SideTrail: Verifying Time-Balancing of Cryptosystems

Konstantinos Athanasiou[1], Byron Cook[2], Michael Emmi[3], Colm MacCarthaigh[2], Daniel Schwartz-Narbonne[2(✉)], and Serdar Tasiran[2]

[1] Northeastern University, Boston, USA
konathan@ccs.neu.edu
[2] Amazon Web Services, Seattle, USA
{byron,colmmacc,dsn,tasirans}@amazon.com
[3] SRI International, Menlo Park, USA
michael.emmi@sri.com

Abstract. Timing-based side-channel attacks are a serious security risk for modern cryptosystems. The time-balancing countermeasure used by several TLS implementations (*e.g.* s2n, GnuTLS) ensures that execution timing is negligibly influenced by secrets, and hence no *attacker-observable timing behavior* depends on secrets. These implementations can be difficult to validate, since time-balancing countermeasures depend on global properties across multiple executions. In this work we introduce the tool SideTrail, which we use to prove the correctness of time-balancing countermeasures in s2n, the open-source TLS implementation used across a range of products from AWS, including S3. SideTrail is used in s2n's continuous integration process, and has detected three side-channel issues that the s2n team confirmed and repaired before the affected code was deployed to production systems.

1 Introduction

Timing-based side-channel attacks are a serious security risk for modern cryptosystems; the Lucky 13 attack is a recent example [1]. Current systems deploy one of two prevailing countermeasures to prevent such attacks. One possible mitigation against this threat is to apply the *constant-time* coding principle, where secrets must not influence control-flow paths, memory access patterns, or the cycle counts of instructions. This simplifies local reasoning about timing leaks: if secrets are not used in the prohibited manner, then the code does not exhibit timing side-channels. However, constant-time coding often requires replacing a program's natural control-flow with complicated bitwise operations, and can require significant changes to standard data-structures and APIs, making it difficult to reason about functional correctness. The developers of OpenSSL recently applied a 500+ LOC patch to perform constant-time cipher block chaining (CBC) decoding; the complexity of which led to subsequent issues [2].

The second approach, dubbed *time-balancing*, ensures that execution time is negligibly influenced by secrets. This relaxation from constant-time enables simpler and more readable countermeasures: developers must *balance* a program's

R. Piskac and P. Rümmer (Eds.): VSTTE 2018, LNCS 11294, pp. 215–228, 2018.
https://doi.org/10.1007/978-3-030-03592-1_12

executions to have similar timing footprints, allowing the use of standard operations that depend on secrets. The CBC code from s2n [3], for example, implements time-balancing in fewer than 20 additional lines, and s2n's time-balanced HMAC has been proven functionally correct [4]. However, since time-balancing countermeasures depend on global properties across multiple executions, programmers easily miss subtle timing leaks [5].

In this work, we introduce SideTrail (and its implementation [6]), a deductive verifier for time-balancing countermeasures. SideTrail uses an instruction-level precise timing model and encodes it via a *time counter*. It uses Boogie [7] to precisely reason about control flow and values, including the time counter. We automatically infer invariants over time-counter values with minimal user annotation, and use self-composition [8] to prove that the timing difference between every pair of executions with similar public inputs is below a given bound.

We have used SideTrail to verify the correctness of the timing countermeasures for the data-packet processing stack in s2n. SideTrail is used in s2n's Travis-based continuous integration system [9], has detected three issues that the s2n team has confirmed and repaired before the affected code was used in production systems, and has proved correctness of the repaired countermeasures.

1.1 Related Work

Prior work has proposed verification for constant-time [10–20] and power-balancing [21] side-channel countermeasures including the Everest project which has proven the functional correctness of a side-channel resistant TLS implementation in the constant-time model [22,23].

Different approaches have recently appeared in the context of *time-balancing* countermeasures [24,25]. Blazer [24] uses a *partitioning strategy* instead of self-composition and scales to examples of up to 100 basic blocks in size. Themis [25] uses *Quantitative Cartesian Hoare Logic* to capture timing differences between executions and scales to real-world Java applications. Themis requires functions to be time-balanced; it otherwise reports spurious violations. In contrast, SideTrail takes a *path-based* approach, and can handle time-balancing countermeasures, such as those used in s2n's HMAC, which compose unbalanced functions to ensure that every execution path is time balanced. Both Blazer and Themis target Java programs while SideTrail focuses on low-level C-code implementations. Unlike these two approaches, which approximate leakage using computational complexity and bytecode instruction counts respectively, SideTrail supports fine grained cost models at the instruction level of LLVM's Intermediate Representation language.

The self-composition we describe in Sect. 3 builds on those of existing works [26–29], and our implementation follows ct-verif's [19], replacing its cross-product construction with self-composition; our novelty is in its application to time balancing. Contrary to approaches providing upper bounds of information leakage metrics [30], SideTrail employs relational verification.

2 Time-Balancing

Time-balancing countermeasures provide the security assurance that program secrets have negligible influence on execution time, even in the presence of potentially malicious observers who can control the public inputs to the program. Formally, a program is δ-*secure* if for every possible public-input value, the timing difference between every pair of executions with different secrets is at most δ. In this work, we assume a standard adversarial model: a network man in the middle (MITM), who has the ability to observe and modify both the contents and the timing of any packet on the network, but who cannot execute arbitrary code on the targeted endpoints [31]. This model is powerful enough to capture a wide range of real TLS attacks, ranging from Bleichenbacher's attacks against PKCS #1 [32], to Brumley and Boneh's attack against RSA decryption [33], to the Lucky 13 family of attacks against the TLS HMAC [1].

```
1: procedure CBC-VULNERABLE
2:     pad := packet[len − 1]
3:     payload_len := len − pad
4:     update(mac, packet, payload_len)
5:     digest(mac)
```

```
1: procedure CBC-TIMEBALANCED
2:     pad := packet[len − 1]
3:     payload_len := len − pad
4:     update(mac, packet, payload_len)
5:     update(dummyMAC, packet + payload_len, pad)
6:     digest(mac)
```

Fig. 1. A vulnerable TLS CBC algorithm (a), and its time-balanced version (b).

Example 1. The Lucky 13 family of attacks [1] takes advantage of a weakness in the specification for SSL/TLS; CBC mode uses an HMAC to prevent adversaries from modifying incoming cipher-texts, but neglects to protect the padding [34]. A MITM attacker can trick a TLS implementation into decrypting a selected encrypted byte (*e.g.* from a password) into the padding length field. Figure 1a shows what happens in a naïve implementation: in line 4, `len - pad` bytes are hashed by the HMAC, whose timing strongly depends on the number of bytes hashed. Since `len` is known to the attacker, this creates a timing side-channel leaking the value of `pad`. A constant-time mitigation would be to rewrite the HMAC implementation so that its computation was independent of the value of `pad`. A simpler time-balanced countermeasure is shown in Fig. 1b: apply `update` on a dummy HMAC state `dummyMAC` (line 5), which ensures that no matter the value of `pad`, the HMAC will always process a total of `len` bytes, mitigating the timing side-channel.

Verifying Time-Balancing. Following previous approaches to verifying non-interference [26,29], we reduce verification of δ-security, which is a 2-safety property, *i.e.* a property over pairs of executions, to the verification of a standard safety property, *i.e.* over individual executions. For technical simplicity, we consider a program P as a sequence of instructions $p_0; p_1; \ldots; p_n$ whose variables $V = Sec \uplus Pub$ are the disjoint union of secret and public variables, respectively;

a program state s maps the variables to values. A configuration c is a tuple (s, p) of program state s and the next instruction p to be executed. An execution is defined as a configurations sequence $c_1 c_2 \ldots c_m$.

Effective reasoning about δ-security requires (i) a timing model that accurately captures programs' timing behavior, and (ii) a verification technique able to relate the timing behavior of two distinct program executions. To capture timing, we introduce a leakage function $\ell(c)$, mapping configurations c to timing observations, *i.e.* the cost of executing the next program instruction using the values of variables in c. To keep track of the total cost of an execution we extend the set of variables with a *time counter* `l` as $V_L = V \uplus \{\texttt{l}\}$ and write the *time counter instrumented* program P_L as $l_1; p_1; l_2; p_2 \ldots; l_n; p_n$, in which each instruction l_i updates the time counter variable as `l:= l` $+ \ell(s, p_i)$. Finally to relate the timing cost of two execution paths we compose P_L with its renaming $\hat{P}_L$ over variables $\hat{V}_L$, and form its self-composition $P_L; \hat{P}_L$ ranging over variables $V_L \cup \hat{V}_L$ [8,26]. Accordingly, δ-security can be specified as a safety property over the time-counter variables `l` and `l̂`.

3 Implementation

SideTrail uses the SMACK verification infrastructure [35], which leverages Clang [36] and LLVM [37] to generate and optimize LLVM bitcode before translation to the Boogie [7] intermediate verification language. Using LLVM intermediate representation (IR) allows SideTrail to reason precisely about the effect of compiler optimizations, which can affect the timing of time-balancing countermeasures. Our initial experience verifying simple time-balanced examples showcased the importance of correctly accounting for compiler optimizations. In some cases, the compiler can notice that the extra code added for time-balancing

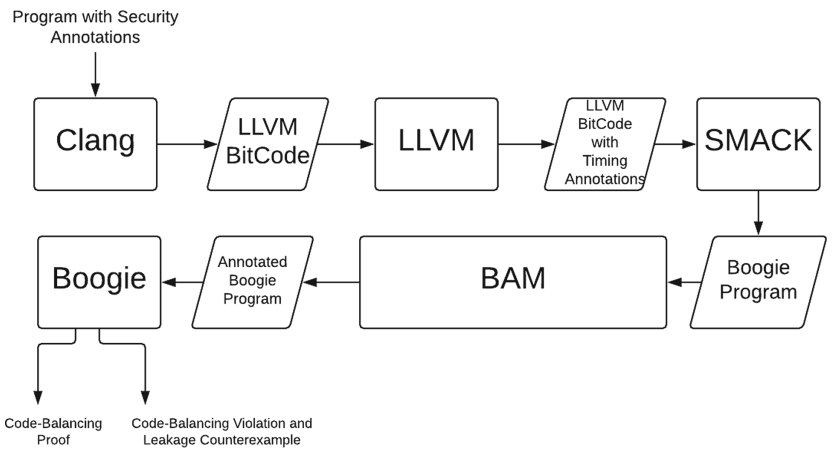

Fig. 2. SideTrail architecture

```
#define DELTA 0

int plus(
  int a, int b){
  public_in(a);
  assert_leakage(
    DELTA);
  return a + b;
}
  ...
```

```
define i32 @plus(
  i32, i32){
  %3 = add nsw i32
    %1, %0,
    !TimeCost !19
  ret i32 %3,
    !TimeCost !27
}
!19 = !{i64 1}
!27 = !{i64 0}
```

```
procedure plus(
  i0: int, i1: int)
  returns (r: int){
  var i2: int;
bb0:
  l  := l + 1;
  i2 := i0 + i1;
  l  := l + 0;
  r  := i2;
  return;
}
```

```
var l, _l: int;
procedure wrapper(
  i0: int, i1: int,
  _i0: int, _i1: int){
  assume i0 == _i0;
  call plus(i0, i1);
  call _plus(_i0, _i1);
  assume l >= _l;
  assert l - _l <= 0;
  return;
}
```

Fig. 3. Stages of SideTrail translation, from left to right: (a) an annotated C-code add function; (b) the corresponding LLVM IR with timing annotations; (c) the translated Boogie code with time counter instrumentation; and (d) the Boogie code for the self-composition.

is side-effect free, and remove it, reintroducing the timing side-channel in the original algorithm.

In addition, using LLVM IR easily allows us to use an instruction-level-precise architectural timing model. We have extended SMACK to introduce the timing cost of LLVM instructions, and implement program transformations at the Boogie code-level, passing a time counter instrumented self-composition to the Boogie verifier [7]. If the program being verified is δ-secure, SideTrail returns with a proof; if there is a violation of δ-security, SideTrail provides a counter-example trace which includes the input values that triggered the exception, the trace leading to the exception, and the amount of leakage calculated for that trace. The SideTrail flow is illustrated in Figs. 2 and 3.

Security Annotations: SideTrail requires a small number of annotations at the source-code level to specify a δ-security proof. The programmer must annotate the appropriate entry-point arguments as public (unannotated arguments are treated as secrets) and specify the non-negative integer timing-difference bound (δ). The `public_in` and `assert_leakage` annotations of Fig. 3a serve these purposes.

Timing Model: SideTrail uses LLVM's Cost Model Analysis for its instruction-level precise timing model. The analysis approximates the timing cost of each instruction when lowered to machine code by mapping it to a positive integer, and assumes that all memory accesses hit their respective caches; we discuss the soundness of this assumption in Sect. 4.3. Figure 3b shows how SideTrail annotates the LLVM IR `add` (time cost: 1) and `ret` (time cost: 0) instructions with timing metadata, represented as metadata pointers `!19` and `!27` respectively. Figure 3c shows how the timing metadata are carried over to the Boogie code. A *time-modeling* transformation, implemented as a Boogie-code transformation, introduces the integer-type time counter variable `l` and updates it in lockstep with program instructions.

Loop Invariants: To capture how the values of the time counter variables are updated throughout a loop's execution, SideTrail automatically inserts *loop-*

timing-invariants in the Boogie code, based on annotations provided by the user.

A loop's cost depends on two factors: the number of times it iterates, and the cost of each iteration. The number of iterations can be captured by annotating a simple continuation invariant—for example, loops of the form `for (i=0; i<n; ++i)` should be user-annotated with a continuation invariant `(i<=n)`. In the common case where the execution time of the loop-body does not vary across iterations, SideTrail can automatically infer the cost of each iteration. If the loop body contains nested control statements the user must provide annotations that describe how many times each branch of the control statement is visited, although we note that we encountered only a single loop with nested control in our experiments. In either case, SideTrail automatically infers timing-invariants of the form `(l = l_prior + i*body_cost)`, where `l_prior` is the value of `l` before entering the loop and `body_cost` is the timing cost of executing the loop body once, and inserts them in the Boogie code.

Self-composition: We implement the self-composition-based reduction of δ-security to assertion checking as an additional Boogie-code transformation, demonstrated in Fig. 3d. We duplicate the program to be verified, making a renamed copy of all functions (`plus` becomes `_plus`), and then transform these duplicated functions to use renamed copies of global variables including the time counter, and to perform nested procedure calls on the renamed procedures. Finally, a `wrapper` procedure enforces the equality of public inputs, makes two consecutive calls to the entry function of the program and its renamed copy, and adds an assertion to check δ-security.

Inter-procedural Analysis: SideTrail supports inter-procedural analysis through function inlining, allowing the analysis of arbitrary entry points which may invoke individually unbalanced functions (s2n uses such functions in its path-balanced HMAC). As we discuss in Sect. 4, this approach is able to handle industrial codebases such as the s2n HMAC. SideTrail also supports modular verification through timing stubs, described below.

Timing Stubs: SideTrail allows the user to specify the expected leakage from a function by providing support for *timing stubs*. Users specify these stubs by adding `assume_leakage(expr)` statements to the body of a function, where `expr` is any expression computable within the function. When the *time-modeling* transformation encounters this call, it increases the time counter variable by `expr`. This allows stubs to represent complex timing behaviour which may depend on properties of both the input and current state of the function. It is the responsibility of the user to ensure that the stub correctly models the timing behaviour of the concrete implementation; we discuss how SideTrail can be used to verify the correctness of stubs in Sect. 4.2.

4 Case Study of the s2n TLS Library

s2n is an open-source TLS library used by Amazon, including S3, and AWS services [38]. Its design goals are that it be: "small, fast, with simplicity as a priority" [39]. Its time-balanced CBC mode requires fewer than 20 additional lines of code, compared to the 500+ LOC patch to perform constant-time CBC decoding in OpenSSL. We have used SideTrail to verify the correctness of the timing countermeasures for the whole data-packet processing stack (which includes CBC verification as a sub-component) in the current s2n release. In the process we have discovered three previously unknown issues that violate δ-security. We note s2n has a belt-and-suspenders security model with randomized delay on error and a secure default configuration [40], which would have prevented these issues from affecting data in production.

The proofs described below are automatically rerun as part of s2n's Travis based continuous integration system. This ensures that code changes are only accepted to the s2n repository after they have been validated using SideTrail.

```
int s2n_hash_update(struct s2n_hash_state *state, const void *data, uint32_t size)
{
    assume_leakage(PER_BYTE_COST * size);
    state->currently_in_hash_block += size;
    int num_filled_blocks = state->currently_in_hash_block / BLOCK_SIZE;
    assume_leakage(num_filled_blocks * PER_BLOCK_COST);
    state->currently_in_hash_block = state->currently_in_hash_block % BLOCK_SIZE;
    return SUCCESS;
}
```

Fig. 4. Hash function timing stub.

4.1 Timing Stubs

s2n is written in a modular fashion, and does not implement its own cryptographic primitives – instead they are linked from the system `libcrypto`. We provide timing stubs for each cryptographic primitive used by s2n, following the approach described in Sect. 3. An example stub for `hash_update` is shown in 4. This stub has two components: a per-byte cost, representing the cost of `memcpy`ing the data, and a per-block cost, representing the cost of a hash compression round. The stubs were validated by using SideTrail to verify that the stub had the same timing behaviour as a C implementation, as described in Experiment 3 (Sect. 4.2).

4.2 Experiments

Except as noted below, all experiments had approximatly 400 lines of initial source, 5 security annotations, 3 loop invariants, 100 lines of additional code (stubs + test harness), and expanded to an order of 1000 SMT clauses. All of the experiments listed below completed in less than 8 min on a 3.1 GHz Intel Core

i7 with 16 GB of RAM running OSX 10.11, using Z3 4.6.0's integer arithmetic theory.[1] The code for all experiments is available online.[2]

For each experiment, we determined the precise amount of δ-leakage by varying δ to find the boundary where verification moved from unsuccessful to successful. Our experience in this process suggests that refuting values of δ that are too small is typically faster than verifying values of the correct size.

Properties Verified. We performed four related verification experiments using SideTrail. Experiment 1 validated our ability to detect a previously reported timing issue in the CBC mode of s2n. Experiment 2 validated the correctness of the current implementation of this code. Experiment 3 validated the correctness of the timing stubs used in Experiment 2. Experiment 4 extends this proof to provide an end to end guarantee for data-packet processing.

[Experiment 1] One of the motivations for this work is a previously reported and repaired timing side-channel issue [41] in s2n's time-balanced CBC decoder. The issue is caused by an off-by-one error in the code that tracks how many bytes have been hashed and triggers for a particular edge-case in the padding size. SideTrail reports an error trace that includes the concrete value of the padding size, as well as the expected leakage (equivalent to one hash compression round, approximately 1 µs).

[Experiment 2] In this experiment, we verify the correctness of the CBC mode time-balancing countermeasures for all protocol versions (SSL3, TLS[1.0–1.2]) and hash functions (MD5, SHA1, SHA-224, SHA-256, SHA-384, SHA-512) supported by s2n. In order to handle this wide range of modes and functions, both the test-harness and the hash-stubs were written in a generic fashion, allowing different modes to be tested by setting the appropriate compile-time constants. The experiment could then be rerun with varying parameters to cover all modes.

We experienced two types of scalability issues in this experiment. Firstly, s2n does not contain its own cryptographic primitives, leading us to use a modular verification approach using timing stubs as described in Sect. 4.1. Secondly, the CBC code makes branching decisions based on non-linear operators (`mod` and `div`), which the backend SMT solver had difficulty solving in a reasonable time. We worked around this issue by replacing the `div` operation with a handwritten variant that uses pre-calculated values for the block-sizes being tested. Our proofs for the various modes all showed a small δ-leakage, caused by the extra call to `hash_digest` necessary to affect time-balancing. Since this δ (approximately 0.03 µs) is significantly smaller than the time granularity visible to a network based attacker (estimated at 1 µs based on timing experiments done between

[1] Using integer arithmetic provides performance benefits at the cost of losing information about the underlying C types. In a few cases, we needed to annotate back this information, for example by adding `assume(x >= 0 && x < 256)` after an assignment to a `uint8_t x;`.

[2] https://github.com/danielsn/s2n/tree/sidetrail-vstte-artifact/tests/sidewinder.

co-located machines, which is in agreement with [31]), the code is successfully balanced.

[Experiment 3] The soundness of Experiment 2 depends on correct modeling of the timing behaviour of the timing stubs. In this experiment we verify that our SHA-1 timing stub accurately captures the timing cost of an open-source C implementation [42]. Our proof methodology is similar to the one used for δ-security, but in this case we form a composition of the timing stub (Fig. 4) with the C implementation and assert that their time counter variables are equal (i.e. $\delta = 0$). The SHA-1 C code contains a loop with a nested control statement, which requires us to indicate the times each branch is exercised via an additional invariant. After experimenting to find the correct values for `PER_BYTE_COST` and `PER_BLOCK_COST`, we verify that the δ between the stub and the C implementation is 0.

[Experiment 4] (450LOC Source, 150LOC stubs, 20LOC Annotations, 1860 SMT clauses) In this experiment we verify that data-packet processing is time-balanced, for all protocols (SSL3, TLS[1.0–1.2]) and modes (AEAD, CBC, Composite, and Stream) currently supported by s2n. This provides end-to-end confidence that from the time a data-packet is decrypted, until when the bytes are returned to the client, all paths are time-balanced.

The proof decomposes packet processing into three phases. Phase one, packet parsing, operates on public data, such as header fields, which are already known to a MITM attacker, and hence can be treated as public inputs. Phase two, decryption, is handled by stubs which havoc the decrypted data buffer, making the values non-deterministically different across the self-composed program executions. Phase three validates the decrypted data and returns it to the user. This phase uses the decrypted data from stage two, and hence is the stage which could potentially leak confidential information via a timing side-channel. We leverage the δ-security proof and the concrete value of δ reported from Experiment 2 to validate the timing stub for HMAC verification. All modes reported a $\delta < 0.05$ µs.

Previously Unknown Issues Discovered. Using SideTrail, we reported three previously unknown issues in the s2n time-balancing countermeasures, which have been acknowledged and fixed by the s2n team, and validated the repaired code.

[Issue 1] s2n time-balancing code counts the number of bytes in the hash block, and uses this value as part of its time-balancing countermeasures. As an optimization, s2n took advantage of the fact that the standard HMAC specification [43] pads keys to be multiples of the hash-block in length, which means that hashing a padded key does not affect the number of bytes remaining in the hash block. Unfortunately, the SSLv3 specification follows this recommendation for MD5, but in SHA1 mode uses a padded key that is 4 blocks short of the hash

block size [44], causing the time-balancing code to incorrectly count the number of bytes.

[Issue 2] s2n's HMAC had two variables with similar names: `hash_block_size` and `block_size`. Due to a typo, the wrong variable was used in a modular operation which determines the need for time-balancing operations. Interestingly, this issue only exposes itself in SSLv3 mode; in all other modes, `block_size` and `hash_block_size` have the same value. This issue was accepted and repaired by the s2n team, who also renamed the variables to have more descriptive names.

[Issue 3] As discussed in Issue 1, s2n uses a count of the number of bytes hashed as part of its time-balancing countermeasures. In addition to the bytes directly added by calls to `hash_update()`, s2n must track the number of bytes added behind the scenes by the hash algorithm. In particular, before generating a hash digest, most hash algorithms append padding, which typically includes an integer specifying the total number of bytes that have been hashed. Most hash functions used in TLS (e.g. MD5, SHA1) use an 64-bit (i.e. 8-byte) integer for this purpose, and hence s2n's time-balancing code adds 8 to the number of bytes that have been hashed when determining the need for time-balancing operations. However, some hash algorithms, such as SHA-384 and SHA-512, append a 128 bit (16-byte) integer, which would cause s2n time-balancing code to miscount the number of bytes hashed when using these algorithms.

4.3 Discussion on the Timing Model

SideTrail assumes an architectural timing model, which abstracts away micro-architectural features such as caching and branch prediction, similarly to other approaches [24,25]. This model captures the capabilities of a MITM network attacker against a TLS endpoint who can measure final execution time (with limited precision due to network jitter) but cannot affect or directly observe machine state. Consequently the attacker can only observe the cumulative number of cache hits/misses, and cannot influence them by altering the state of the cache. In the context of our s2n case study, the TLS code we verify preloads a data-packet whose maximum size is 16 KB into the cache, and then spends approximately 50 µs doing a linear scan across it (based on the speed of the hash functions used in TLS). Since the data-packet fits in the L1 cache (typically at least 32 KB), all memory accesses results in cache hits. Additionally, 50 µs are considerably less than the Linux quantum which is on the order of milliseconds, meaning that cache interference effects from context switches are minimal.

4.4 Verification Inspired Refactoring

Verifying an industrial code-base requires forming a clear understanding of the code being verified. As part of this effort, we discovered a number of refactorings that made the code more modular, clean, and easy to verify. These code changes

were shared with the s2n team as GitHub PRs, and have been merged into the mainline code-base.

The proposed changes ranged from small optimizations such as removing an unnecessary loop, to refactoring of larger portions of the code-base. For example, as part of Experiment 2, we discovered that different HMAC modes had duplicated functionality; merging this functionality into a common function simplified both the code, and the proof effort. Conversely, in Experiment 4, the data-packet processing code interleaved functionality from several different modes, requiring a large number of local variables and making it difficult to write a test harness that covered all cases. We split the code into four simpler functions, with four simple test harnesses. The s2n team accepted the PR with the comment "this looks much better, thanks." The most interesting fix was to s2n's error handling which follows a disciplined methodology. As we analysed several error handling code paths, we realized that every case followed the same template, and could be simplified with a macro that made error-handling easier to annotate (since we only needed to add the annotation in one place). As an added benefit, this change removed 400 LOC from an approximately 6 KLOC code-base.

Formal verification both inspired and enabled these changes. Our refactoring touched large portions of the overall code-base, and made changes to security-critical functionality. Without automated formal proofs to give us confidence that our changes would not introduce new timing regressions, the amount of effort to manually validate the changes would have made these changes impractical.

5 Future Work

As future work we first aim to improve the accuracy of SideTrail's timing model. Modeling the behaviour of micro-architectural components, such as the cache and the branch predictor, will extend the class of attackers that SideTrail can reason about with on-machine active attackers. Additionally, replacing LLVM's instruction cost model with a model based on micro-benchmarks will increase the precision of SideTrail, especially for operations such as `div` and `mod` which take a different number of cycles depending on the values of their input. We have demonstrated SideTrail's capability to utilize and validate timing stubs. As a second direction of future work, we envision extending the tool's usability by inferring timing stubs in an automated fashion.

6 Conclusion

Ideal cryptographic practice is to design algorithms for an easy and straightforward implementation that is naturally constant-time. For legacy algorithms that were not designed with this restriction in mind, developers must use alternate approaches such as time-balancing. SideTrail allows developers of industrial cryptographic code-bases such as s2n to verify the correctness of these mitigations, and to detect issues and regressions when they occur.

References

1. AlFardan, N.J., Paterson, K.G.: Lucky thirteen: breaking the TLS and DTLS record protocols. In: 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, 19–22 May 2013, pp. 526–540. IEEE Computer Society (2013)
2. Somorovsky, V.J.: Curious Padding oracle in OpenSSL (CVE-2016-2107) (2016). https://web-in-security.blogspot.co.uk/2016/05/curious-padding-oracle-in-openssl-cve.html. Accessed 15 Jan 2018
3. Amazon Web Services: s2n : an implementation of the TLS/SSL protocols (2018). https://github.com/awslabs/s2n
4. Dodds, J.: Part one: verifying s2n HMAC with SAW (2016). https://galois.com/blog/2016/09/verifying-s2n-hmac-with-saw/. Accessed 15 Jan 2018
5. Albrecht, M.R., Paterson, K.G.: Lucky microseconds: a timing attack on Amazon's *s2n* implementation of TLS. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9665, pp. 622–643. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49890-3_24
6. Sidewinder: Time-balanced Verification Tests (2018). https://github.com/awslabs/s2n/tree/master/tests/sidewinder
7. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
8. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: 2004 Proceedings of 17th IEEE Computer Security Foundations Workshop, pp. 100–114. IEEE (2004)
9. Amazon Web Services: s2n Travis CI integration page (2018). https://travis-ci.org/awslabs/s2n/
10. Agat, J.: Transforming out timing leaks. In: Wegman, M.N., Reps, T.W. (eds.) POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, 19–21 January 2000, pp. 40–53. ACM (2000)
11. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The program counter security model: automatic detection and removal of control-flow side channel attacks. In: Won, D.H., Kim, S. (eds.) ICISC 2005. LNCS, vol. 3935, pp. 156–168. Springer, Heidelberg (2006). https://doi.org/10.1007/11734727_14
12. Svenningsson, J., Sands, D.: Specification and verification of side channel declassification. In: Degano, P., Guttman, J.D. (eds.) FAST 2009. LNCS, vol. 5983, pp. 111–125. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12459-4_9
13. Stefan, D., et al.: Eliminating cache-based timing attacks with instruction-based scheduling. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 718–735. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40203-6_40
14. Almeida, J.B., Barbosa, M., Pinto, J.S., Vieira, B.: Formal verification of side-channel countermeasures using self-composition. Sci. Comput. Program. **78**(7), 796–812 (2013)
15. Barthe, G., Betarte, G., Campo, J.D., Luna, C.D., Pichardie, D.: System-level non-interference for constant-time cryptography. In: Ahn, G., Yung, M., Li, N. (eds.) Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 3–7 November 2014, pp. 1267–1279. ACM (2014)

16. Zhang, D., Wang, Y., Suh, G.E., Myers, A.C.: A hardware design language for timing-sensitive information-flow security. In: Özturk, Ö., Ebcioglu, K., Dwarkadas, S. (eds.) Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, 14–18 March 2015, pp. 503–516. ACM (2015)
17. Doychev, G., Köpf, B., Mauborgne, L., Reineke, J.: Cacheaudit: a tool for the static analysis of cache side channels. ACM Trans. Inf. Syst. Secur. **18**(1), 4:1–4:32 (2015)
18. Rodrigues, B., Pereira, F.M.Q., Aranha, D.F.: Sparse representation of implicit flows with applications to side-channel detection. In Zaks, A., Hermenegildo, M.V. (eds.) Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, 12–18 March 2016, pp. 110–120. ACM (2016)
19. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M.: Verifying constant-time implementations. In: 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, 10–12 August 2016, pp. 53–70 (2016)
20. Blazy, S., Pichardie, D., Trieu, A.: Verifying constant-time implementations by abstract interpretation. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) ESORICS 2017. LNCS, vol. 10492, pp. 260–277. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66402-6_16
21. Fang, X., Luo, P., Fei, Y., Leeser, M.: Leakage evaluation on power balance countermeasure against side-channel attack on FPGAs. In: 2015 IEEE High Performance Extreme Computing Conference, HPEC 2015, Waltham, MA, USA, 15–17 September 2015, pp. 1–6. IEEE (2015)
22. Bond, B., et al.: Vale: verifying high-performance cryptographic assembly code. In: 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, 16–18 August 2017, pp. 917–934 (2017)
23. Bhargavan, K., et al.: Everest: towards a verified, drop-in replacement of HTTPS. In: Lerner, B.S., Bodík, R., Krishnamurthi, S. (eds.) 2nd Summit on Advances in Programming Languages, SNAPL 2017, Volume 71 of LIPIcs., Asilomar, CA, USA, 7–10 May 2017, pp. 1:1–1:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
24. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, pp. 362–375. ACM, New York (2017)
25. Chen, J., Feng, Y., Dillig, I.: Precise detection of side-channel vulnerabilities using quantitative Cartesian hoare logic. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, pp. 875–890. ACM, New York (2017)
26. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005). https://doi.org/10.1007/11547662_24
27. Zaks, A., Pnueli, A.: CoVaC: compiler validation by program analysis of the cross-product. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 35–51. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_5
28. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_17

29. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Math. Struct. Comput. Sci. **21**(6), 1207–1252 (2011)
30. Pasareanu, C.S., Phan, Q.S., Malacaria, P.: Multi-run side-channel analysis using symbolic execution and max-SMT. In: 2016 IEEE 29th Conference on Computer Security Foundations Symposium (CSF), pp. 387–400. IEEE (2016)
31. Crosby, S.A., Wallach, D.S., Riedi, R.H.: Opportunities and limits of remote timing attacks. ACM Trans. Inf. Syst. Secur. **12**(3), 17:1–17:29 (2009)
32. Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 1–12. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055716
33. Brumley, D., Boneh, D.: Remote timing attacks are practical. In: Proceedings of the 12th Conference on USENIX Security Symposium, SSYM 2003, vol. 12, p. 1. USENIX Association, Berkeley (2003)
34. Rescorla, E., Dierks, T.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008
35. Rakamarić, Z., Emmi, M.: SMACK: decoupling source language details from verifier implementations. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 106–113. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_7
36. LLVM: clang: a C language family frontend for LLVM (2018). https://clang.llvm.org/
37. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO 2004), Palo Alto, California, March 2004
38. Schmidt, S.: s2n is now handling 100 percent of SSL traffic for Amazon S3 (2017). https://aws.amazon.com/blogs/security/s2n-is-now-handling-100-percent-of-of-ssl-traffic-for-amazon-s3/. Accessed 15 Jan 2018
39. Schmidt, S.: Introducing s2n, a new open source TLS implementation (2015). https://aws.amazon.com/blogs/security/introducing-s2n-a-new-open-source-tls-implementation/. Accessed 15 Jan 2018
40. MacCarthaigh, C.: s2n and Lucky 13 (2015). https://aws.amazon.com/blogs/security/s2n-and-lucky-13/. Accessed 15 Jan 2018
41. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F.: Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC. In: Peyrin, T. (ed.) FSE 2016. LNCS, vol. 9783, pp. 163–184. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-52993-5_9
42. Brad Conte: Basic implementations of standard cryptography algorithms, like AES and SHA-1 (2018). https://github.com/B-Con/crypto-algorithms. Commit: 02b66ec38b474445d10a5d1f0114bc0e8326707e
43. Krawczyk, H., Bellare, M., Canetti, R.: HMAC: Keyed-Hashing for Message Authentication. RFC 2104, RFC Editor, February 1997. http://www.rfc-editor.org/rfc/rfc2104.txt
44. Freier, A., Karlton, P., Kocher, P.: The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101, RFC Editor, August 2011. http://www.rfc-editor.org/rfc/rfc6101.txt