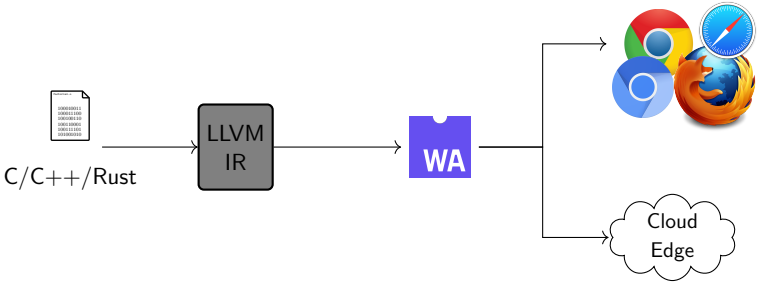
Vivienne: Relational Verification of Cryptographic Implementations in WebAssembly

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

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

```
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

```
1  int tls1_cbc_remove_padding(const SSL *s,
2              SSL3_RECORD *rec, unsigned bs,
3              unsigned mac_size) {
4   int ii, i, j;
5   int l = rec->length;
6   ii = i = rec->data[l-1]; /* padding_length */
7   i++;
8   ...
9   for (j=(int)(l-i); j<(int)l; j++)
10   if (rec->data[j] != ii) /* Incorrect padding */
11    return -1;
12  ...
13 }
```

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 = 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 = 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

```
1  int tls1_cbc_remove_padding(const SSL *s,
2              SSL3_RECORD *rec, unsigned bs,
3              unsigned mac_size) {
4   int ii, i, j;
5   int l = rec->length;
6   ii = i = rec->data[l-1]; /* padding_length */
7   i++;
8   ...
9   for (j=(int)(l-i); j<(int)l; j++)
10   if (rec->data[j] != ii) /* Incorrect padding */
11    return -1;
12  ...
13 }
```

- **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

```
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

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 = 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)

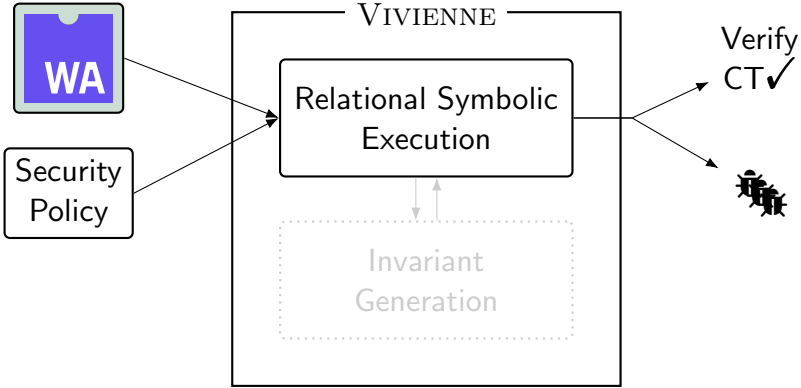
- **RelSE for constant-time**[2,3]: Symbolic execution of two identical versions of the program with equal initial **public** values.
  - The **execution path** does not depend on **secret values**.
  - The **index of memory operations** does not depend on **secret values**.

[1] Watt, Conrad, et al. "Ct-wasm: Type-driven Secure Cryptography for the Web Ecosystem." POPL (2019): 1-29.

[2] Farina, Gian Pietro et al. "Relational symbolic execution." PPDP (2019).

[3] Daniel, Lesly-Ann et al. "Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level." IEEE SP (2020).

Vivienne


WA
Security Policy
VIVIENNE
Relational Symbolic Execution
Invariant Generation
Verify CT✓

Vivienne - Relational Symbolic Execution (RelSE)

lv_1	🔑
…	…
lv_6	👁
…	…

```
...
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
```

Vivienne - Relational Symbolic Execution (RelSE)

lv_1	🔑
…	…
lv_6	👁
…	…

```
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
```

$vs :: \langle lv_6 \rangle$

$lv_6 = lv_6' :$ public

Vivienne - Relational Symbolic Execution (RelSE)

lv_1	🔑
…	…
lv_6	👁
…	…

```
...
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
```

$vs :: \langle lv_6 \rangle$ $\quad lv_6 = lv_6' : \text{public}$

$vs :: \langle lv_6 \rangle :: \langle lv_1, lv_1' \rangle$ $\quad lv_1 \neq lv_1' : \text{secret}$

Vivienne - Relational Symbolic Execution (RelSE)

lv_1	🔑
...	...
lv_6	👁
...	...

```
...
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
```

$vs :: \langle lv_6 \rangle$

$vs :: \langle lv_6 \rangle :: \langle lv_1, lv_1' \rangle$

$vs :: add(\langle lv_6 \rangle, \langle lv_1, lv_1' \rangle)$

$lv_6 = lv_6'$: public

$lv_1 \neq lv_1'$: secret

Vivienne - Relational Symbolic Execution (RelSE)

lv_1	🔑
...	...
lv_6	👁
...	...

```
...
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
```

$vs :: \langle lv_6 \rangle$ $\quad lv_6 = lv_6' :$ public

$vs :: \langle lv_6 \rangle :: \langle lv_1, lv_1' \rangle$ $\quad lv_1 \neq lv_1' :$ secret

$vs :: add(\langle lv_6 \rangle, \langle lv_1, lv_1' \rangle)$

$vs :: mem[add(\langle lv_6 \rangle, \langle lv_1, lv_1' \rangle)]$

Vivienne - Relational Symbolic Execution (RelSE)

```
...
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
```

lv_1	🔑
…	…
lv_6	👁
…	…

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

$vs :: \langle lv_6 \rangle$

$vs :: \langle lv_6 \rangle :: \langle lv_1, lv_1' \rangle$

$vs :: add(\langle lv_6 \rangle, \langle lv_1, lv_1' \rangle)$

$vs :: mem[add(\langle lv_6 \rangle, \langle lv_1, lv_1' \rangle)]$

$lv_6 = lv_6'$: public

$lv_1 \neq lv_1'$: secret

Vivienne - Relational Symbolic Execution (RelSE)

```
...
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
```

lv_1	🔑
…	…
lv_6	👁
…	…

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

$vs :: \langle lv_6 \rangle$
$vs :: \langle lv_6 \rangle :: \langle lv_1, lv_1' \rangle$
$vs :: add(\langle lv_6 \rangle, \langle lv_1, lv_1' \rangle)$
$vs :: mem[add(\langle lv_6 \rangle, \langle lv_1, lv_1' \rangle)]$

$lv_6 = lv_6'$: public
$lv_1 \neq lv_1'$: secret

We generate formula:
$\phi \equiv lv_6 + lv_1 \neq lv_6 + lv_1'$

Vivienne - Relational Symbolic Execution (RelSE)

lv_1	🔑
…	…
lv_6	👁
…	…

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

```
...
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
```

$vs :: \langle lv_6 \rangle$
$vs :: \langle lv_6 \rangle :: \langle lv_1, lv_1' \rangle$
$vs :: add(\langle lv_6 \rangle, \langle lv_1, lv_1' \rangle)$
$vs :: mem[add(\langle lv_6 \rangle, \langle lv_1, lv_1' \rangle)]$

$lv_6 = lv_6'$: public
$lv_1 \neq lv_1'$: secret

We generate formula:
$\phi \equiv lv_6 + lv_1 \neq lv_6 + lv_1'$

If ϕ is **sat** then we have a **vulnerability**, if ϕ is **unsat**, there is **no vulnerability**

Vivienne - Relational Symbolic Execution (RelSE)

lv_1	🔑
…	…
lv_6	👁
…	…

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

```
...
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
```

$vs :: \langle lv_6 \rangle$
$vs :: \langle lv_6 \rangle :: \langle lv_1, lv_1' \rangle$
$vs :: add(\langle lv_6 \rangle, \langle lv_1, lv_1' \rangle)$
$vs :: mem[add(\langle lv_6 \rangle, \langle lv_1, lv_1' \rangle)]$

$lv_6 = lv_6'$: public
$lv_1 \neq lv_1'$: secret

We generate formula:
$\phi \equiv lv_6 + lv_1 \neq lv_6 + lv_1'$

If ϕ is **sat** then we have a **vulnerability**, if ϕ is **unsat**, there is **no vulnerability**

Vivienne - Relational Symbolic Execution (RelSE)

lv_1	🔑
…	…
lv_6	👁
…	…

```
...
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
```

Vivienne - Relational Symbolic Execution (RelSE)

lv_1	🔑
...	...
lv_6	👁
...	...

```
...
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
```

$vs :: v_{top}$

Vivienne - Relational Symbolic Execution (RelSE)

lv_1	🔑
...	...
lv_6	👁
...	...

```
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
```

$vs :: v_{top}$

vs, if $v_{top} \neq 0$ then *break* else *continue*

Vivienne - Relational Symbolic Execution (RelSE)

lv_1	🔑
...	...
lv_6	👁
...	...

```
...
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 v_{top} does not depend on **secrets**!

$vs :: v_{top}$

vs, if $v_{top} \neq 0$ then *break* else *continue*

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

```
loop  ;; label = @3
  ...
  i32.const 1
  local.set 4 ;; store return value
  ...
  local.set 1 ;; lv1 + 1
  ...
  br_if 2 (;@1;) ;; break if lv1 < N
  br 0 (;@3;) ;; continue to loop
end
```

1. Find modified variables: $V = \{lv_1, lv_4, ...\}$
2. Symbolically run the loop body:
   1st: $lv_1 = pub; lv_4 = 1$
   2nd: $lv_1 = pub + 1; lv_4 = 1$
3. Generate candidate invariant:
   $I \equiv \{lv_1 = lv_1' \ \wedge \ lv_4 = lv_4' = 1\}$
4. $\texttt{Havoc}(\{lv_1, lv_4, ...\}) = \{lv_1 \leftarrow \texttt{ns1}, lv_1' \leftarrow \texttt{ns1}', ...\}$
5. $\texttt{Assume}(\{\texttt{ns1} = \texttt{ns1}', \texttt{ns4} = \texttt{ns4}' = 1\})$
6. run RelSE
7. $\texttt{Assert}(\{lv_1 = lv_1 \ \wedge \ lv_4 = lv_4' = 1\})$

Evaluation: Is RelSE practical for constant-time analysis in WebAssembly?

- Completes the analysis for 55/57 cases

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

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

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

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"

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!