \text{Vcash} \) :: \( v_{top} \), if \( v_{top} \neq 0 \) then break else continue

Need to prove that \( v_{top} \) does not depend on secrets!
Vivienne - Relational Symbolic Execution (RelSE)

1. ...
2. local.get 6 ;; get l
3. local.get 1 ;; get (j - l)
4. i32.add
5. i32.load8_u ;; load data[j] from mem
6. ...
7. br_if 2 (;@1;) ;; break if j >= l
8. br 0 (;@3;) ;; continue to loop

Need to prove that the index does not depend on secrets!

We generate formula:

\[ \phi \equiv lv_6 + lv_1 \neq lv_6 + lv_1^\prime \]

If \( \phi \) is sat then we have a vulnerability, if \( \phi \) is unsat, there is no vulnerability.

\[ \text{vs} :: \text{v}_{\text{top}} \]
\[ \text{vs}, \text{if } v_{\text{top}} \neq 0 \text{ then } \text{break} \text{ else } \text{continue} \]
### Vivienne - Relational Symbolic Execution (RelSE)

#### Need to prove that the index does not depend on secrets!

\[
\phi \equiv \text{lv}_6 + \text{lv}_1 \neq \text{lv}_6 + \text{lv}'_1
\]

If \( \phi \) is sat then we have a vulnerability, if \( \phi \) is unsat, there is no vulnerability.

\[
\text{vs} :: \text{v}_\text{top}
\]

vs, if \( \text{v}_\text{top} \neq 0 \) then \text{break} else \text{continue}

---

1. ... 
2. local.get 6 ;; get l 
3. local.get 1 ;; get \((j - l)\) 
4. i32.add 
5. i32.load8_u ;; load data[\(j\)] from mem 
6. ... 
7. \textbf{br_if} 2 (;@1;) ;; break if \(j \geq l\) 
8. br 0 (;@3;) ;; continue to loop
Vivienne - Relational Inductive Invariant Generation

Loop Analysis

- Loop **unrolling** leads to **large overhead** for loops with **large bounds**.
  - **Bounded analysis** is unsound
  - **Automatic loop invariant** generation is a hard problem.

Relational Invariant Inference

1. Find all variables, $V$, that are **modified** in the loop.
2. Symbolically run **two passes** of the loop.
3. The variables that contain **public** values are added to the candidate invariant,
   \[ I \equiv \{ \forall x \in V_p \subseteq V, x = x' \} \]
4. **Havoc**(V), **Assume**(I), run **RelSE**, **Assert**(I)
Vivienne - Relational Inductive Invariant Inference

1. Find modified variables: \( V = \{l_{v1}, l_{v4}, \ldots\} \)

2. Symbolically run the loop body:
   1st: \( l_{v1} = pub; l_{v4} = 1 \)
   2nd: \( l_{v1} = pub + 1; l_{v4} = 1 \)

3. Generate candidate invariant:
   \( I \equiv \{l_{v1} = l'_{v1} \land l_{v4} = l'_{v4} = 1\} \)

4. Havoc(\( \{l_{v1}, l_{v4}, \ldots\} \)) = \( \{l_{v1} \leftarrow ns1, l'_{v1} \leftarrow ns1', \ldots\} \)

5. Assume(\( \{ns1 = ns1', ns4 = ns4' = 1\} \))

6. run RelSE

7. Assert(\( \{l_{v1} = l_{v1} \land l_{v4} = l'_{v4} = 1\} \))
Evaluation: Is RelSE practical for constant-time analysis in WebAssembly?

- Completes the analysis for 55/57 cases

<table>
<thead>
<tr>
<th>BS</th>
<th>A</th>
<th>✓</th>
<th>✗</th>
<th>#FS</th>
<th>#SS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTw</td>
<td>6</td>
<td>6/6</td>
<td>0/0</td>
<td>4K</td>
<td>0</td>
</tr>
<tr>
<td>Tw</td>
<td>3</td>
<td>3/3</td>
<td>0/0</td>
<td>181</td>
<td>0</td>
</tr>
<tr>
<td>WH</td>
<td>6</td>
<td>5/6</td>
<td>0/0</td>
<td>126K</td>
<td>0</td>
</tr>
<tr>
<td>B0</td>
<td>4</td>
<td>2/2</td>
<td>2/2</td>
<td>32K</td>
<td>40</td>
</tr>
<tr>
<td>B3</td>
<td>4</td>
<td>2/2</td>
<td>2/2</td>
<td>2K</td>
<td>40</td>
</tr>
<tr>
<td>L0</td>
<td>8</td>
<td>8/8</td>
<td>0/0</td>
<td>113K</td>
<td>18</td>
</tr>
<tr>
<td>L3</td>
<td>8</td>
<td>8/8</td>
<td>0/0</td>
<td>9K</td>
<td>18</td>
</tr>
<tr>
<td>A0</td>
<td>8</td>
<td>5/5</td>
<td>3/3</td>
<td>683</td>
<td>31</td>
</tr>
<tr>
<td>A3</td>
<td>8</td>
<td>5/5</td>
<td>3/3</td>
<td>55</td>
<td>9</td>
</tr>
<tr>
<td>Lu0</td>
<td>1</td>
<td>0/0</td>
<td>0/1</td>
<td>25K</td>
<td>4K</td>
</tr>
<tr>
<td>Lu3</td>
<td>1</td>
<td>0/0</td>
<td>1/1</td>
<td>3K</td>
<td>3K</td>
</tr>
<tr>
<td>Sum</td>
<td>57</td>
<td>44/45</td>
<td>11/12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Benchmarks Abbreviations

- **CTw**: Ct-wasm
- **Tw**: Wasm TweetNaCl
- **WH**: WHACL*
- **B0**: BearSSL with -O0
- **B3**: BearSSL with -O3
- **L0**: Libsodium with -O0
- **L3**: Libsodium with -O3
- **A0**: Almeida et al.\(^5\) with -O0
- **A3**: Almeida et al. with -O3
- **Lu0**: Lucky 13 with -O0
- **Lu3**: Lucky 13 with -O3

\(^5\) Almeida, José Bacelar, et al. "Verifying Constant-time Implementations"
Evaluation: How effective is the relational invariant for RelSE?

- Verifies all WH and identifies the Lu0 bugs

### Benchmarks Abbreviations

<table>
<thead>
<tr>
<th>BS</th>
<th>A</th>
<th>✓</th>
<th>✗</th>
<th>#FS</th>
<th>#SS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTw</td>
<td>6</td>
<td>6/6</td>
<td>0/0</td>
<td>814</td>
<td>412</td>
</tr>
<tr>
<td>Tw</td>
<td>3</td>
<td>3/3</td>
<td>0/0</td>
<td>320</td>
<td>164</td>
</tr>
<tr>
<td>WH</td>
<td>6</td>
<td>6/6</td>
<td>0/0</td>
<td>70K</td>
<td>7K</td>
</tr>
<tr>
<td>B0</td>
<td>4</td>
<td>1/2</td>
<td>0/2</td>
<td>10K</td>
<td>873</td>
</tr>
<tr>
<td>B3</td>
<td>4</td>
<td>0/2</td>
<td>1/2</td>
<td>158K</td>
<td>3K</td>
</tr>
<tr>
<td>L0</td>
<td>8</td>
<td>2/8</td>
<td>0/0</td>
<td>21K</td>
<td>347</td>
</tr>
<tr>
<td>L3</td>
<td>8</td>
<td>3/8</td>
<td>0/0</td>
<td>3K</td>
<td>309</td>
</tr>
<tr>
<td>A0</td>
<td>8</td>
<td>No loops</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lu0</td>
<td>1</td>
<td>0/0</td>
<td>1/1</td>
<td>539</td>
<td>217</td>
</tr>
<tr>
<td>Lu3</td>
<td>1</td>
<td>0/0</td>
<td>0/1</td>
<td>94</td>
<td>63</td>
</tr>
<tr>
<td>Sum</td>
<td>57</td>
<td>21/35</td>
<td>2/6</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

5 Almeida, José Bacelar, et al. "Verifying Constant-time Implementations"
Summary

- Design a RelSE approach to check constant-time implementations in WebAssembly.
- Propose an automatic relational invariant generation technique that analyzes more than half of the benchmarks including two implementations not terminating with unrolling.
- Analyze WHACL* and the compiled WebAssembly code of libsodium and BearSSL for the first time.

Check out Vivienne at: https://github.com/romits800/Vivienne
Contact: tsoupidi@kth.se

Thanks!