Vivienne: Relational Verification of Cryptographic Implementations in WebAssembly

Rodothea Myrsini Tsoupidi
KTH Royal Institute of Technology
Stockholm, Sweden
tsoupidi@kth.se

Musard Balliu
KTH Royal Institute of Technology
Stockholm, Sweden
musard@kth.se

Benoit Baudry
KTH Royal Institute of Technology
Stockholm, Sweden
baudry@kth.se

***Abstract*—We investigate the use of relational symbolic execution to counter timing side channels in WebAssembly programs. We design and implement VIVIENNE, an open-source tool to automatically analyze WebAssembly cryptographic libraries for constant-time violations. Our approach features various optimizations that leverage the structure of WebAssembly and automated theorem provers, including support for loops via relational invariants. We evaluate Vivienne on 57 real-world cryptographic implementations, including a previously unverified implementation of the HACL* library in WebAssembly. The results indicate that Vivienne is a practical solution for constant-time analysis of cryptographic libraries in WebAssembly.**

I. INTRODUCTION

The introduction of WebAssembly [1], a portable low-level language with focus on security and efficiency, has led to an array of security-sensitive applications. Cryptography libraries such as libsodium [2] and HACL* [3] are a prime example of such applications. Unfortunately, WebAssembly programs can be vulnerable to different types of attacks [4], including timing side channels.

The constant-time programming discipline is a well-known practice to defend against timing attacks [5], [6]. The main idea is to disallow the program's control flow and the memory access patterns that depend on program secrets. This is surprisingly challenging because many cryptographic routines are human-written [2], [7], [8] and thus, prone to errors, while compilers that preserve constant time are yet to emerge [2], [7]. This motivates the need for verification of constant-time implementations in WebAssembly.

Drawing on the verification-friendly structure of WebAssembly, existing solutions such as CT-wasm [9] enrich the WebAssembly type system with security annotations to enforce constant time. The efficiency of CT-wasm comes at the expense of a conservative analysis, e.g., by considering the whole memory as secret, thus leading to false positives or refactoring of constant-time programs. This paper explores the use of Relational Symbolic Execution (RelSE) to verify constant-time implementations in WebAssembly. The approach relies on an accurate modelling of the memory and other program optimizations, enabling a precise analysis that scales to real-world cryptographic implementations. In summary, this paper offers the following contributions:

- An RelSE-based approach for verifying constant-time implementations in WebAssembly programs.
- An automated invariant generation technique for analyzing implementations with loops.
- A thorough evaluation on 45 secure implementations and 12 insecure implementations in WebAssembly, including the previously non-verified WebAssembly implementation of HACL* (WHACL*).
- VIVIENNE, an open-source implementation of the approach.

II. PROBLEM SETTING

This section presents the problem setting, including the constant-time policy, and background on WebAssembly and related works.

A. Constant-time Policy

Constant-time programming discipline is a software-based defense against timing side-channel attacks. This discipline relies on the constant-time policy [10], which classifies values as secret (`high`) and public (`low`). The policy constrains the control-flow instructions and the memory operations to solely depend on public values, thus disallowing any secret-dependent control-flow instructions and memory accesses. Intuitively, the policy requires that any program executions with the same `low` values execute the same instructions and yield the same memory access patterns, independently of `high` values. This indicates that execution time of the program is not affected by secret data.

Listing 1
C FUNCTION `TLS1_CBC_REMOVE_PADDING`

```
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 }
```

Listing 1 reports a code snippet of the OpenSSL's Lucky 13 timing vulnerability [11] to illustrate the issue. Function `tls1_cbc_remove_padding` removes the padding from

a decrypted message that contains the plain text (secret), the Message Authentication Code (MAC) tag, and the padding. The size of the padding affects the execution time, which in turn reveals information about the size of the plain text. Specifically, `rec->data` holds the decrypted message together with the MAC tag and the padding, and is thus secret. Variables `i` and `ii` (line 6) contain the last item of array `rec->data`, which holds the padding size. Hence, the number of iterations of the `for` loop at line 9 depends on the secret-dependent variable `i`, which affects the execution time of the function. Similarly, the guard of `if` statement at line 10 depends on `ii`, which is also secret. Memory accesses also reveal information through timing due to the presence of caches. At line 10, the access to `rec->data[j]` reveals information about the value of index `j` by timing its presence in the cache.

B. WebAssembly

WebAssembly [1] is a stack-based typed low-level language serving as backend for both client-side computations, e.g., web browsers, and server-side computations [4] including standalone applications [12]. With some exceptions [8], WebAssembly code is compiler generated, e.g., via LLVM with support for C, C++, and Rust. Other languages, like Python and Julia, also provide support for WebAssembly. WASI Libc [12] is a library built on top of WASI system calls to enable I/O and memory management for WebAssembly programs.

The execution model of WebAssembly [1] consists of 1) an execution stack es that stores the instructions; 2) a value stack vs that holds the input arguments of the instructions, 3) a linear memory, and 4) the local and the global stores. WebAssembly has a structured control flow; for indirect calls (`call_indirect`), the call destination is an index to a function table; for conditional branch (`br_if`), the branch destination is an index i to enter (`loop`) or exit (`block`) the ith scope. Memory operations read from (`load`) and write to (`store`) the linear memory, and global variables are visible to all functions in a module. A function may also define local variables `lv`n including the function parameters. Modules are collections of functions with their own linear memory, and global variables [1].

Listing 2 shows an example WebAssembly module. The code is a simplified compiled version (using clang-10) of the C code in Listing 1. The code consists of a module (line 1-33), which imports a memory instance (`"_memory"`) from another module `$env` (line 3) and declares function `tls1_cbc_remove_padding` (line 4). The function takes four input parameters of type 32-bit integer and returns a 32-bit value (line 5). At line 6, the function declares five local variables and the rest of the function consists of the function body. The block at line 8 performs multiple initializations before the beginning of the loop (line 15). At line 10, instruction `local.tee` stores the top value of vs (here `rec->data + 1`) to `lv6` and pushes the same value back to vs. At line 15, the loop starts by loading `lv6` and `lv1` to vs. Instruction `i32.add` adds these two values and pushes back the result to vs. Finally, instruction `i32.load8_u` loads from the linear memory (`"_memory"`) the value at the index taken from the top of vs, i.e. the result of the addition. The loop body executes until instruction `br_if`, which reads one value from vs; if the value is non zero (`true`), the execution breaks out of the outermost block (lines 8-31), whereas if the value is zero (`false`), the execution continues to the next instruction, `br`, which unconditionally jumps back to the beginning of the loop (line 15).

Listing 2
WASM FUNCTION `TLS1_CBC_REMOVE_PADDING`

```
1  (module
2    ...
3    (import "env" "_memory" (memory (;0;) 2))
4    (func $tls1_cbc_remove_padding (type 2)
5          (param i32 i32 i32 i32) (result i32)
6      (local i32 i32 i32 i32 i32)
7      ...
8      block  ;; label = @1
9        ...
10       local.tee 6 ;; tee (rec->data + 1)
11       ...
12       local.tee 1 ;; tee (1 - ii)
13       ...
14       block  ;; label = @2
15         loop  ;; label = @3
16           local.get 6 ;; get 1
17           local.get 1 ;; get (j - 1)
18           i32.add
19           i32.load8_u ;; load data[j] from memory
20           ...
21           i32.const 1
22           local.set 4 ;; store return value
23           ...
24           local.set 1 ;; j++
25           ...
26           br_if 2 (;@1;) ;; break if j >= l
27           br 0 (;@3;) ;; continue to loop
28         end
29       end
30       ...
31     end
32     ...))
```

WebAssembly programs may be vulnerable to timing side-channel attacks. The constant-time policy for WebAssembly concerns control-flow instructions, i.e. `br_if`, `if`, `br_table`, and `call_indirect`, and the memory operations, i.e. `load` and `store`.

C. Related Work

Several works have aimed at improving the security of WebAssembly [4], [9], [13], [14], [15]. CT-wasm [9] proposes a type system to check the constant-time policy. Type checking is very efficient but it suffers from the annotation burden and the conservative nature of the analysis. In CT-wasm, this is reflected by the treatment of the whole memory as secret, e.g. requiring that every `load` operation returns a `high` value, which may require refactoring of the programs to make them amenable to the analysis (e.g., `poly1305_blocks` and `poly1305_update` functions of a WebAssembly TweetNaCl implementation [8]). Our approach aims at overcoming these limitations by means of RelSE, using a more accurate memory model and no extensive annotation burden. Moreover, we expect our analysis to yield less false positives because

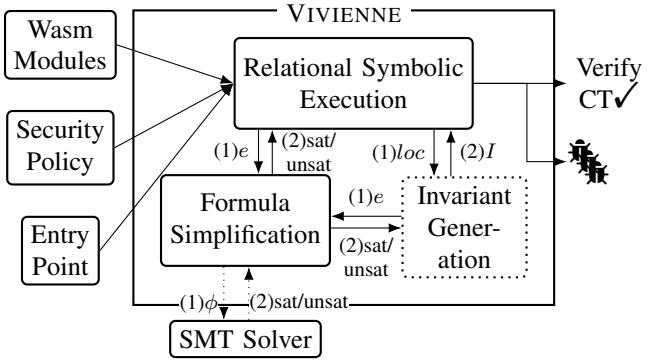

Fig. 1. VIVIENNE Architecture

$$
\begin{array}{lll}
v\ (values) & ::= h_n \mid l_n \mid c, & c \in \mathbb{Z}, n \in \mathbb{N}_0 \\
\rho\ (relational\ values) & ::= \langle v, v \rangle & \\
e\ (expressions) & ::= \rho \mid \texttt{Add}(e,e) \mid \texttt{Sub}(e,e) \mid ... & \\
 & \quad \mid \texttt{Le}(e,e) \mid \texttt{Load}(e,\mu) & \\
i\ (instructions) & ::= \texttt{br\_if}\ l \mid ... \mid \texttt{load}, & l \in \mathbb{N}_0 \\
\hline
\mu\ (memory) & ::= \bot \mid \texttt{Store}(e,e,\mu) & \\
st\ (stack) & ::= \varnothing \mid e :: st & \\
pc\ (path\ condition) & ::= \texttt{true} \mid e \wedge pc & \\
es\ (execution\ stack) & ::= \varnothing \mid i :: es & \\
lv\ (local\ variables) & ::= \{lv_0 \mapsto e, ..., lv_n \mapsto e\} &
\end{array}
$$

Fig. 2. Symbolic Data Structures

it relies on symbolic execution which is more precise than security type systems. For example, an expression such as `secret − secret` would be correctly identified as the constant `0`. However, as we will see, our solution comes with a computation cost due to the increased precision.

Almeida et al. [10] use product programs to verify constant-time for C implementations. A drawback of verifying the constant-time policy for high-level languages is that the analysis does not provide guarantees on the security of the generated code (see `ct_select` implementations [16]). Daniel et al. [16] verify constant-time programs at the binary level using RelSE. Web browsers using WebAssembly typically leverage Just-in-time (JIT) compilation, which does not result in binary file generation. Moreover, the verification of constant-time at the WebAssembly level provides opportunities for optimization due to WebAssembly's structured design. HACL* [17] uses a high-level specification language to generate a formally verified cryptographic library that is available in different languages including C and WebAssembly [3].

III. VIVIENNE: RELSE FOR WEBASSEMBLY

VIVIENNE analyzes WebAssembly implementations with respect to constant time. Figure 1 shows a high-level view of the tool. VIVIENNE takes three inputs: 1) the *WebAssembly modules* containing the functions to analyze, 2) the *security policy* annotating the memory regions and the parameters of the entry function, and 3) the *entry point* describing the entry function to analyze. Then, VIVIENNE performs RelSE on the entry function, reporting the discovered constant-time vulnerabilities (if any). We describe the different components of VIVIENNE using Listing 2 as a running example.

WebAssembly Modules The modules include the entry function to verify and its dependencies, possibly involving different modules. For example, the module in Listing 2 imports the memory from another module `$env` (line 3) and defines function `tls1_cbc_remove_padding` (lines 4-28).

Security Policy and Entry Point The security policy specifies the parts of the memory and the arguments of the entry function that contain public or secret values. Listing 3 reports the policy for function `tls1_cbc_remove_padding`. The policy specifies the bytes 2000 to 2039 (i.e. pointer `s`) and the memory of struct `rec` as public (not shown), and the bytes 2048 to 2111 (i.e. `rec->data`) as secret, thus reflecting the specification in Listing 1. Moreover, VIVIENNE requires the code of the modules (line 8) and the *entry function* (lines 9-11). The latter includes the security policy for its arguments which can be either concrete or symbolic values. Lines 9–11 specify the concrete and symbolic arguments for analyzing function `tls1_cbc_remove_padding` via RelSE. The function takes four arguments: 1) the memory index of `s`; 2) the memory index of struct `rec`; 3) the block size, which is a public symbolic value; and 4) the `MAC` size which is also a public symbolic value. VIVIENNE recognizes public (secret) symbolic values that start with letter `l` (`h`).

Listing 3
SECURITY POLICY AND ENTRY FUNCTION

```
1  (module $env
2   (memory (;0;) $memory (export "_memory") 2)
3   (public (i32.const 2000) (i32.const 2039));;s
4   ...
5   (secret (i32.const 2048) (i32.const 2111));;data
6  )
7  ;;definition of tls1_cbc_remove_padding-Listing 2
8  ...
9  (symb_exec "tls1_cbc_remove_padding"
10   (i32.sconst 2000) (i32.sconst 2040) ;; concrete
11   (i32.sconst l1)   (i32.sconst l2))  ;; symbolic
```

Relational Symbolic Execution VIVIENNE uses the above-mentioned inputs to initiate RelSE [18] for the entry function. RelSE performs symbolic execution on relational states representing two program executions with identical public values but different secret values. We now describe the ingredients underpinning the constant-time analysis with VIVIENNE.

a) Symbolic State: A symbolic state σ consists of 1) the execution stack es, that contains the WebAssembly instructions, 2) the symbolic stack st, 3) the symbolic memory μ, 4) the symbolic local (and global) variables lv, and 5) the path condition pc. Figure 2 summarizes these five components of a symbolic state $\sigma = \langle es, st, \mu, lv, pc \rangle$. By convention, the values starting with h (l) are secret (public). Our symbolic analysis operates on pairs of symbolic values ρ. We write $\rho_{|l}$ ($\rho_{|r}$) to denote the first (second) element of a pair ρ. For public values, we have that $\rho_{|l} = \rho_{|r}$ and write $\langle v \rangle$, while for

secret values $\rho_{|l}$ and $\rho_{|r}$ may differ. We lift this notation to expressions and the memory as expected.

b) Execution Path Exploration: We use small-step symbolic evaluation to analyze the instructions. At every step, the analysis takes a symbolic state as input and returns a list of symbolic states that correspond to the feasible execution paths. We visit the instructions in a depth-first search fashion and collect all path conditions pc to check path feasibility using an Satisfiability Modulo Theories (SMT) solver.

c) Symbolic Stack: The symbolic stack holds symbolic expressions e resulting from stack operations on symbolic values. Consider the `get` instructions at lines 16–17 in Listing 2 with the current symbolic memory μ and empty symbolic stack st. The program loads the symbolic expressions of `lv6` i.e. $\langle 2112 \rangle$, and `lv1` i.e. $\texttt{Sub}(\langle 1 \rangle, \texttt{Load}(\langle 2111 \rangle, \mu))$ to the stack st. At line 18, the analysis of instruction `add` pops the two symbolic expressions off the stack st and pushes back the result, $\texttt{Add}(\langle 2112 \rangle, \texttt{Sub}(\langle 1 \rangle, \texttt{Load}(\langle 2111 \rangle, \mu)))$.

d) Memory Operations: When analyzing a memory operation at index e, as in $\langle \texttt{load} :: es, e :: st, \mu, lv, pc \rangle$ or $\langle \texttt{store} :: es, e_1 :: e :: st, \mu, lv, pc \rangle$, the analysis generates a formula, $\phi = (T(e)|_{|r} \neq T(e)_{|l})$ to check that the index is not secret-dependent. The function $T : e \rightarrow \langle Exp, Exp \rangle$ translates the index expression e to a pair of SMT expressions Exp. If e only depends on public values, then for all valuations of e, $e_{|r} = e_{|l}$, thus ϕ is *unsatisfiable* and the memory operation is `safe`. However, if ϕ is *satisfiable*, then there are concrete values, such that the memory addresses for the two executions, $e_{|r}$ and $e_{|l}$, are different. This is only possible if expression e depends on secret values, and, thus, the solution to ϕ reveals a violation of constant time. In our example in Listing 2, load operation `load8_u` at line 19 has as index the top value of st, $\texttt{Add}(\langle 2112 \rangle, \texttt{Sub}(\langle 1 \rangle, \texttt{Load}(\langle 2111 \rangle, \mu)))$. The policy in Listing 3 specifies $\texttt{Load}(\langle 2111 \rangle, \mu)$ as secret, i.e. $\texttt{Load}(\langle 2111 \rangle, \mu) = \langle h_1, h_1' \rangle$ with $h_1 \neq h_1'$. Thus, the generated formula $\phi = (2112 + (1 - h_1)) \neq (2112 + (1 - h_1'))$ is satisfiable for different values of h_1 and h_1'. This means that there exist the two concrete executions that differ with regards to the memory index, which violates constant-time.

e) Control-flow Instructions: Like memory operations, control-flow instructions require checking that boolean expression e, as in $\langle \texttt{br\_if}\ 0 :: es, e :: st, \mu, lv, pc \rangle$, is not secret-dependent. Our analysis generates a formula to check whether the two paths of the relational state take different branches. WebAssembly considers value *zero* as false and *any non-zero* value as true, hence the generated formula is $\phi = (T(e)_{|r} = 0) \wedge (T(e)_{|l} \neq 0)$. Formula ϕ is satisfiable only if there is a valuation of e such that the two executions follow different execution paths, indicating a violation of the constant-time policy.

Formula Simplification When RelSE needs to check the constant-time policy for an expression e, it first passes e to the simplification step (SS). SS translates the expression to a pair of SMT expressions, $e' = T(e)$, using the theory of bitvectors and arrays (32-bit indexed byte array), `QF_ABV`. The transformation includes simplification and memoization steps to reduce the recalculation overhead. Finally, based on the type of the query, namely memory operation or control-flow statement, this step generates formula ϕ. For our previous example, SS first translates expression $e = \texttt{Add}(\langle 2112 \rangle, \texttt{Sub}(\langle 1 \rangle, \texttt{Load}(\langle 2111 \rangle, \mu)))$ to two SMT expressions $2112 + (1 - h_1)$ and $2112 + (1 - h_1')$, which are then simplified to $2113 - h_1$ and $2113 - h_1'$, hence the final formula becomes $\phi = (2113 - h_1) \neq (2113 - h_1')$. To solve the simplified formula, VIVIENNE invokes an SMT solver. For simple formulas, however, the resulting ϕ may already be a concrete boolean, e.g., `false`, allowing VIVIENNE skip a call to the SMT solver.

SMT Solver After the simplification step, VIVIENNE invokes an SMT solver for solving the simplified formula, ϕ. The SMT solver of VIVIENNE has two modes, one for small formulas and one for large and complex formulas. For small formulas, VIVIENNE uses a solver that provides bindings to the implementation language of VIVIENNE and thus, has a reduced communication cost. However, for larger formulas, the communication overhead is less significant compared to the benefit of using a more powerful SMT solver. In particular, for larger queries VIVIENNE uses a portfolio solver were many solvers take as input the same formula and the solver that finishes first returns the result. To decide over which solver mode to use, VIVIENNE uses the *number of expressions* in the formula.

Invariant Generation VIVIENNE has an optional invariant generation step for analyzing loops. When invariant generation is enabled and the analysis visits a loop at location loc, VIVIENNE starts a preprocessing step to automatically generate a relational invariant I. The invariant defines the variables (local variables, global variables, and memory) that are public, i.e. $I = \{\forall x \in V_p \subseteq V.\ x_{|l} = x_{|r}\}$, where V is the set of all variables modified in the loop and V_p is the subset of the modified variables that are public. To discover whether a variable is public or secret, the preprocessing step queries the SMT solver about the security policies of the modified variables, V, after symbolically executing one loop iteration. That is, given a variable $x \in V$, the preprocessing step generates a query, $\phi = (x_{|l} \neq x_{|r})$. If the query is unsatisfiable, then the variable is assumed to be public and x is added to V_p, otherwise, it is assumed to be secret. In the special case of $x_{|l} = x_{|r} = c \in \mathbb{Z}$, the analysis assumes that x has a symbolic value c and adds the equality constraint $x = c$ to the invariant, I. After generating invariant I, the analysis continues with verifying this invariant. To do that, VIVIENNE 1) generates fresh symbolic variables (havoc) for all modified variables $x \in V$, 2) assumes that the invariant, I, holds, 3) performs RelSE on the loop body with the havoced values and discovers possible vulnerabilities, 4) verifies that the invariant holds by asserting I on the new relational state. If the generated invariant is not a loop invariant, then the last step will fail. After analyzing the loop body, the analysis continues outside the loop. The invariant verification algorithm is a generalization of standard (functional) invariant checking, hence we expect the loop analysis to be sound, as supported

by the experiments.

Consider the loop at $loc = 15$ in Listing 2. Local variables 1 and 4 are modified in the loop body, i.e. $V = \{lv1, lv4\}$. Of these, $lv1$ stores `j` (line 24), which is secret because it depends on `rec->data[l-1]` and $lv4$ stores value 1, which is public. Thus, $V_p = \{lv4\}$, hence the invariant is $I = \{lv4_{|l} = lv4_{|r}\}$. To analyze the loop, VIVIENNE 1) havocs $lv1$ and $lv4$, 2) assumes the invariant I, i.e. that $lv4$ is initially public, 3) performs RelSE at the loop body to discover constant-time vulnerabilities, and 4) asserts the invariant I. Here, the program assigns $lv4$ only once in the loop body, at line 22, where, $lv4$ takes value one, which is public, and thus, the invariant I holds.

Output VIVIENNE outputs the discovered constant-time violations (🐞), if any, as well as the SMT solver-generated counterexamples that witness these violations.

VIVIENNE is implemented as an extension of the WebAssembly reference interpreter [19] in OCaml, using OCaml compiler 4.06. VIVIENNE uses the OCaml interface of z3 [20] to generate and simplify the constant-time formulas, and solve queries that have a small number of expressions. For larger formulas, VIVIENNE uses a portfolio solver consisting of four solvers, i.e. Boolector [21], Yices2 [22], CVC4 [23], and Z3 [20] running in parallel. VIVIENNE is publicly available online at https://github.com/romits800/Vivienne.

IV. EVALUATION

We evaluate VIVIENNE with respect to three research questions:

RQ1: Can we use RelSE for constant-time analysis of real-world cryptographic implementations in WebAssembly? To investigate the effectiveness and efficiency of RelSE for constant-time analysis on WebAssembly programs, we use VIVIENNE to analyze the implementations of seven cryptographic libraries within a time limit of 90 minutes.

RQ2. To what extent do the automatically generated loop invariants affect the scalability and precision of RelSE? We evaluate VIVIENNE's support for automatic invariant generation on our benchmarks and compare it to the results of RQ1.

RQ3. How does VIVIENNE compare to existing approaches for constant-time analysis of WebAssembly? We compare VIVIENNE with CT-wasm [9] with regards to simplicity, permissiveness, and efficiency.

A. Experimental Setup and Overview of Benchmarks

We run the experiments on a machine running Debian GNU/Linux 10 (buster) on an IntelCore™i9-9920X processor 3.50GHz with 64GB of RAM. We used the LLVM-10 compiler with WASI libc [12] and two optimization levels (`-O0` and `-O3`) for compiling our C benchmarks to WebAssembly. VIVIENNE uses a time limit of 90 minutes for each benchmark and a threshold of 1500 expressions to trigger a call to the portfolio solver.

We evaluate VIVIENNE with seven cryptography libraries, including both constant-time and non-constant-time implementations. Some benchmarks have been used in prior works [9], [16] to evaluate constant-time policies, which provides us with common ground for comparison. We extract the security policies for the first two libraries from the type annotations of CT-wasm [24] and use the policies of Binsec/Rel [25] for the other libraries. The full details of our benchmarks are available at https://github.com/romits800/Vivienne_eval.

Bench.		VIVIENNE$_{unroll}$				VIVIENNE$_{inv}$			
BS	A	✓	✗	#FS	#SS	✓	✗	#FS	#SS
CTw	6	6/6	0/0	4K	0	6/6	0/0	814	412
Tw	3	3/3	0/0	181	0	3/3	0/0	320	164
WH	6	5/6	0/0	126K	0	6/6	0/0	70K	7K
B0	4	2/2	2/2	32K	40	1/2	0/2	10K	873
B3	4	2/2	2/2	2K	40	0/2	1/2	158K	3K
L0	8	8/8	0/0	113K	18	2/8	0/0	21K	347
L3	8	8/8	0/0	9K	18	3/8	0/0	3K	309
A0	8	5/5	3/3	683	31	No loops			
A3	8	5/5	3/3	55	9				
Lu0	1	0/0	0/1	25K	4K	0/0	1/1	539	217
Lu3	1	0/0	1/1	3K	3K	0/0	0/1	94	63
Sum	57	44/45	11/12	-	-	21/35	2/6	-	-

TABLE I
VERIFYING 57 CRYPTOGRAPHY FUNCTIONS WITH VIVIENNE, WITH UNROLLING AND WITH INVARIANT INFERENCE. THE NUMBERS IN RED DENOTE INCOMPLETE RESULTS.

CT-wasm benchmarks (CTw): Three handwritten WebAssembly benchmarks from CT-wasm [9]. We verify the `encrypt` and `decrypt` functions of `Salsa20` and `TEA`, and the `transform` and `update` functions of `SHA256`.

TweetNaCl WebAssembly (Tw): WebAssembly implementation of TweetNaCl [8] previously verified by CT-wasm [9]. We verify `core_hsalsa20`, `core_salsa20`, and `crypto_onetimeauth`.

WHACL* (WH): A formally verified cryptography library compiled to WebAssembly [3]. We verify `Chacha20`, `Curve25519_51`, `Poly1305_32`, `Salsa20`, and `Hash_SHA2` in WHACL* v3.0.0. To our best knowledge, this is the first time WHACL* is verified.

Libsodium (L0, L3): A cryptography library written in C [2]. VIVIENNE verifies the constant-time implementations of `crypto_aead`, `crypto_auth`, `crypto_stream`, `crypto_onetimeauth`, `crypto_core`, and `crypto_hash` for Libsodium v.1.0.18-stable with optimization levels `-O0` and `-O3`.

BearSSL (B0, B3): An implementation of SSL/TLS in C. We verify the constant-time functions `aes_ct_cbcenc` and `des_ct_cbcenc` and the non constant-time functions `aes_big_cbcenc` and `des_tab_cbcenc`. B0 includes the functions with optimization level `-O0` and B3 is optimization `-O3`.

Almeida et al. [10] (A0, A3): Five constant-time and three non-constant-time implementations of `select` and `sort`. We analyze WebAssembly binaries compiled with optimization levels `-O0` and `-O3`.

Lucky 13 (Lu0, Lu3): A known timing vulnerability [11] of TLS implementations (see Listing 1). We analyze function `tls1_cbc_remove_padding` of OpenSSL 1.0.1 [26] with optimization levels `-O0` and `-O3`.

B. Results

This section discusses the evaluation results for each of the research questions. Table I presents the aggregated results of the analysis with VIVIENNE. The columns under `Bench` describe the benchmarks, i.e. the abbreviated library name, `BS`, and the number of analyzed algorithms, `A`. The next two columns present VIVIENNE's results with loop unrolling (VIVIENNE$_{\text{unroll}}$) and with loop invariant (VIVIENNE$_{\text{inv}}$). We report the number of verified constant-time implementations, ✓, the number of vulnerable implementations ✗, the number of formulas subject to simplification, `#FS`, and the number of queries that VIVIENNE propagates to the SMT solver, `#SS`. Note that `#SS` is the subset of `#FS` that requires a call to the SMT solver. We highlight in red the incomplete results. For example, VIVIENNE with loop unrolling (VIVIENNE$_{\text{unroll}}$) was able to verify successfully five out of six implementations of WH within the time limit of 90 minutes. Appendix A includes the full evaluation results for VIVIENNE$_{\text{unroll}}$, while the results for VIVIENNE$_{\text{inv}}$ are available as supplementary material online [27].

1) RQ1: Can we use RelSE for constant-time analysis of real-world cryptographic implementations in WebAssembly? : To evaluate the effectiveness of VIVIENNE in analyzing cryptographic libraries, we consider the rate of successfully analyzed algorithms for both secure (✓) and insecure (✗) implementations. The summarized results (Sum) in Table I show that VIVIENNE$_{\text{unroll}}$ analyzes successfully 44 out of 45 constant-time implementations and 11 out of 12 non-constant-time implementations for a total 55/57 implementations. This corresponds to 96% success rate while reporting no false positive. The two outliers are `Hacl_Curve25519_51_scalarmult` of WH and `tls1_cbc_remove_padding` of Lu0. The former contains a loop with 256 iterations, each generating 9108 queries. One of these queries affects an increasingly large part of the total execution time for an iteration. The corresponding formula models the satisfiability of a branch condition that depends on the stack pointer, which WHACL* stores in memory. As a result, the formula has to encode the whole memory, which contributes with 3054 new memory stores for every iteration, thus increasing the time for the generation and simplification of the formula. This can be inferred from the results of Table I, where the total six implementations of WH generate 126K formulas (`#FS`), of which 80896 correspond to `Hacl_Curve25519_51_scalarmult`.

The second outlier is `tls1_cbc_remove_padding` with `-O0`, which contains a loop with non-constant bound, as reported in line 9 in Listing 1. The lack of a constant bound forces VIVIENNE$_{\text{unroll}}$ to consider all possible values for `rec->data[l-1]`, which is an eight-byte value. This leads to maximum 256 iterations for every path that visits the loop. We find that the optimization level `-O0` includes a number of stack operations that modify the memory at every iteration. As we can see in Table I, this leads to 25K `#FS` and 4K `#SS`. The former requires on average 0.01 seconds (4 minutes in total) for simplification, whereas the latter requires 0.87 seconds (58 minutes in total) for SMT solving.

In summary, our results show that RelSE can be used to analyze real-world cryptographic implementations, while the memory operations and loops remain the main bottleneck for the SMT solver. VIVIENNE$_{\text{inv}}$ addresses the challenge of loops by generating relational loop invariants automatically. Further discussion about the SMT solver results of our analysis are available as supplementary material [27].

2) RQ2. To what extent do the automatically generated loop invariants affect the scalability and precision of RelSE?: Our results in Table I show that VIVIENNE$_{\text{inv}}$ is able to successfully analyze constant-time implementations for the first three benchmark libraries. It also analyzes successfully the implementations of WH and Lu0 that VIVIENNE$_{\text{unroll}}$ could not handle. Perhaps surprisingly, VIVIENNE$_{\text{inv}}$ performs poorly on the benchmarks B0, B3, L0, and L3, analyzing only 29% of the implementations. The main reason is that the havocing of modified variables during the invariant generation replaces constant values with unbounded symbolic values. This triggers a path explosion whenever a conditional instruction is analyzed with the new symbolic values. Moreover, it increases the search space for the solver and the complexity of queries whenever a symbolic value indexes the memory in store operations. In Table I, the number of solver queries, `#SS`, for VIVIENNE$_{\text{inv}}$ is larger than for VIVIENNE$_{\text{unroll}}$, which reflects the increase in the complexity because the solver queries (`#SS`) report the formulas that cannot be resolved during the simplification stage. For the benchmarks Tw and B3, the number of queries, `#FS`, also increases due to path explosion. By contrast, for the benchmarks that VIVIENNE$_{\text{inv}}$ analyzes successfully, `#FS` decreases due to the reduction of loop iterations by the loop invariant.

In summary, VIVIENNE$_{\text{inv}}$ analyzes successfully 56% of the implementations, including two implementations for which VIVIENNE$_{\text{unroll}}$ failed. This shows that VIVIENNE$_{\text{inv}}$ complements VIVIENNE$_{\text{unroll}}$ for constant-time analysis.

3) RQ3: How does VIVIENNE *compare to existing approaches for constant-time analysis of WebAssembly?:* To our best knowledge, CT-wasm [9] is the only constant-time analysis tool for WebAssembly. We consider three dimensions for comparison: 1) simplicity, 2) permissiveness, and 3) efficiency. Simplicity refers to the required user effort to verifying a target implementation. CT-wasm relies on type annotations for the program, which can be partially inferred [9]. By contrast, VIVIENNE requires only the security policies and entry-point function, otherwise no further modifications to the generated WebAssembly binary are needed. This reduces the user effort for analyzing a program. Permissiveness refers to the ability of the method to analyze and successfully verify cryptographic implementations. CT-wasm considers the whole memory as secret, which rules out any secure programs that store public values in memory. For example, CT-wasm required refactoring three functions of the TweetNaCl [8] library, i.e. `poly1305_blocks`, `poly1305_update`, and `poly1305_finish`, to make it amenable to verification.

By contrast, VIVIENNE analyzes and verifies the whole implementation of (`crypto_onetimeauth`), with no modifications to the original code. Moreover, VIVIENNE could analyze and verify 57 WebAssembly implementations, including the two libraries CTw and Tw which were verified by CT-wasm [9]. With regards to efficiency, CT-wasm is clearly superior to VIVIENNE because it relies on type checking, while VIVIENNE performs expensive symbolic analysis and constraint solving. However, as we have seen in RQ1, VIVIENNE was still able to analyze real-world WebAssembly implementations within a reasonable time limit.

To summarize, VIVIENNE verifies a larger number of cryptographic implementations than CT-wasm with no need for refactoring and with minimal annotation efforts at the expense of an efficiency cost.

4) Discussion: Ideally, an accurate analysis should be implemented as close to the hardware as possible to avoid vulnerabilities introduced by compiler transformations. For VIVIENNE, the structured control flow of WebAssembly facilitates the analysis, while binary-level analyses face challenges with unstructured control flow and diversity of architectures [28], [16]. This raises the question of whether constant-time programs at WebAssembly level preserve the property at the machine level.

The machine code generated from a WebAssembly binary relies on the compiler of the respective runtime system. Unfortunately, a direct analysis of this machine code with tools like Binsec/Rel [16] is not possible due to the different calling conventions and implementation details of Binsec/Rel. A comparison of VIVIENNE's results at the WebAssembly level with Binsec/Rel's results at the machine level for the benchmarks L0, L3, B0, B3, A0, A3, Lu0, and Lu3 in Table I shows that both tools yield the same result on all benchmarks, except of the `select` implementations of the benchmarks of Almeida et al. [10]. The difference is manifested in the compilation of the `select` implementations at optimization level `-O3`, which Binsec/Rel identifies as insecure. In our experiments, LLVM-10 with flag `-O3` compiles all the C implementations of `select` (A3 in Table I) to one WebAssembly `select` instruction. The compilation from WebAssembly to machine code translates the WebAssembly `select` instruction either to a constant-time conditional assignment (safe), e.g. `cmov` for x86, or to a set of instructions that include a branch instruction (unsafe), depending on the target machine and the compiler implementation. To account for these differences, VIVIENNE provides a command-line option for treating the WebAssembly `select` instruction as unsafe.

V. CONCLUSION

This paper presented VIVIENNE, an open-source tool for analyzing constant-time for WebAssembly programs. VIVIENNE relies on RelSE and leverages the structure of WebAssembly to implement several optimizations, including automated invariant generation. We used VIVIENNE to analyze successfully 57 cryptographic implementations with minimal annotation overhead and no code refactoring. Moreover, VIVIENNE is the first tool to verify constant time for the WebAssembly implementation of HACL*.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their helpful feedback. This work is partially supported by the Wallenberg AI, Autonomous Systems, and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation, the TrustFull project funded by the Swedish Foundation for Strategic Research (SSF), the JointForce project funded by the Swedish Research Council (VR), and Digital Futures.

REFERENCES

[1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, 2017, pp. 185–200.

[2] Libsodium Community, "The sodium cryptography library (Libsodium)," 2018. [Online]. Available: https://libsodium.gitbook.io/doc

[3] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan, "Formally Verified Cryptographic Web Applications in WebAssembly," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 1256–1274.

[4] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of WebAssembly," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 217–234.

[5] D. Molnar, M. Piotrowski, D. Schultz, and D. A. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*, 2005, pp. 156–168.

[6] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, "Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC," in *Fast Software Encryption*, ser. Lecture Notes in Computer Science, T. Peyrin, Ed. Berlin, Heidelberg: Springer, 2016, pp. 163–184.

[7] T. Pornin, "Bearssl, a smaller SSL/TLS library," last accessed May 14, 2021. [Online]. Available: https://bearssl.org/

[8] T. Stüber, "TorstenStueber/TweetNacl-WebAssembly," Oct. 2019. [Online]. Available: https://github.com/TorstenStueber/TweetNacl-WebAssembly

[9] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "CT-wasm: type-driven secure cryptography for the web ecosystem," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 77:1–77:29, Jan. 2019. [Online]. Available: http://doi.org/10.1145/3290390

[10] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX security symposium (USENIX security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 53–70. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida

[11] N. J. Al Fardan and K. G. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 526–540, iSSN: 1081-6011.

[12] L. Clark, "Standardizing wasi: A system interface to run webassembly outside the web," *Mozilla Hacks–the Web developer blog, March*, 2019.

[13] C. Watt, A. Rossberg, and J. Pichon-Pharabod, "Weakening WebAssembly," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 133:1–133:28, Oct. 2019.

[14] M. Vassena, C. Disselkoen, K. v. Gleissenthall, S. Cauligi, R. G. Kıcı, R. Jhala, D. Tullsen, and D. Stefan, "Automatically eliminating speculative leaks from cryptographic code with blade," in *Proc. Symp. on Principles of Programming Languages (POPL 2021)*, 2021. [Online]. Available: http://arxiv.org/abs/2005.00294

[15] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, "Swivel: Hardening WebAssembly against Spectre," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/narayan

[16] L.-A. Daniel, S. Bardin, and T. Rezk, "Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level," in *2020 IEEE Symposium on Security and Privacy (SP)*, May 2020, pp. 1021–1038, iSSN: 2375-1207.

[17] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A Verified Modern Cryptographic Library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 1789–1806. [Online]. Available: http://doi.org/10.1145/3133956.3134043

[18] G. P. Farina, S. Chong, and M. Gaboardi, "Relational Symbolic Execution," in *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, ser. PPDP '19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 1–14.

[19] W. C. Group, "Webassembly Reference Interpreter," 2018. [Online]. Available: https://github.com/WebAssembly/spec/tree/master/interpreter

[20] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer, 2008, pp. 337–340.

[21] R. Brummayer and A. Biere, "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, S. Kowalewski and A. Philippou, Eds. Berlin, Heidelberg: Springer, 2009, pp. 174–177.

[22] B. Dutertre, "Yices 2.2," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 737–744.

[23] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer, 2011, pp. 171–177.

[24] "Ct-wasm," https://github.com/PLSysSec/ct-wasm-ports, accessed: 2021-06-11.

[25] "Binsec/rel," https://github.com/binsec/rel_bench, accessed: 2021-06-11.

[26] "Openssl dtls," https://github.com/openssl/openssl/blob/OpenSSL_1_0_1/ssl/d1_enc.c, accessed: 2021-06-11.

[27] R. M. Tsoupidi, M. Balliu, and B. Baudry, "Supplementary Material for Vivienne: Relational Verification of Cryptographic Implementations in WebAssembly," https://doi.org/10.5281/zenodo.5409477, 2021, accessed: 2021-09-02.

[28] M. Balliu, M. Dam, and R. Guanciale, "Automating Information Flow Analysis of Low Level Code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 1080–1091. [Online]. Available: https://doi.org/10.1145/2660267.2660322

APPENDIX A
EVALUATION RESULTS

Table II shows the complete results of the evaluation for VIVIENNE$_{\text{unroll}}$. The experiments use a time limit of 90 minutes and the reported time values are in seconds and consist of the average and standard deviation after five runs. The first column shows the file name followed by the function that corresponds to the entry point for the analysis. Column `LoC` shows the number of WebAssembly instructions that the analysis accesses, column `AN time` is the analysis time in seconds. When `AN time` is -1, then VIVIENNE was not able to successfully analyze the respective implementations. Column 🐞 shows the number of discovered timing vulnerabilities. #FS is the number of formulas during the analysis and next column shows the time in seconds for the simplification step. #SS is the number of formulas that VIVIENNE forwards to the SMT solver, followed by the average number of expressions in each formula, #Exprs, and the solving time `SS time`. #Exprs is the value that decides selecting the *bindings* solver or the *portfolio* solver. In these experiments, for #Expr $\leq$ 1500, VIVIENNE uses the *bindings* solver, otherwise the portfolio solver.

For example, the first entry for Libsodium -O0 in Table II shows the results for the analysis of functio `crypto_aead_chacha20poly1305_encrypt` from libsodium `aead` module. VIVIENNE goes through 7720 different WebAssembly instructions, not including the multiple accesses for loops. The analysis time is 7.89 ± 0.09 seconds and the analysis did not discover any timing vulnerabilities, generated 11507 formulas that took 0.03 ± 0.48 seconds to simplify. The high value (0.48) of the variance depends on the difference in the complexity of these 11507 formulas. Of the 11507 formulas, 16 where forwarded to the SMT portfolio solver, whereas the rest were simple enough for the analysis to infer their result. The average number of expressions in these 16 formulas is four and the average solving time was 0.04 seconds.

bench/function	LoC	AN time	🐞	#FS	FS time	#SS	#Exprs	SS time
CT-wasm								
salsa20/decrypt	515	0.09 ± 0.00	0	602	< 0.01	0		
salsa20/encrypt	512	0.10 ± 0.01	0	602	< 0.01	0		
sha256/transform	372	0.05 ± 0.01	0	926	< 0.01	0		
sha256/update	409	0.18 ± 0.01	0	1312	< 0.01	0		
tea/decrypt	80	< 0.01	0	72	< 0.01	0		
tea/encrypt	80	0.01 ± 0.00	0	72	< 0.01	0		
TweetNaCl								
core_hsalsa20/core_hsalsa20	356	< 0.01	0	46	< 0.01	0		
core_salsa20/core_salsa20	412	0.01 ± 0.00	0	54	< 0.01	0		
poly1305/crypto_onetimeauth	787	0.11 ± 0.00	0	81	< 0.01	0		
WHACL*								
chacha20/Hacl_Chacha20_chacha20_encrypt	1777	669.91 ± 3.53	0	9665	0.07 ± 2.77	0		
curve25519_51/Hacl_Curve25519_51_scalarmult	-1	-1	0	80896	0.07 ± 0.26	0		
poly1305/Hacl_Poly1305_32_poly1305_mac	1440	1.34 ± 0.01	0	829	< 0.01	0		
salsa20/Hacl_Salsa20_salsa20_encrypt	1887	162.86 ± 1.56	0	8596	0.02 ± 0.71	0		
sha256/Hacl_Hash_SHA2_hash_256	1147	1323.51 ± 7.13	0	14512	0.09 ± 4.56	0		
sha512/Hacl_Hash_SHA2_hash_512	1550	456.20 ± 4.14	0	12287	0.04 ± 1.62	0		
BearSSL -O0								
aes_big/br_aes_big_cbcenc_run	2089	13.04 ± 0.11	32	1111	< 0.01	32	3711	0.36 ± 0.37
aes_ct/br_aes_ct_cbcenc_run	4857	46.54 ± 0.76	0	4233	0.01 ± 0.13	0		
des_ct/br_des_ct_cbcenc_run	3841	1560.52 ± 6.80	0	23463	0.07 ± 1.23	0		
des_tab/br_des_tab_cbcenc_run	1920	24.94 ± 0.16	8	3301	0.01 ± 0.05	8	262	< 0.01
BearSSL -O3								
aes_big/br_aes_big_cbcenc_run	791	7.89 ± 0.09	32	218	< 0.01	32	3327	0.22 ± 0.22
aes_ct/br_aes_ct_cbcenc_run	1717	1.69 ± 0.01	0	493	< 0.01	0		
des_ct/br_des_ct_cbcenc_run	993	6.49 ± 0.03	0	952	0.01 ± 0.19	0		
des_tab/br_des_tab_cbcenc_run	581	3.20 ± 0.03	8	381	0.01 ± 0.15	8	262	< 0.01
Libsodium -O0								
aead/crypto_aead_chacha20poly1305_encrypt	7720	369.83 ± 1.33	0	11507	0.03 ± 0.48	16	4	0.04 ± 0.00
auth/crypto_auth_hmacsha256	13913	4856.64 ± 27.94	0	47679	0.10 ± 0.52	0		
chacha20/crypto_stream_chacha20	3313	228.04 ± 1.61	0	8756	0.03 ± 0.51	2	4	0.04 ± 0.00
poly1305/crypto_onetimeauth_poly1305_donna	3685	20.78 ± 0.09	0	1671	0.01 ± 0.07	0		
salsa20/crypto_core_salsa20	1628	11.99 ± 0.04	0	3513	< 0.01	0		
sha256/SHA256_Transform	11692	136.11 ± 0.95	0	8299	0.02 ± 0.06	0		
sha256/crypto_hash_sha256	13225	536.25 ± 3.84	0	18712	0.03 ± 0.11	0		
sha512/crypto_hash_sha512	13351	295.80 ± 3.18	0	12993	0.02 ± 0.08	0		
Libsodium -O3								
aead/crypto_aead_chacha20poly1305_encrypt	1971	45.06 ± 0.29	0	896	0.05 ± 0.67	16	4	0.04 ± 0.00
auth/crypto_auth_hmacsha256	3256	562.00 ± 4.19	0	4559	0.12 ± 5.32	0		
chacha20/crypto_stream_chacha20	956	0.29 ± 0.01	0	253	< 0.01	2	4	0.04 ± 0.00
poly1305/crypto_onetimeauth_poly1305_donna	940	11.20 ± 0.07	0	223	0.05 ± 0.58	0		
salsa20/crypto_core_salsa20	483	0.01 ± 0.00	0	52	< 0.01	0		
sha256/SHA256_Transform	2171	0.01 ± 0.00	0	479	< 0.01	0		
sha256/crypto_hash_sha256	2980	28.06 ± 0.66	0	1643	0.02 ± 0.66	0		
sha512/crypto_hash_sha512	2844	6.20 ± 0.06	0	1344	< 0.01	0		
Almeida -O0								
naive_select/ct_select_u32_naive	49	0.03 ± 0.00	1	9	< 0.01	3	15	< 0.01
select_v1/ct_select_u32_v1	149	< 0.01	0	14	< 0.01	0		
select_v2/ct_select_u32_v2	93	< 0.01	0	10	< 0.01	0		
select_v3/ct_select_u32_v3	70	< 0.01	0	9	< 0.01	0		
select_v4/ct_select_u32_v4	70	< 0.01	0	9	< 0.01	0		
sort/sort3	254	0.18 ± 0.00	1	298	< 0.01	14	68	< 0.01
sort_multiplex/sort3_multiplex	276	0.02 ± 0.00	0	89	< 0.01	0		
sort_negative/sort3_negative	209	0.16 ± 0.01	1	245	< 0.01	14	68	< 0.01
Almeida -O3								
naive_select/ct_select_u32_naive	5	< 0.01	0	0		0		
select_v1/ct_select_u32_v1	5	< 0.01	0	0		0		
select_v2/ct_select_u32_v2	5	< 0.01	0	0		0		
select_v3/ct_select_u32_v3	5	< 0.01	0	0		0		
select_v4/ct_select_u32_v4	5	< 0.01	0	0		0		
sort/sort3	84	0.07 ± 0.01	3	21	< 0.01	3	229	0.02 ± 0.01
sort_multiplex/sort3_multiplex	74	0.10 ± 0.00	3	17	< 0.01	3	229	0.02 ± 0.02
sort_negative/sort3_negative	74	0.09 ± 0.01	3	17	< 0.01	3	229	0.02 ± 0.02
lucky13 -O0								
tls1_cbc_remove_padding_lucky13/tls1_..._lucky13	-1	-1	5	24978	0.01 ± 0.06	4027	35698	0.87 ± 0.60
lucky13 -O3								
tls1_cbc_remove_padding_lucky13/tls1_..._lucky13	133	960.17 ± 15.52	5	3144	< 0.01	3106	3080	0.25 ± 1.03

TABLE II
EVALUATION RESULTS WITH VIVIENNE$_{\text{UNROLL}}$