Finding and Eliminating Timing Side-Channels in Crypto Code with Pitchfork

Craig Disselkoen
UC San Diego

Sunjay Cauligi
UC San Diego

Dean Tullsen
UC San Diego

Deian Stefan
UC San Diego

Abstract

We present Pitchfork, a symbolic analysis tool which verifies that code is constant-time and does not leak secret values. Writing constant-time code is the de-facto defense against timing side-channel attacks, used today by many major cryptographic libraries. Unfortunately, writing constant-time code is notoriously difficult. Even experts have repeatedly written buggy code and overlooked critical vulnerabilities in widely used cryptographic libraries. To address these issues, Pitchfork verifies that code is constant-time. In particular, we used Pitchfork to verify that a large portion of Mozilla's NSS cryptographic library is constant-time, while also finding several constant-time vulnerabilities, including a critical vulnerability which was assigned CVE-2019-11745.

1 Introduction

Constant-time programming is the de-facto approach to writing critical code—in particular cryptographic code—that is robust against timing side-channel attacks. Unfortunately, it is notoriously hard to write constant-time code: Not only do experts fail to adequately write truly constant-time code [5,7,20], but even the process of fixing these mistakes can lead to further vulnerabilities [1, 28]. Almost yearly, attackers break cryptosystems using these attacks. For example, earlier this year, timing side-channels were found in several libraries implementing elliptic curve cryptography, as well as in code running on smart cards and TPM chips [15, 21].

The only way we can hope to eliminate these kinds of vulnerabilities is with rigorous verification. Formal verification allows us to mathematically prove that code is constant-time (and give a counterexample when it's not). To this end, we present a verification tool, Pitchfork, which ensures functions are constant-time with respect to secret values such as encryption keys or message plaintexts.

We designed Pitchfork to easily analyze both cryptographic primitives and protocol-level cryptographic code. Protocol-level code has been left underanalyzed by much of the previous work on constant-time verification, which has instead focused on cryptographic primitives. However, flaws in the implementation of protocol-level code are responsible for high-profile vulnerabilities such as Lucky 13 [1]. Indeed, as we describe later, Pitchfork has found several constant-time violations in protocol-level code from real-world codebases.

Pitchfork uses underconstrained symbolic execution augmented with dynamic taint tracking to verify that code is constant-time. In particular, it uses a *shadow memory* to track secrets even as they are stored to and loaded from memory. Pitchfork also allows the user to specify *function hooks* which are executed in lieu of a call to a particular function; this allows Pitchfork to focus on protocol-level code while ignoring implementation details of cryptographic primitives. With these techniques, Pitchfork was able to verify protocol-level code in both `libsignal` [27] and Mozilla's NSS cryptographic library [22]. Our verification effort, however, also revealed several constant-time vulnerabilities in NSS, including a critical vulnerability which was assigned CVE-2019-11745.

2 Motivation

In this section, we give a brief introduction to constant-time programming and show how Pitchfork verifies constant-time properties.

2.1 Constant-time programming

Timing side-channel attacks have repeatedly been used to leak sensitive data, particularly in cryptographic code [1, 18, 28]. Timing vulnerabilities in cryptographic code arise when secret data influences either control flow or the addresses of memory accesses. For example, the function `check_password` below is vulnerable to timing attacks, as secret data—namely, the correct password—influences a branch condition.

```
int check_password(char* password, char* guess) {
    for (int i = 0; i < 32; i++) {
        if (password[i] != guess[i]) {
```

```
            return -1;  // fail
        }
    }
    return 0;  // success
}
```

Specifically, this function's execution time depends on how many of the initial characters of the `guess` were correct. If an attacker is allowed to make repeated guesses, they can efficiently discover the secret password by successively brute-forcing each character, moving to the next character when the function's execution time increases.

Writing code that is free from timing vulnerabilities is difficult. In practice, experts use a collection of constant-time recipes, which can be distilled down to three rules:

1. Secret values must not influence the program's control flow—otherwise, an attacker could infer secret values from the program's execution time.[1]

2. Secret values must not influence the addresses of memory accesses—otherwise, an attacker could infer secret values using cache side-channel attacks.

3. Secret values must not influence the inputs to any variable-time machine operation (such as integer division on many processors).

To fix the above function, we need to remove the control flow dependency on the secret password data:

```
int check_password(char* password, char* guess) {
    int rv = 0;
    for (int i = 0; i < 32; i++) {
        rv |= (password[i] ^ guess[i]);
    }
    return rv;
}
```

Here, we keep track of the final return value `rv`, updating it using bitwise operations so that its final value is nonzero if any character mismatched. Crucially, all 32 characters are compared regardless of whether the initial characters were guessed correctly.

Timing vulnerabilities in cryptographic code can be very difficult to find, often going unnoticed for years even in major cryptographic libraries. Furthermore, attempts to fix these vulnerabilities are often incomplete, or even introduce new timing vulnerabilities in the process. As an example, the recent fix for a timing side-channel vulnerability in Mozilla's NSS cryptographic library [22] failed to fully eliminate all dependency of the control flow on secret data. Using Pitchfork, we analyzed the "fixed" code and found the remaining vulnerability. That vulnerability was fixed in a subsequent patch, and the resulting code was verified with Pitchfork before it was committed. This example demonstrates the need for a tool which can rigorously verify cryptographic code to be constant-time.

[1] In addition, an attacker may be able to infer secret values from a cache side-channel attack, if the control flow leads to different memory access patterns, even for public data.

2.2 Constant-time verification

Unfortunately, most safety-critical constant-time code has not been formally verified to be constant-time. Instead, it simply undergoes manual review—a process which has repeatedly missed timing vulnerabilities in the past [1, 28]. Some work has been done on programming languages with formally verifiable constant-time properties [10, 29, 30], but these tools cannot easily be applied to existing C/C++ code. Furthermore, previous constant-time verification efforts focus on verifying cryptographic primitives [3, 12]; however, modern cryptographic primitives such as Salsa20, Poly1305, and Ed25519 are designed to be constant-time by construction. In contrast, flaws in the implementation of protocol-level code have resulted in high-profile vulnerabilities such as Lucky 13 [1]. Finally, previous tools for constant-time verification [3] don't provide meaningful feedback to aid the development process; they simply report whether a given program is constant-time or not.

Our experience using these tools identified the need for a tool which can verify not only cryptographic primitives, but more complex cryptographic protocol code. We also need this tool to provide detailed feedback to the developer about vulnerabilities, such as the vulnerability's location in the source code and the conditions under which it occurs.

3 Constant-time verification with Pitchfork

We designed Pitchfork to fulfill these needs: to verify both cryptographic primitives and protocol-level code, and to provide detailed feedback to developers about vulnerabilities. Pitchfork uses underconstrained symbolic execution augmented with dynamic taint tracking to directly verify the three simple properties given in Section 2.1. In this section, we describe how Pitchfork analyzes cryptographic code to verify these properties.

3.1 Taint propagation

Consider the `mbed-crypto` [4] AES decryption function in Figure 1. Pitchfork is designed to find code locations where *secrets* could leak via timing channels. Hence, to start, the developer must indicate an initial set of variables, function arguments, or struct fields which contain secret data. Here, we want to ensure that the AES keys are not leaked. Accordingly, we need to mark the AES keys secret, and other inputs (including the ciphertext) as public. To do this, we note that these keys are stored in the `rk` field of the struct `mbedtls_aes_context`:

```
int mbedtls_internal_aes_decrypt(
      mbedtls_aes_context *ctx,
      const unsigned char input[16],
      unsigned char output[16]
) {
    // ...
    RK = ctx->rk;  // secret AES round keys

    X0 = ( (uint32_t) input[0]       )    \
       | ( (uint32_t) input[1] <<  8 )    \
       | ( (uint32_t) input[2] << 16 )    \
       | ( (uint32_t) input[3] << 24 );   \
    X0 ^= *RK++;
    // {repeat to define X1, X2, X3,
    //    using input[4] through input[15]}

    for ( i = ( ctx->nr >> 1 ) - 1; i > 0; i-- )
    {
        Y0 = *RK++ ^ RT0[ ( X0       ) & 0xFF ] ^ \
                     RT1[ ( X3 >>  8 ) & 0xFF ] ^ \
                     RT2[ ( X2 >> 16 ) & 0xFF ] ^ \
                     RT3[ ( X1 >> 24 ) & 0xFF ];
        // loop body continues ...
    }

    // function continues ...
}
```

Figure 1: Excerpt from the function `mbedtls_internal_aes_decrypt()`, with macros expanded and inlined, and some formatting adjusted to fit the page.

```
typedef struct mbedtls_aes_context {
    int nr;         /*!< The number of rounds. */
    uint32_t *rk;   /*!< AES round keys. */
    // {more fields ...}
} mbedtls_aes_context;
```

In this struct, we mark the `nr` field as public, and the `rk` field—which holds the AES keys—as a (public) pointer to an array of secret data. We do this by creating a Rust object describing the struct:

```
_struct("mbedtls_aes_context", vec![
  default(), // nr
  pub_pointer_to(array_of(sec_i32(), 64)), // rk
  // ... more fields ...
])
```

and then passing this object to Pitchfork using its API.

Pitchfork then *propagates* the tainted-secret values as it symbolically executes the function, marking the result of each operation secret if any of its inputs were secret. Furthermore, to track secret values even as they are stored to and loaded from memory, Pitchfork uses a *shadow memory*: While the primary memory stores symbolic values (as in standard symbolic execution), the shadow memory stores for each address a flag indicating whether that address's current contents are tainted or not. Thus on line 13 when the pointer `*RK` is dereferenced, Pitchfork looks up the corresponding

```
if (context->doPad) {
  if (context->padDataLength != 0) {
    rv = (*context->update)(context->cipherInfo,
                            pLastPart, &outlen, maxout,
                            context->padBuf,
                            context->blockSize);
    if (rv != SECSuccess) {
      // ...
    } else {
      unsigned int padSize = (unsigned int)
        ↪ pLastPart[context->blockSize - 1];
      if ((padSize > context->blockSize) || (padSize ==
        ↪ 0)) {
        crv = CKR_ENCRYPTED_DATA_INVALID;
      } else {
        // ...
      }
    }
  }
}
```

Figure 2: Excerpt from the function `NSC_DecryptFinal()` in the NSS cryptographic library, version 3.46. Some comments not relevant to the current discussion have been omitted, and some formatting has been adjusted to fit the page.

address in the shadow memory and finds that the value currently stored there is secret—it stems from the `rk` struct field which was previously marked secret. Therefore, Pitchfork also marks the resulting value `X0` as secret.

Finally, Pitchfork reports a constant-time violation when secret-tainted data is used in a branch condition or memory address. In this example, Pitchfork reports a violation on line 19, as the array index (the expression `X0 & 0xFF`) uses `X0`, which has been marked secret.

3.2 Analyzing protocol-level code

In this section, we show how Pitchfork finds a real constant-time violation in the protocol-level function `NSC_DecryptFinal` (Figure 2) from Mozilla's Network Security Services (NSS) cryptographic library [22]. NSS is used by many applications, most notably Firefox.

When analyzing protocol-level code, we don't want to get bogged down analyzing complicated but uninteresting functions. Thus, Pitchfork allows the user to specify *function hooks* which are executed in lieu of a call to a particular function. For example, on line 3, the call to the function pointer `context->update` invokes a block cipher decryption primitive. Since we wish to analyze the protocol and not the primitives, we allow the user to choose to supply a function hook which is executed instead of the decryption primitive itself. The function hook writes data marked secret into the output buffer `pLastPart`, and writes an appropriately constrained length value to the output-length parameter `&outlen`. Then, Pitchfork can analyze the critical protocol-level code

abstract from the implementation details of the block cipher primitive, which can be verified separately.

After `context->update` writes (secret) decrypted data into the buffer `pLastPart`, the following code strips padding from the block cipher output. On line 10, the code determines the size of this padding by reading the last byte of the decrypted data. Since the buffer contents were marked secret, Pitchfork also marks the resulting value `padSize` as secret. Then, on line 11, the code uses `padSize` as part of a branch condition. This is dangerous, as the conditional branch may leak information about the padding through a timing side-channel: Specifically, it could lead to a *padding-oracle attack*, allowing an attacker to recover the plaintext. Pitchfork identifies this leak and correctly reports a constant-time violation.

We use function hooks for several purposes. In the example above, we used a function hook to ignore the implementation details of a block cipher primitive, which could be analyzed separately. Similarly, function hooks allow Pitchfork to avoid analyzing the implementation of random number generation functions, concurrency primitives, or logging frameworks, instead treating these functions as black boxes. This allows Pitchfork's analysis to focus on the code which is most likely to contain vulnerabilities.

4 Implementation of Pitchfork

Haybale. At the core of Pitchfork is Haybale, a new symbolic execution engine which we implemented for this work. Haybale implements *underconstrained symbolic execution*: it can analyze single functions rather than entire programs, as in UC-KLEE [24] or Sys [6]. For instance, Haybale can analyze each function in a library individually, using symbolic reasoning to consider all possible values of the function arguments, rather than having to analyze an entire executable which contains calls to the library. Haybale, like other existing symbolic execution engines, explores all *paths* through a function (sequences of control flow decisions). Moreover, Haybale allows the user to perform various analyses on these paths—e.g., find values of the function inputs which exercise that path, ask whether the path could result in a particular return value, or determine whether a particular pointer value encountered along the path could ever be `NULL`.

Incremental solving. Haybale leverages the *incremental solving* mode of modern SMT solvers to perform efficient backtracking while exploring related paths. With incremental solving, SMT solvers can partially revert their state, removing recent constraints but retaining important solving work already completed. When Haybale completes its analysis of a path, it uses incremental solving to revert to the program state where the completed path diverges from the next path to analyze. This allows the SMT solver to reuse its analysis of the entire common prefix of the two paths.

Symbolic memory. The symbolic representation of memory contents is considered a crucial design consideration in symbolic execution engines [6, 8, 11, 13]. For Haybale, we chose a simple flat memory model inspired by Sys [6], in which the memory is represented by a single symbolic array which maps from indices to 8-bit values. Our experience, surprisingly, showed us that this simple model outperformed a more complex model based on 64-bit memory "cells", which was designed to optimize for the common case of 64-bit or 32-bit memory operations. We hypothesize that modern SMT solvers are optimized to handle array operations well—Haybale uses Boolector [23], an SMT solver specializing in bitvector theory which has done well at recent SMT competitions—and a flat memory model is the simplest way to express Haybale's intent to the SMT solver.

Pitchfork. Pitchfork extends Haybale with dynamic taint tracking in order to determine which values in registers and/or memory are influenced by secrets and which are not. Pitchfork expects the user to annotate some function arguments or struct fields as secret—e.g., secret keys or message plaintext—and then it propagates these annotations through the program as it executes, marking the result of an operation secret if any of its inputs were secret. Whenever it encounters conditional branches or memory accesses, Pitchfork reports a constant-time violation if the branch condition or memory address is marked secret.

LLVM. Pitchfork and Haybale operate at the level of LLVM IR. This allows them to analyze code written in C/C++, Rust, Swift, Go, or any other language which can compile to LLVM IR. Compared to some other symbolic execution tools which analyze executable binaries [11, 26], operating on LLVM IR allows Pitchfork to remain closer to the source level, allowing it to, e.g., report the source line number and filename for any constant-time violations it finds. It also allows Pitchfork to leverage the type-level information in the LLVM bitcode; Pitchfork uses this to greatly accelerate and simplify the process of providing public/secret annotations for function arguments and even complicated data structures.

Rust. Both Pitchfork and Haybale are implemented in the Rust language. This is a departure from previous notable symbolic execution engines, which have been implemented in C++ [8], Python [26], or Haskell [6]. Compared to Python, using Rust avoids the performance overheads of garbage collection or a global interpreter. It also provides strong static typing, which avoids the problem of having a long-running analysis halted by a simple type error in results-reporting code. In contrast, compared with C++, we found Rust's strong memory safety guarantees helped us avoid many tricky bugs such as segmentation faults or use-after-frees, although we did still experience a few of these bugs at the interface between Haybale and Boolector [23], which is written in C. (Our experience in this respect aligns with that reported by the Sys developers in [6].) Although Rust demands some extra work to satisfy its strict rules (e.g., its borrow checker), this paid

```
1  if (context->doPad && context->multi) {
2    // ...
3    crv = NSC_DecryptUpdate(hSession, pEncryptedData,
     ↪  ulEncryptedDataLen, pData, &updateLen);
4    if (crv == CKR_OK) {
5      maxoutlen -= updateLen;
6      pData += updateLen;
7    }
8    finalLen = maxoutlen;
9    crv2 = NSC_DecryptFinal(hSession, pData, &finalLen);
10   if (crv == CKR_OK && crv2 == CKR_OK) {
11     *pulDataLen = updateLen + finalLen;
12   }
13   return crv == CKR_OK ? crv2 : crv;
14 }
```

Figure 3: Excerpt from the function `NSC_Decrypt()` in the NSS cryptographic library, version 3.46.

off when dealing with tricky memory safety bugs, and gave us confidence to write and refactor our code freely.

5 Evaluation

We used Pitchfork to verify several functions from Mozilla's NSS cryptographic library [22]—widely used by many applications including Firefox—and from the `libsignal` cryptographic library [27]. Table 1 shows the results of Pitchfork's analysis on `libsignal` commit `71954c5` and on NSS commit `ee786a6d6`; most of these functions were verified by Pitchfork to be constant-time, up to its assumptions.[2] For some functions, Pitchfork's analysis timed out; in these cases, more aggressive assumptions—e.g., more constraints on input variables, or more function hooks to assume correct the implementations of more helper functions—may allow Pitchfork to verify the functions more quickly.

Pitchfork found several constant-time violations in NSS version 3.46, in the functions marked with † in Table 1. (As shown in Table 1, Pitchfork's analysis of NSS commit `ee786a6d6` confirms that these violations have been fixed.) We discussed one of those vulnerabilities in Section 3.2. In the remainder of this section, we discuss two other vulnerabilities in NSS 3.46 which were found by Pitchfork, including CVE-2019-11745.

`NSC_Decrypt`. Figure 3 shows a constant-time violation in NSS version 3.46, which was found by Pitchfork. In this code, `NSC_Decrypt()` first calls `NSC_DecryptUpdate()` and then `NSC_DecryptFinal()` before returning. We can see the efforts taken by the programmers to have this protocol-level code remain constant-time; for instance, even if the call to `NSC_DecryptUpdate()` returns an error value, `NSC_Decrypt()` still calls `NSC_DecryptFinal()`, discarding the result, so as not to expose the return value of `NSC_DecryptUpdate` via a timing channel. However, Pitchfork's analysis of the called function `NSC_DecryptFinal()` shows that its return value, stored in `crv2` on line 9, could be influenced by secrets—in particular, whether the padding was valid in the decrypted plaintext. This dependence of the return value on the padding validity persists even after the vulnerability in `NSC_DecryptFinal()` shown in Figure 2 (Section 3.2) is fixed: although the fixed `NSC_DecryptFinal()` no longer *leaks* that value through the timing channel, the return value is still dependent on the padding validity, due to the program logic. Unfortunately, the code shown here immediately leaks the `NSC_DecryptFinal()` return value through a timing channel again, as `crv2` is used to influence a branch condition on line 10. Pitchfork correctly

Library	Function	Result
NSS	sftk_SSLMACVerify	Verified †
	stfk_SSLMACSign	Verified
	NSC_EncryptUpdate	Timed out †
	NSC_EncryptFinal	Timed out
	NSC_Encrypt	Timed out
	NSC_DecryptUpdate	Verified*
	NSC_DecryptFinal	Verified †
	NSC_Decrypt	Timed out †
	ssl3_AESGCM	Verified
	ssl3_ChaCha20Poly1305	Verified
	ssl3_SignHashesWithPrivKey	Timed out
	ssl3_MACEncryptRecord	Timed out
	ssl3_ConsumeHandshake	Timed out
	ssl_ConstructServerHello	Timed out
libsignal	sender_chain_key_create	Verified*
	sender_chain_key_get_iteration	Verified
	sender_chain_key_create_message_key	Verified*
	sender_chain_key_create_next	Verified*
	sender_chain_key_get_seed	Verified
	ratchet_chain_key_create	Verified*
	ratchet_chain_key_get_key	Verified
	ratchet_chain_key_get_index	Verified
	ratchet_chain_key_get_message_keys	Verified*
	ratchet_chain_key_create_next	Verified*
	ratchet_root_key_create	Verified*
	ratchet_root_key_create_chain	Verified*
	ratchet_root_key_get_key	Verified
	ratchet_root_key_compare	Verified
	group_cipher_encrypt	Verified*
	group_cipher_decrypt	Verified*
	session_cipher_create	Verified

Table 1: Functions verified by Pitchfork, up to its assumptions. NSS functions are taken from NSS commit `ee786a6d6`; a † indicates that Pitchfork found a constant-time violation in that function in NSS version 3.46, which was fixed before commit `ee786a6d6`. `libsignal` functions are taken from commit `71954c5`. Verified* means that Pitchfork verified the function, but due to its assumptions and/or constraints on the function inputs, its analysis did not reach 100% code coverage. Timed out means that Pitchfork's analysis did not complete due to timing out (or reaching other resource limits), but Pitchfork did not find any vulnerabilities in the allotted time.

[2] Like UC-KLEE [24], Pitchfork makes a number of assumptions which simplify the solver's burden: e.g., it bounds the number of iterations allowed for loops, bounds the sizes of some operations like `memcpy`, and chooses an arbitrary layout of objects in memory rather than considering all possibilities.

flags this as a violation; the correct fix is to use constant-time techniques to update `*pulDataLen` without a timing dependency on `crv2`.

Unlike in the previous examples, here we see how Pitchfork found a violation through its ability to *enter* a call rather than use a function hook to ignore the implementation. It was only because of Pitchfork's analysis of the called function `NSC_DecryptFinal()` that Pitchfork found the value `crv2` to be tainted, thus discovering the constant-time violation. Both function hooks and full interprocedural analysis are important tools; Pitchfork supports both, giving the user maximum freedom to make the tradeoffs appropriate for each analysis.

CVE-2019-11745. Figure 4 shows an excerpt from the function `NSC_EncryptUpdate()` in NSS 3.46, containing the critical-severity bug CVE-2019-11745 which was found by Pitchfork.

Specifically, in the call to `context->update` on line 17, the fourth parameter indicates the maximum length which can be written to the output buffer `pEncryptedPart`. The code sets this parameter to `context->blockSize`, even though only `maxout` bytes remain in the output buffer, and nowhere does `NSC_EncryptUpdate()` confirm that `maxout >= context->blockSize`. This may result in a small out-of-bounds write; the call to `context->update` on line 17 can write up to about `context->blockSize` bytes off of the end of the output buffer.

However, that's not the most serious consequence of the error here, nor was it the problem discovered by Pitchfork. After writing to the output buffer, `context->update` writes to the variable `padoutlen` indicating the number of bytes written. Supposing `context->update` does write the allowed maximum of `context->blockSize` bytes to the buffer, we will have `padoutlen == context->blockSize`. Then, on line 27, the code updates `maxout`, which formerly contained the true number of bytes remaining in the output buffer, by subtracting `padoutlen`, which is greater than `maxout` in this case. Unfortunately, since `maxout` is declared with an `unsigned` type, the subtraction will wrap around and give a very large positive value for `maxout`. Subsequently, on line 31, the next call to `context->update` may write up to `maxout` many bytes—i.e., effectively arbitrarily many bytes—to the output buffer, far out of bounds of the buffer's allocation.

This arbitrarily large out-of-bounds write easily results in a constant-time violation for Pitchfork to detect. In particular, Pitchfork reported the possibility for the out-of-bounds write to overwrite a later pointer with a secret value, so that dereferencing that pointer is a constant-time violation.

```
1  if (context->doPad) {
2    /* deal with previous buffered data */
3    if (context->padDataLength != 0) {
4      /* fill in the padded to a full block size */
5      for (i = context->padDataLength;
6            (ulPartLen != 0) && i < context->blockSize;
7            i++) {
8          context->padBuf[i] = *pPart++;
9          ulPartLen--;
10         context->padDataLength++;
11     }
12     /* not enough data to encrypt yet? then return
       ↪ */
13     if (context->padDataLength != context->blockSize) {
14       // ...
15     }
16     /* encrypt the current padded data */
17     rv = (*context->update)(context->cipherInfo,
18                             pEncryptedPart,
19                             &padoutlen,
20                             context->blockSize,
21                             context->padBuf,
22                             context->blockSize);
23     if (rv != SECSuccess) {
24       // ...
25     }
26     pEncryptedPart += padoutlen;
27     maxout -= padoutlen;
28   }
29   // ...
30 }
31 rv = (*context->update)(context->cipherInfo,
32                         pEncryptedPart,
33                         &outlen, maxout,
34                         pPart, ulPartLen);
35 // ...
```

Figure 4: Excerpt from the function `NSC_EncryptUpdate()` in the NSS cryptographic library, version 3.46. Some formatting has been adjusted to fit the page.

6 Future and related work

There has been a lot of work related to constant-time programming, including both constant-time verification and constant-time language design.

Constant-time verification. Both `ct-verif` [3] and Binsec/Rel [12] verify constant-time properties for existing code. `ct-verif`, like Pitchfork, focuses on existing code in languages such as C/C++; while Binsec/Rel verifies compiled binaries instead. Both of these efforts focus on verifying cryptographic primitives, whereas Pitchfork is designed to verify protocol-level code.

Constant-time language design. Other work on constant-time verification proposes rewriting existing cryptographic code in new languages designed for formal verification. For example, FaCT [10] is a C-like language with formal constant-time guarantees; Jasmin [2] more closely resembles an assembly language; and HACL* [30] provides a formally-verified cryptographic library written in the F* language. Similarly, CT-Wasm [29] extends the WebAssembly portable bytecode language with support for annotating secret data and providing constant-time semantics. In contrast, Raccoon [25] transforms existing LLVM IR to make it constant-time.

Spectre. Even if we eliminate all side-channels due to

constant-time violations, we still have to worry about Spectre attacks [17] and their variants (e.g., [14, 16, 19]). Spectre attacks have received significant attention recently due to the extensive powers they provide to attackers. However, Spectre vulnerabilities are notoriously difficult to exploit. Indeed, for this reason, traditional constant-time violations are *even more dangerous* than Spectre vulnerabilities, as they are far easier for attackers to exploit. In this paper, following most recent work on constant-time programming (e.g., [3, 10, 29]), we consider only sequential execution and leave speculative execution for future work. We recently developed a theoretical foundation for extending today's constant-time programming to the context of speculative execution, providing a defense against Spectre vulnerabilities as well [9]. Our contributions in this paper therefore provide a strong foundation for future work on a program analysis tool which captures both traditional timing side-channel vulnerabilities and Spectre vulnerabilities.

7 Conclusion

Pitchfork uses underconstrained symbolic execution to identify violations of the constant-time programming principles in real cryptographic libraries, including a critical vulnerability in NSS which was assigned CVE-2019-11745. Pitchfork can analyze both cryptographic primitives and protocol-level code, making it a valuable tool for defending today's cryptographic implementations from side-channel vulnerabilities.

Acknowledgments

We would like to thank the NSS developers and Patrick Liu for their help and useful discussions. This work was supported in part by a gift from Cisco, the NSF under Grant Number CCF-1918573, the Global Research Outreach program of Samsung Research, and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

Availability

Both Pitchfork (our constant-time verifier) and Haybale (the general symbolic execution engine on which Pitchfork is built) are available open-source. Pitchfork can be found on GitHub at https://github.com/PLSysSec/haybale-pitchfork, or on Rust's package manager at https://crates.io/crates/haybale-pitchfork Likewise, Haybale can be found on GitHub at https://github.com/PLSysSec/haybale or on Rust's package manager at https://crates.io/crates/haybale.

References

[1] Nadhem J. Al Fardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Security and Privacy (S&P)*, 2013.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

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

[4] ARM. Mbed Crypto. https://github.com/ARMmbed/mbed-crypto, 2019.

[5] Daniel J. Bernstein. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[6] Fraser Brown, Deian Stefan, and Dawson Engler. Sys: a static/symbolic tool for finding good bugs in good (browser) code. In *USENIX Security Symposium*, 2020.

[7] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 2005.

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation (OSDI)*, 2008.

[9] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new Spectre era. In *Programming Language Design and Implementation (PLDI)*, 2020.

[10] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Gregoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. FaCT: A DSL for timing-sensitive computation. In *Programming Language Design and Implementation (PLDI)*, June 2019.

[11] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Security and Privacy (S&P)*, 2012.

[12] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/Rel: Efficient relational symbolic execution for constant-time at binary-level, 2019.

[13] Bassem Elkarablieh, Patrice Godefroid, and Michael Y. Levin. Precise pointer reasoning for dynamic test generation. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[14] Jann Horn. Issue 1528: speculative execution, variant 4: speculative store bypass, 2018.

[15] Jan Jancar. Minerva: A ladder has no windows but can still leak. https://minerva.crocs.fi.muni.cz/, 2019.

[16] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *CoRR*, abs/1807.03757, 2018.

[17] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Security and Privacy (S&P)*, 2019.

[18] Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology*. Springer, 1996.

[19] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! Speculation attacks using the return stack buffer. In *USENIX Workshop on Offensive Technologies (WOOT)*, Baltimore, MD, 2018.

[20] Bodo Moeller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures, 2004.

[21] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. Tpm-fail: TPM meets timing and lattice attacks. In *USENIX Security Symposium*, August 2020.

[22] Mozilla. Network Security Services. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS, 2019.

[23] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector at the SMT competition 2019. In *Proceedings of the 17th International Workshop on Satisfiability Modulo Theories (SMT 2019)*, 2019.

[24] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *USENIX Security Symposium*, August 2015.

[25] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium*, 2015.

[26] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Security and Privacy (S&P)*, 2016.

[27] Signal. Signal protocol C library. https://github.com/signalapp/libsignal-protocol-c, 2019.

[28] Juraj Somorovsky. Curious padding oracle in OpenSSL (CVE-2016-2107). https://web-in-security.blogspot.co.uk/2016/05/curious-padding-oracle-in-openssl-cve.html, 2016.

[29] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. CT-Wasm: Type-driven secure cryptography for the web ecosystem. In *Principles of Programming Languages (POPL)*, January 2019.

[30] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL* : A verified modern cryptographic library. In *ACM Conference on Computer and Communications Security (CCS)*, October 2017.