Verifying Constant-Time Implementations

Mahyar Emami, Rishabh Iyer, Sahand Kashani
firstname.lastname@epfl.ch

January 13, 2022

1 Introduction

Timing attacks—attacks that extract secrets by measuring timing differences under adversary-controlled inputs—on cryptographic libraries [12, 13] pose a major challenge to information security today. Popular cryptographic libraries (e.g., OpenSSL [9]) run on millions of devices; hence a vulnerability in such a library has the potential to compromise all such devices simultaneously.

Constant-Time programming is the most effective software countermeasure against such attacks. Constant-time programming involves rewriting a program such that (1) its control flow does not depend on program secrets, (2) it does not perform any secret-dependent memory accesses, and (3) it does not use variable-latency instructions to operate on secrets.

However, writing and/or reasoning about constant-time programs is challenging and prone to errors since it typically involves the use of low-level programming languages and programming practices that deviate from software engineering principles. For example, two constant-time violations[1] were found in Amazon's s2n library [2] soon after its release with the second exploiting a timing-related vulnerability introduced when fixing the first.

Prior work (Almeida et. al [11]) propose using formal methods to *automatically* verify whether a program runs in constant time. To do so, they provide a precise framework to model constant-time properties, a sound and complete reduction of constant-timeliness of input programs to assertion safety of a self-product, and design & evaluate an automated tool that verifies the constant-timeliness of cryptographic algorithms from widely used libraries. We provide a summary of this prior work in §2.

In this work, we go beyond the work of Almeida et. al in three ways.

- We reproduced their published results and in doing so identified a limitation that prevents it from scaling to more complex, but ubiquitously-used cryptographic programs (§3).

[1] See pull requests #147 and #179 in [2]

- We showed that an alternate approach might be a better fit for the domain of cryptographic programs by implementing and evaluating an approach based on exhaustive symbolic execution and static dataflow analysis (§4).
- We showed (using existing tools) how to lower the guarantees provided by tools that analyze intermediate representations to the actual binary that runs on hardware (§5).

Our code and results are publicly available [10].

2 Background

In this section, we provide a succint summary of `ct-verif`, in particular its definition of constant time (§2.1), how the tool verifies that a program is constant-time (§2.2) and finally its implementation (§2.3). Here we restrict ourselves to describing only those details that are needed for the rest of the report; detailed outlines of each of the above can be found in our midterm report [5].

2.1 Defining Constant-Time Execution

For a program to be "constant-time", it must not leak any secret data that can then be identified/recovered by an adversary using timing differences.

In their work, Almeida et. al [11] consider three sources of leakage: (1) branches which can leak the evaluated branch conditions, (2) memory operations which can leak the address accessed in load and store instructions due to cache-based timing differences. (3) instructions whose execution time may vary on modern processors based on the provided operands (e.g., integer divide on x86 processors [7]). Here onwards, we refer to such instructions as "variable latency instructions".

In summary, Almeida et. al [11] define a program as constant time if it does not (1) branch based on a secret, (2) access a secret-dependent memeory addresss, and (3) use a secret-dependent value as an operand for variable latency instructions. For the formal definition used by Almeida et. al [11], please refer to our midterm report [5].

2.2 Verifying Constant-Timeliness

We now illustrate using an example how `ct-verif` verifies that a program is indeed constant-time; Fig. 1 describes the example. The program in question copies a sub-array of length `sub_len`, starting at index `l_idx`, from array `in` to array `out`. Both that the starting addresses and lengths of both arrays are publicly observable but the value of `l_idx` and the array contents must

be kept a secret. Note, this program is *not constant-time secure* since the branches on line 5 clearly leak information about `l_idx`.

```
1  void copy_subarray(uint8 *out, const uint8 *in,
2      uint32 len, uint32 l_idx, uint32 sub_len){
3    uint32 i,j;
4    for(i=0,j=0; i<len; i++){
5      if((i >= l_idx) && (i<l_idx + sub_len)){
6        out[j] = in[i];
7        j++;
8      }
9    }
10 }
```

Figure 1: Running example - sub-array copy

To verify that a program runs in constant-time, `ct-verif` verifies that its *self-product* is *safe w.r.t leakage assertions.*

The *self-product* of a program P is a second program Q in which two abstract executions of P take place simultaneously, with the two executions only differing in the value of *secret* inputs and outputs. Said differently, Q interleaves P with a copy of itself and ties all public inputs of P together in both executions.

Fig. 2 illustrates the self-product for our running example; focus now on lines $1-5$ and $12-14$. We first see that all public inputs are tied together (lines $1-4$) and assumed equal in both executions of P. Then both executions proceed in lock-step with each computation performed in both programs simultaneously (lines $5, 12-14$).

`ct-verif` utilizes the fact that the two executions differ only in the values of secret inputs to reduce constant-timeliness to safety w.r.t leakage assertions. For each of the categories of leakage it considers, `ct-verif` inserts into the self-product an assertion that the leakage is the same in both executions. If the assertion holds, then the information leaked was publicly available (since only public inputs are assumed equal), if not, then the leakage differs based on the secret input which violated constant-timeliness.

Fig. 2 illustrates how assertion safety verifies constant-timeliness; focus now on lines $6, 8, 11$. Each of these three lines contain leakage assertions—line 6 pertains to the branch on line 7, line $8, 9$ the branch on line 10 and line 11 the memory access on line 12. When a source of leakage operates only on public inputs (e.g., line 7), the corresponding assertion holds (line 6). However, when constant-time is violated and a source of leakage operates on secret-dependent values (line 10), the corresponding assertion (line $8, 9$) cannot be proved. Hence, if the self-product is free of leakage-assertion failures, it must be constant time.

```
1  assume in = in̂;
2  assume out = out̂;
3  assume len = l̂en;
4  assume sub_len = sub_l̂en;
5  i = 0; î = 0; j = 0; ĵ = 0;
6  assert (i < len) = (î < l̂en); // trivial
7  while (i < len) do:
8      assert ((i ≥ l_idx) && (i < l_idx + sub_len))
9          = ((î ≥ l_îdx) && (î ≥ l_îdx + sub_l̂en) // fails;
10     if ((i ≥ l_idx) && (i < l_idx + sub_len)) then
11         assert i = î && j = ĵ; // trivial
12         out[j] = in[i]; out̂[ĵ] = in̂[î];
13         j = j + 1; ĵ = ĵ + 1;
14     i = i + 1; î = î + 1;
```

Figure 2: Example program product of the sub-array copy program.

2.3 `ct-verif` Implementation

`ct-verif` accepts two inputs—the C implementation of the program of interest and a simple proof harness which invokes the program and annotates its inputs as either public or secret. Fig. 4 illustrates the proof harness for our running example. Given these inputs, `ct-verif` either proves that the program runs in constant time or provides sources of leakage that violate constant-time.

Fig. 3 illustrates the underlying components of `ct-verif`. The program and its harness are compiled down to `llvm` bitcode. The bitcodes are linked together and converted to *Boogie Programming Language* (`.bpl`). Compilation down to `.bpl` is automated through *SMACK* [15], but it is possible to customize the flow if needed and only rely on the the `bitcode`-to-`bpl` conversion of SMACK. The linked `bpl` files are then fed to *BAM! BAM! Boogieman* which performs the key task of generating the self-product, and finally the self-product—which contains assertions and assumptions—is given to *Boogie* [3] to verify assertion safety.

`ct-verif` is publicly available [6].

3 Reproducing Results and Identifying Limitations

In this section we describe our attempts to both reproduce the published results and apply `ct-verif` to more general cryptographic programs.

3.1 Verifying toy programs

To ensure the tool works as advertised, we first tried proving/disproving small examples with obvious answers before reproducing the results reported

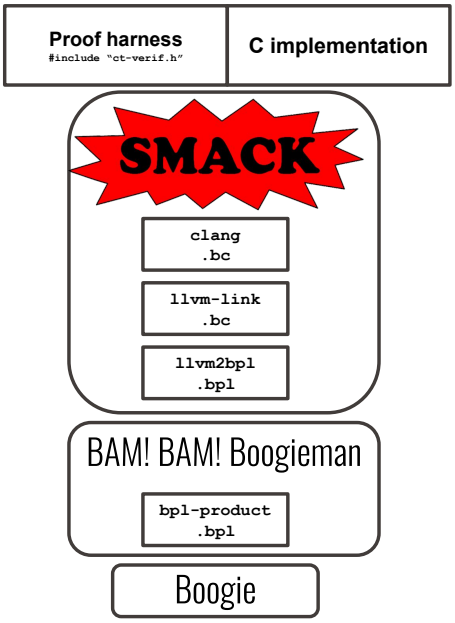


Figure 3: `ct-verif` tool flow

```
1  #include "ct-verif.h"
2
3  void copy_subarray_wrapper(uint8_t *out, const uint8_t *in,
4      uint32_t len, uint32_t l_idx, uint32_t sub_len) {
5
6        public_in(__SMACK_value(out));
7        public_in(__SMACK_value(in));
8        public_in(__SMACK_value(len));
9        public_in(__SMACK_value(sub_len));
10       copy_subarray(out, in, len, l_idx, sub_len);
11 }
```

Figure 4: Proof harness for sub-array copy in Fig. 1.

in the paper [11].

We started by disproving constant-timeness of the code given in Fig. 1. `ct-verif` correctly identifies this code as unsafe (i.e., not constant-time).

A more interesting example is the slightly unusual `vector_add` function in Fig. 5. Notice how by declaring `a[N - 1]` as a public input we make the code constant time. `ct-verif` also successfully identifies this code as constant time. Now if we remove the public input declaration on line 10, then the code will no longer be constant time because the for loop will leak the length of arrays and `ct-verif` also correctly marks the program as non-constant time.

```
#define N 100
void vector_add(uint32_t* a, uint32_t* b, uint32_t* c) {
  int vec_len = a[N - 1];
  for (int i = 1; i < vec_len; i++)
    c[i] = a[i] + b[i] ;
}
void vector_add_wrapper(uint32_t *a, uint32_t* b, uint32_t* c)
    {

    public_in(__SMACK_value(a));
    public_in(__SMACK_value(a[N - 1]));
    public_in(__SMACK_value(b));
    public_in(__SMACK_value(c));

    vector_add(a, b, c);
}
```

Figure 5: A slightly unusual vector addition.

3.2 Reproducing published results

While set out to reproduce all the results in the paper, we ended up only verifying some of them. There were two reasons:

1. Functions reported as constant-time in the paper could not be proven to be constant-time using `ct-verif`;

2. The reported functions could not be found in the `ct-verif` repository, or their build procedures were broken/incomplete.

3.2.1 `tea`

`tea` is a tiny encryption algorithm that consists of about 20 lines of code. We successfully verified `tea` to be constant-time.

3.2.2 `libfixedpointfixedtime`

`libfixedpointfixedtime` is a library that implements a large variety of fixed-point arithmetic operations in constant-time. The paper only reports constant-timeness for 10 functions in the library, but we managed to verify an additional 24 functions using `ct-verif` that were not reported in the paper. These additional functions are: `fix_is_neg`, `fix_is_nan`, `fix_is_inf_pos`, `fix_is_inf_neg`, `fix_eq`, `fix_eq_nan`, `fix_ne`, `fix_cmp`, `fix_le`, `fix_ge`, `fix_lt`, `fix_gt`, `fix_neg`, `fix_abs`, `fix_add`, `fix_sub`, `fix_mul`, `fix_div`, `fix_floor`, `fix_ceil`, `fix_ln`, `fix_log2`, `fix_log10`, `fix_convert_from_int64`, `fix_convert_to_int64`, `fix_round_up_int64`, `fix_ceil64`, `fix_floor64`, `fix_sin`, `fix_cos`, `fix_tan`, `fix_exp`, `fix_sqrt`, `fix_pow`.

3.2.3 `curve25519-donna`

We successfully re-verified the C implementation of the `curve25519-donna` elliptic curve. This function was the hardest among all and took the longest time to prove (about 7 minutes).

3.2.4 `libsodium`

`libsodium` is an easy-to-use encryption library. We attempted to verify 4 functions here (`chacha20, salsa20, sha256, sha512`). We used the Makefiles provided by `ct-verif` which isolate the C files used by each function (i.e., the whole library is not compiled to `llvm` bitcode). We successfully reproduced the paper results by verifying constant-timeness of `chacha20` and `salsa20`, but failed prove it for `sha256` and `sha512` (unlike what the paper claims).

Upon inspecting the C implementation of `sha256` and `sha512` we couldn't identify exactly why these two functions are not constant-time. This implored us look for other methods of asserting constant-timeness which we will follows up on in §4.

3.3 Scalability of `ct-verif`

While reproducing `ct-verif`'s published results, we noticed that the functions under test always had the following characteristics—they needed to be clearly isolated functions in code, they were always of medium complexity (¡1k LOC) and were often internal functions as opposed to client-facing functions. For example, for the ubiquitously used AES encryption algorithm [1], `ct-verif` only verifies individual functions that correctly pad and unpad cipherblocks, as opposed to testing the top-level encryption, decryption functions. In fact, on closer inspection, we noticed that the authors admit to this fact in the code with a comment that states that verifying more complex, but widely used functions such as md5, sha1 is beyond Smack's abilities [4].

To evaluate whether `ct-verif` indeed scaled to more complex cryptographic functions beyond the ones tested in the paper, we performed two experiments.

First, we tried to verify constant-timeness of function `s2n_verify_cbc` from Amazon's `libs2n` library. The test harness for this function is included as one of the examples in the `ct-verif` repository but it had a simple bug where the test harness was calling itself instead of the `s2n_verify_cbc` function. However, there was no build script for this specific function and we had to modify the `Makefile`s in the `libs2n` library to generate `llvm` bitcode instead of a binary. We then manually linked the bitcode library with the test harness and invoked `llvm2bpl` pass—provided by SMACK—and fed the generated `.bpl` files to BAM! BAM! Boogie to generate the program product. Unfortunately. `llvml2pl` resulted in many warnings regarding memory safety and self-product program generation failed to due an resolved reference.

Since the above attempt didn't give us a clear picture, we next compiled the OpenSSL library [9] down to `llvm` bitcode to verify the SHA-1 function. Transformation from bitcode to bpl resulted in over 70GiB of memory usage

and did not terminate after 30 minutes.

Unlike the functions verified in the paper, these two functions are not isolated and the bitcode given to `ct-verif` contains unused code. It seems like SMACK transforms the whole program irrespective of the target function. This approach limits the usability of `ct-verif` since verification requires a good understanding of the underlying library structure to be able to isolate the used code which is often extremely tedious in large codebases.

4 Overcoming Scalabity Issues with an Alternate Approach

Given that constructing self-products failed due to poor dead-code elimination, we implemented an alternative approach based on symbolic execution. The reasoning behind this was that actually stepping through and interpreting the program would ensure that only the relevant code is analyzed. Our tool takes the same two inputs as `ct-verif`—C code containing the function of interest and a test harness that annotates inputs as public or secret. The main difference is that the C code can now also contain arbitrary dead code, unlike for `ct-verif`.

Our approach is fairly straightforward and combines two well known techniques—symbolic execution [14] and static dataflow analysis [17]. Given the program of interest, our approach first assumes that all inputs (both private and public) are symbolic (and not concrete) variables. Then it executes the program of interest with these symbolic inputs, allowing it to explore all feasible paths through the code. While stepping through the code, it also propagates the "secret" taint using static dataflow analysis. Finally, when interpreting instructions corresponding to the leakages being considered, it adds assertions that checks if the leakage does not involve a secret-tainted value. If the tool can explore all paths through the code without assertion failures, the code is indeed constant time, if not, the assertion failure points to a constant-time violation just like in `ct-verif`.

Given that the two primary techniques underlying our tool are so well known, we were able to find an open-source tool (`klee-taint` [8]) that combines the two of them. We were able to use `klee-taint` off-the-shelf for symbolic execution and dataflow analysis. We only had to add the assertions to check for secret dependent leakages to `klee-taint`'s LLVM interpreter and this took only 16 LOC. Note, we used `klee-taint` in direct taint mode which propagates the taint only for direct operations (e.g., assignments, arithmetic operations, etc).

To evaluate `klee-taint`'s scalability, we used it to verify the constant-timeliness for 7 top-level cryptographic primitives from OpenSSL 3.0's libcrypto, and 3 programs analyzed by `ct-verif`. To ensure we had both positive and negative examples, we included the running example and a previously

patched vulnerability in OpenSSL. Note, `ct-verif` does not scale to any of the 7 programs from OpenSSL.

Program	**Remarks**
OpenSSL 3.0 AES	Verified that code runs in constant-time
OpenSSL 3.0 ChaCha	Verified that code runs in constant-time
OpenSSL 3.0 ECDHE	Verified that code runs in constant-time
OpenSSL 3.0 MD5	Verified that code runs in constant-time
OpenSSL 3.0 MD4	Verified that code runs in constant-time
OpenSSL 3.0 Poly1305	Verified that code runs in constant-time
OpenSSL 3.0 SHA-256	Verified that code runs in constant-time
OpenSSL 1.1 RSA (CVE-2018-0737)	Reproduced cache-based leakage
libsodium ChaCha20	Verified that code runs in constant-time
Tiny Encryption Algorithm (TEA)	Verified that code runs in constant-time
Running example from Fig. 1	Correctly identified branch-based leakage

Table 1: Programs we tested `klee-taint` on.

Table 1 lists the programs we used. For each of the above programs, `klee-taint` is able to either verify constant-timeliness or provide a source of leakage within minutes. For all programs except ECHDE `klee-taint` terminates its analysis in ¡5 mins. ECHDE takes 22 mins since it is the most complex algorithm and the only one that involves symmetric key generation.

Limitations: A symbolic execution-based tool is not a panacea and has its own limitations. Since symbolic execution explores all paths through the code in a naive manner, it does not scale well to some kinds of cryptographic code, particularly code with loops. Loops with a constant bound can be handled properly, but those that could run an arbitrary number of times cause path explosion during symbolic execution. For instance, generating a prime number by repeatedly generating a likely prime and checking whether it really is prime could, in theory, never terminate. Symbolic execution will thus not terminate either. This is an operation that can happen during key generation if we model the source of randomness as symbolic.

5 Does constant-time hold for binaries?

One shortcoming of both `ct-verif` and `klee-taint` is that they both operate only at the level of LLVM bitcode and can therefore only formally attest to whether a program's representation as bitcode is constant-time. Translating a bitcode program into binary is a straightforward process and is gener-

ally a simple one-to-one mapping between bitcode instructions and assembly. In theory an assembly program should therefore preserve the constant-time property if the source LLVM bitcode is proven to be constant-time.

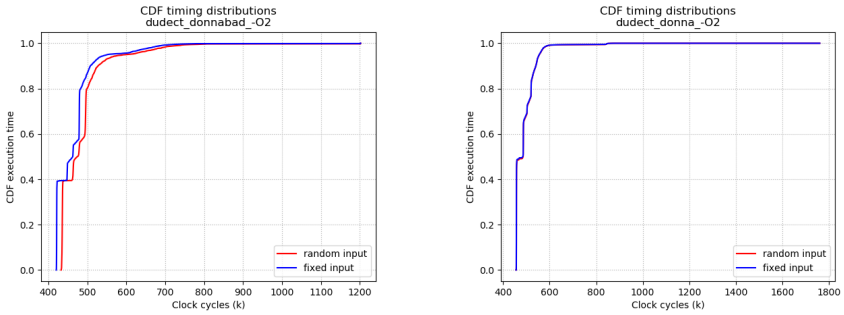
However, while this argument does hold for simple RISC-like ISAs, it does not for complex CISC-like machines like the x86 architecture. There is in reality no true "x86" machine as no x86 processor actually executes the CISC instructions it receives as input directly. Instead, these CISC instructions are decomposed into RISC-like μOps before being scheduled for execution on a RISC-like backend. This decomposition is handled transparently by microcode sequencers in the CPU's frontend and is not under the control of the user. Microcode updates can also change instruction behavior in ways that conflict with a formal tool's prior modelling. Asserting that a constant-time LLVM bitcode program remains constant-time when translated to x86 assembly is unclear as users have no way of knowing how an instruction is decoded into μOps.

The `dudect` project appeared after `ct-verif` and suggested the only way to decide whether a program is constant-time or not is to actually run it and time its execution [16]. The main idea behind `dudect` is to run a supposedly constant-time program under two environments and measure its execution time: (1) using fixed secret inputs, and (2) using random secret inputs. If the program is constant-time, then the timing traces of these two programs should be indistinguishable. However, machines do not have deterministic execution times due to caches and OS scheduling, so relying on the execution time alone to determine constant-timeness would be incorrect. `dudect` therefore proposes using a statistical method—an "equivalence of means" t-test—to determine whether multiple program executions under the two aforementioned environments are statistically indistinguishable. If yes, then the program is *possibly* constant-time. If not, then the program is *definitely not* constant-time. Combined with a formal constant-time proof obtained from `ct-verif`, the mechanism proposed by `dudect` can validate that the μOp decomposition of a program is constant-time.

In fact, `dudect`'s method of operation lends itself well to a graphical validation when a constant-time violation is detected. Indeed, repetitively running the program with fixed/random private inputs and plotting the CDF of its average execution time generates curves that should perfectly overlap if the program is constant-time. We modified `dudect` to automatically extract and plot this timing information.

We ported the 9 examples from the `ct-verif` paper to use `dudect`'s test harness and were able to validate their constant-time property. We only show 2 examples below that show what a non-CT and a CT CDF of the same algorithm should look like as all other examples in `ct-verif` and `dudect` are constant-time and their curves overlap perfectly. Figure 6 shows an example of running the `dudect` method on two implementations of `curve25519` that differ in the constant-time property (that `ct-verif` was

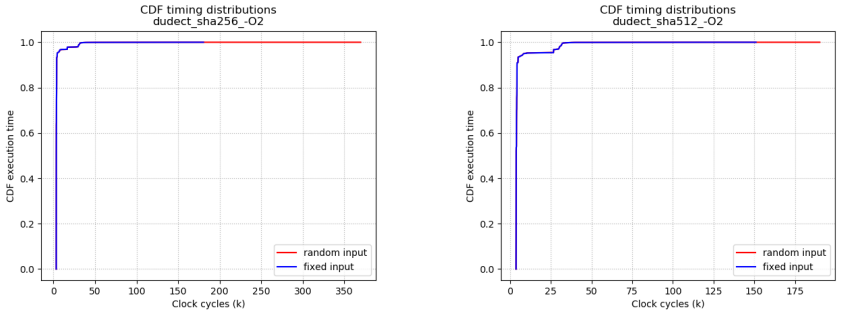
also able to formally verify).



(a) Non-CT implementation (b) CT implementation

Figure 6: CDF timing behavior of a non-CT vs CT implementation of an elliptic curve.

Figure 7 in turn shows how an example that `ct-verif` could not prove as constant-time *could* be constant-time since the execution curves overlap perfectly. The examples shown here are `libsodium`'s `sha256` and `sha512` implementations which the `ct-verif` paper claimed could prove the constant-time property for, but which we were not able to replicate as discussed in §3. Running these examples through `dudect` suggests that they are constant-time as their CDF execution time curves perfectly overlap.



(a) SHA256 (b) SHA512

Figure 7: CDF timing behavior of `libsodium`'s SHA256/SHA512 implemenation. Both are statistically constant-time.

In summary, we find `dudect`'s statistical methodology is a good way to translate a formal tool's proof of constant-timeness to a probable guarantee on he end-binary that runs on a processor.

6 Conclusion

In this work, we explored the use of formal methods to verify whether a program runs in "constant-time". We first reproduced the results from an existing tool that uses program self-products and identified its scalability limitations. We then showed that an alternate (if less elegant) approach might be a better fit for the domain by implementing an approach based on exhaustive symbolic execution. Finally, we showed how to lower the guarantees (albeit empirically) provided by tools that analyze intermediate representations to the actual binary that runs on hardware.

References

[1] Advanced Encryption Standard (AES). https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[2] Amazon s2n-tls Github Repository. https://github.com/aws/s2n-tls.

[3] Boogie Github Repository. https://github.com/boogie-org/boogie.

[4] Comment re: Smack's scalability in ct-verif source code. https://github.com/imdea-software/verifying-constant-time/blob/master/examples/openssl/ssl3_cbc_digest_record.c#L570.

[5] CS550 Project Midterm Report. https://github.com/rishabh246/cs550-code/blob/main/reports/midterm.pdf.

[6] CT-Verif Github Repository. https://github.com/imdea-software/verifying-constant-time.

[7] Intel 64 and 32 bit architecture manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[8] KLEE-taint Github Repository. https://github.com/feliam/klee-taint.

[9] OpenSSL Github Repository. https://github.com/openssl/openssl.

[10] Project Repository. https://github.com/rishabh246/cs550-code.

[11] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security Symposium*, 2016.

[12] Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

[13] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make sure dsa signing exponentiations really are constant-time. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[14] James C. King. Symbolic execution and program testing. 19(7), 1976.

[15] Zvonimir Rakamaric and Michael Emmi. SMACK: decoupling source language details from verifier implementations. In *Computer Aided Verification CAV*, 2014.

[16] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '17, page 1701–1706, Leuven, BEL, 2017. European Design and Automation Association.

[17] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.