2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)

ct-fuzz: Fuzzing for Timing Leaks

Shaobo He
University of Utah, USA
shaobo@cs.utah.edu

Michael Emmi
SRI International, USA
michael.emmi@gmail.com

Gabriela Ciocarlie
SRI International, USA
gabriela.ciocarlie@sri.com

***Abstract*—Testing-based methodologies like fuzzing are able to analyze complex software which is not amenable to traditional formal approaches like verification, model checking, and abstract interpretation. Despite enormous success at exposing countless security vulnerabilities in many popular software projects, applications of testing-based approaches have mainly targeted checking traditional safety properties like memory safety. While unquestionably important, this class of properties does not precisely characterize other important security aspects such as information leakage, e.g., through side channels.**

In this work we extend testing-based software analysis methodologies to two-safety properties, which enables the precise discovery of information leaks in complex software. In particular, we present the ct-fuzz tool, which lends coverage-guided greybox fuzzers the ability to detect two-safety property violations. Our approach is capable of exposing violations to any two-safety property expressed as equality between two program traces. Empirically, we demonstrate that ct-fuzz swiftly reveals timing leaks in popular cryptographic implementations.

I. Introduction

Security is a primary concern for software systems, where errors like out-of-bounds memory accesses and inexhaustive input validation are responsible for dangerous and costly incidents. Accordingly, many mechanisms exist for protecting systems against common vulnerabilities like memory-safety errors and input injection. Among the most effective automated approaches is *coverage-based greybox fuzzing* [1] popularized by afl-fuzz,[1] which has uncovered many critical vulnerabilities. Its efficacy is due to broad applicability and direct feedback: AFL's genetic input-generation algorithm uses lightweight instrumentation to determine vulnerability-witnessing inputs. It works without user interaction and even source code, and sidesteps the computational and methodological bottlenecks imposed by traditional program analysis techniques.

Fuzzers like afl-fuzz target traditional temporal safety properties [2], i.e., properties concerning individual system executions. Important security aspects like secure information flow [3] are *two-safety properties* [4], i.e., properties concerning *pairs* of executions, and cannot be precisely characterized as traditional safety properties [5]. Secure information flow is particularly insidious given the potential for *side channels*, through which adversaries may infer privileged information about the data accessed by a program, e.g., by correlating execution-time differences with control-flow differences. Side-channels are difficult for programmers to reason about since they are generally hardware-dependent. Furthermore, while compiler optimizations are obliged to respect traditional safety properties, they are not designed to respect two-safety properties, and can thus generate insecure machine code.

In this work we extend fuzzing to two-safety properties. In particular, we present ct-fuzz, [2] which enables the precise discovery of timing-based information leaks while retaining the efficacy of afl-fuzz. Via *self-composition* [6], we reduce testing two-safety properties to testing traditional safety properties by program transformation, effectively enabling the application of any fuzzer. Our implementation tackles three basic challenges. First, the program under test must efficiently simulate execution-pairs of the original; to avoid overhead, each execution's address space is copy-on-write shared. Second, structured input-pairs must be derived from random fuzzer-provided input; leaks are only witnessed when inputs differ solely by secret content. Third, leakage-inducing actions must be monitored; leaks are witnessed when, e.g., control flow or memory-access traces diverge, given inputs differing solely by secret content.

Our implementation focuses on timing leaks related to program control-flow and CPU cache; the name ct-fuzz refers to the *constant-time* property, asserting that the control-flow and accessed memory locations of two executions differing solely by secrets are identical. Besides constant-time, we also apply ct-fuzz to finer-grained cache models to validate secure yet non-constant-time programs which ensure identical cache timing despite potentially-divergent memory-access patterns, e.g., by *preloading* all accessed memory into the cache. While our implementation focuses on side-channel leakage related to timing, it is extensible to other forms of leakage, e.g., power or electro-magnetic radiation, by further extending the instrumentation mechanism to record other aspects of program behavior. In principle, ct-fuzz could be extended to expose violations to any two-safety property expressible as equality between two program traces.

In the remainder we describe several aspects of the ct-fuzz tool. Sections II–III cover foundations and implementation concerns in testing two-safety properties. Sections IV–V cover ct-fuzz's software architecture and basic functionality. Sections VI–VII describe case studies and evaluation, demonstrating the application of ct-fuzz to many cryptographic libraries. We outline future work in Section VIII.

[1] American Fuzzy Lop: http://lcamtuf.coredump.cx/afl/

[2] ct-fuzz is on GitHub: https://github.com/michael-emmi/ct-fuzz, as are experiments: https://github.com/shaobo-he/ct-fuzz-artifact/releases/tag/icst2020.

978-1-7281-5778-8/20/$31.00 ©2020 IEEE
DOI 10.1109/ICST46399.2020.00063

II. Theoretical Foundations

In this section we characterize ct-fuzz's foundations, including secure information flow, its reduction to traditional safety properties, and coverage-guided greybox fuzzing.

A. Secure Information Flow

We consider an abstract notion of secure information flow [3] over pairs of executions. Two executions e_1 and e_2 are *f-equivalent* with respect to some function f when $f(e_1) = f(e_2)$. We model program secrets by a *declassification function* D mapping each execution e to its declassified content $D(e)$, and say D-equivalent executions are *comparable*. Intuitively, this equivalence relates executions whose input and output values differ solely by secret content, e.g., by the value of a secret key. Similarly, we model attacker capabilities by an *observation function* O mapping each execution e to its observable content $O(e)$, and say two O-equivalent executions are *indistinguishable*. Intuitively, this equivalence relates executions which a given observer cannot differentiate, e.g., because of negligible timing differences. In practice, we distinguish timing variations by observing divergences in control-flow decisions and accessed memory locations.

Definition 1. A program is *secure* when every pair of comparable executions are indistinguishable, for given declassification and observation functions.

B. Reducing Security to Safety

Self-composition [6] is program transformation reducing secure information flow to traditional safety properties. Here we say a given program is *safe* when it cannot crash. This simple notion of safety can capture violations to any temporal safety property [2] given adequate program instrumentation. For instance, LLVM's thread, address, and memory sanitizers signal crashes upon thread- and memory-safety violations, and uses of uninitialized memory, respectively

The *self-composition* $\mathrm{sc}(P)$ of program P is a program (Fig. 1) which executes two copies of P in isolation, i.e., execution of each copy is independent of the other, which:

- halts when the copies' executions are incomparable, and
- crashes when the copies' executions are both comparable and distinguishable.

Intuitively, safety of the self-composition implies security of the original program. Conversely, if the original is safe and secure, then the self-composition is also safe.

Theorem 1. A safe program P is secure iff $\mathrm{sc}(P)$ is safe [6].

C. Coverage-Guided Greybox Fuzzing

Böhme et al. [1] overview *coverage-guided greybox fuzzing* as implemented by afl-fuzz. For our purposes, a simplistic view (Fig. 2) suffices: fuzzers explore sequences of program executions to expose safety-property violations (manifested as crashes — see §II-B) by randomly mutating the inputs to previously-explored executions according to per-execution feedback. Intuitively, feedback allows the fuzzer to navigate alternate program paths, and in practice, captures basic-blocks transition counts. From the perspective of this work, the salient aspects of afl-fuzz are its broad applicability and efficiency: it works without user interaction and source code, and avoids prohibitive process-creation overheads by sharing executions' address spaces in a copy-on-write fashion. These features enable the rapid exploration of program executions for arbitrarily-complex software.

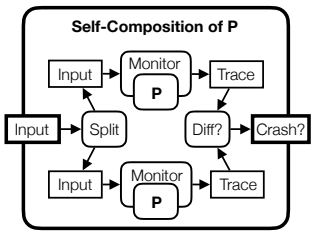

Fig. 1. Self-composition simulates two executions of a given program.

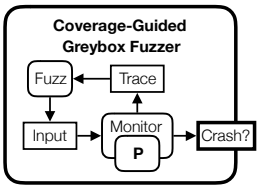

Fig. 2. Coverage-guided fuzzers generate inputs and report crashes.

III. Design and Implementation Concerns

Applying self-composition (§II-B) to coverage-guided greybox fuzzers (§II-C) poses three basic challenges: efficiently simulating execution pairs; deriving structured-input pairs from fuzzer-provided inputs (fuzz); and detecting leakage.

A. Simulating Execution Pairs

Self-composition (§II-B) dictates that two identical copies of a program execute in isolation. In the context of testing, achieving isolation includes separating the address spaces of each simulated execution so that the side-effects of one are invisible to the other. On the one hand, the existing approach of duplicating procedures and variables is effective for symbolic analyses like ct-verif [7]. However, actually executing such duplicated programs on their target platforms can alter the originals' behavior significantly, e.g., due to platform-dependent behavior. On the other hand, executing copies in separate processes undermines the efficacy of fuzzers like afl-fuzz which avoid process-creation overhead.

B. Deriving Structured Inputs from Fuzz

According to secure information flow (§II-A), leaks are only witnessed by comparable executions, i.e., whose declassifications are identical. So on the one hand, testing incomparable executions is wasteful. On the other hand, coverage-guided greybox fuzzers (§II-C) provide inputs (fuzz) by randomly mutating previously-explored inputs. A self-composition which fed fuzz directly to its simulated executions would generate comparable executions with very low likelihood. For instance, when input variable x is declassified in the program $P(x, y)$, the likelihood of randomly generating the fuzz $\langle x_1, y_1, x_2, y_2 \rangle$ with $x_1 = x_2$ for comparable $P(x_1, y_1)$ and $P(x_2, y_2)$ is low for typical datatypes, e.g., 32- and 64-bit integers. It follows that exposing information leaks requires non-trivial transformations from fuzz to structured input pairs.

```
void ct_fuzz_main(void) {
  int status;
  ct_fuzz_initialize();
  ct_fuzz_read_inputs();

  // ensure preconditions
  for (int id = 0; id < 2; ++id)
    ct_fuzz_spec(id);

  // execute each copy
  for (int id = 0; id < 2; ++id) {
    pid_t pid = fork();
    if (pid == -1)
      exit(EXIT_FAILURE);
    else if (pid == 0) {
      ct_fuzz_exec(id);
      exit(EXIT_SUCCESS);
    } else
      waitpid(pid, &status, 0);
  }

  ct_fuzz_check_observations();
}
```

Fig. 3. A fork-based self-composition.

```
// captures observations
// for branch instructions
void ct_fuzz_update_monitor_by_cond(
  bool condition_value,
  char* file_name,
  num_t line_number,
  num_t column_number
);

// captures observations
// for memory accesses
void ct_fuzz_update_monitor_by_addr(
  char* address,
  char* file_name,
  num_t line_number,
  num_t column_number
);
```

Fig. 4. API for observations.

C. Detecting Leakage

According to secure information flow (§II-A), leaks are witnessed by comparable yet distinguishable executions, i.e., whose declassifications are identical, yet observations are distinct. Assuming a reliable mechanism for generating comparable executions (§III-B), monitoring leakage amounts to establishing a notion of observation for a given leakage model, instrumenting the source program to record such observations, and signaling a program crash when observations differ.

IV. Software Architecture

We implement self-composition by a lightweight LLVM program transformation. We invoke it as part of afl-fuzz's LLVM-based instrumentation for convenience, before passing the instrumented program to afl-fuzz. The choice of an LLVM-based implementation was made for familiarity and convenience, e.g., due to LLVM bitcode being typed; in principle, our approach could be implemented at the assembly level. Following the concerns outlined in Section III, our transformation provides three basic capabilities: efficiently simulating execution pairs, deriving structured inputs-pairs from fuzz, and capturing leakage-relevant observations.

A. Efficient Implementation of Self-Composition

To implement self-composition, ct-fuzz borrows the same basic trick that makes afl-fuzz efficient: copy-on-write sharing of address spaces via process forks. Figure 3 sketches our implementation strategy. After initializing the data structures used to capture observations (§IV-C), constructing inputs from fuzz (§IV-B), and ensuring the preconditions of the program under test, ct-fuzz forks the running process twice. After each child executes the original program on its copy of input, the parent checks equality of the children's observation traces. Section V-A describes the specification of programs, including stating their declassifications and preconditions.

Our fork-based self-composition avoids a potentially-complex address-space management entailed by the existing *duplication* approach (see §III-A). Forking the existing process creates children with identical address spaces, thus minimizing the potential for artificial divergence. Furthermore, the cost of forking is low on modern operating systems due to *copy-on-write* optimization: the virtual-memory pages of child processes point to the same physical pages as their parents; pages themselves are only duplicated when either the parent or child dirties them with subsequent writes. While forking provides ample isolation for basic CPU-driven programs like cryptographic primitives, we do not ensure isolation with stateful IO, e.g., interacting with files and sockets; this is a common issue for testing-based approaches.

Our implementation of self-composition also assumes that library functions are deterministic: a sequence of invocations returns identical values in both forked children. Nondeterminism can undermine ct-fuzz's leakage tests, since divergence between execution pairs may be due to nondeterminism rather than dependence on secrets. This potential is especially apparent in memory allocation: malloc is generally free to return the address of any unallocated chunk of memory. ct-fuzz handles this common case by linking the jemalloc allocator which ensures deterministic behavior.

B. Deriving Inputs from Fuzz

To transform the randomly-mutated inputs (fuzz) into program inputs which are likely to generate comparable executions (see §III-B), ct-fuzz generates per-program input processors. These processors depend on the signatures of entry points, as well as *declassification annotations* on program arguments. Besides the program under test, ct-fuzz expects such signatures and annotations to be specified using the API described in Section V. Given this specification, ct-fuzz constructs an input processor, which constructs program-input pairs by reading (from standard input) one fuzzed instance of each declassified argument, and two fuzzed instances of each secret argument. ct-fuzz invokes both program copies with the same fuzz for declassified arguments, and possibly distinct fuzz for secret arguments. While this mechanism does not guarantee that the corresponding executions are ultimately comparable, since further declassification can occur upon execution, e.g., declassification of output values, it does avoid incomparability due to declassified inputs.[3]

C. Recording Observations and Reporting Leakage

To capture alternate leakage models ct-fuzz provides an extensible mechanism for recording observations. As our initial implementation targets control-flow and cache-based timing leaks, our program transformation inserts instrumentation before branch and memory-access instructions; more precisely, we insert calls to the monitor API of Figure 4. The monitor receives branch-condition values and memory addresses, along with the source location of each instruction, and records observations in a shared memory region for access by parent process. When observations diverge, ct-fuzz signals leakage

[3]Our implementation can be extended to handle post-input declassification by monitoring outputs, similarly to monitoring observations — see §IV-C.

```
// encrypt.c
const char book[10] __attribute__((aligned(64)))
  = { 52, 48, 55, 51, 56, 54, 50, 49, 57, 53 };

void encrypt(char* msg, unsigned len) {
  for (unsigned i = 0; i < len; ++i)
    msg[i] = book[msg[i]-48];
}

// ct-fuzz specification of `encrypt`
#include "ct-fuzz.h"

CT_FUZZ_SPEC(void, encrypt, char* msg, unsigned len) {

  // `msg` is a buffer of length `len` at most `4`
  __ct_fuzz_ptr_len(msg, len, 4);

  // declassification of the `len` argument
  __ct_fuzz_public_in(len);

  // precondition that `msg` contains ASCII decimals
  for (unsigned i = 0; i < len; ++i)
    CT_FUZZ_ASSUME(msg[i]>=48 && msg[i]<=57);
}

CT_FUZZ_SEED(void, encrypt, char*, unsigned) {
  SEED_1D_ARR(char, msg, 4, {'1','2','3','4'})
  PRODUCE(encrypt, msg, 4);
}
```

Fig. 5. A (cryptographically-insecure) lookup-table based encryption function.

by inducing a crash which will be reported by the fuzzer; our current implementation causes a segmentation fault by dereferencing address zero.

Our prototype provides two implementations of the monitor API. The first *constant-time monitor* collects traces of branch-condition values and memory addresses directly; executions are distinguishable when they differ on a branch-condition value or memory address. The second *cache-model monitor* records Boolean values indicating cache hits or misses in place of memory accesses, allowing for the precise analysis of non-constant-time programs which are nevertheless safe for a given cache architecture, e.g., cache-preloading implementations (see §VI). Our prototype uses Sung et al.'s cache model [8], though any could be used in its place.

To limit shared-memory region size and equality-checking time, we leverage the xxHash fast hash-function to store one fixed-sized value for arbitrarily-long observation sequences. For each observation o, the monitor updates its hash value m to $f(o \cdot m)$ for a given hash function f. We state the correctness of this optimization as follows, where a *perfect hash function* f is one in which $f(x) = f(y)$ iff $x = y$.

Theorem 2. Given a perfect hash function, two observation sequences are equal iff their lengths and monitors are equal.

V. Functionality and Capabilities

Our ct-fuzz tool works on any code analyzable with afl-fuzz's LLVM mode. In this section we demonstrate ct-fuzz on the simple cryptographic function in Figure 5.

A. Specifying Preconditions, Declassifications, and Defaults

Our current implementation requires a specification of program arguments: preconditions, declassifications, and default values. Similarly to ct-verif [7], annotations in ct-fuzz are written directly in the source language (i.e., C/C++), and processed by our instrumentation at compile time. Specifications are attached to entry points using the `CT_FUZZ_SPEC` and `CT_FUZZ_SEED` macros. The specification in Figure 5 declassifies the `len` argument (`msg` is secret), requires that each byte of the `msg` buffer be an ASCII-encoded decimal digit, and provides a default buffer containing bytes `'1'`–`'4'`.

For fuzzing of dynamically-sized buffers like `msg`, the specification also declares that `msg` is a buffer of length `len` at most 4 bytes; constant-length buffers can also be specified. Default values provide the initial seed values from which fuzzers begin their mutation-based exploration. While not strictly required, reasonable seeds can boost performance substantially. We provide macros for specifying default values, e.g., `SEED_1D_ARR`. Similarly, while preconditions are not strictly required, they allow us to isolate leakage-related crashes from safety-related crashes by assuming that the original program is safe, and thus does not crash, so long as the preconditions are met. The provided `CT_FUZZ_ASSUME` macro triggers exit unless its argument evaluates to true. This mechanism leverages the fuzzer's ability to recognize branches and synthesize inputs which pass precondition checks.

B. Exposure and Diagnosis of Timing Leaks

ct-fuzz applies the self-composition described in §IV on programs with specifications, generating a binary executable and initial seed for fuzzing. The invocation of ct-fuzz:

```
$ ct-fuzz --entry-point=encrypt encrypt.c -o encrypt
```

generates the `encrypt` binary and seed `0x31 0x32 0x33 0x34 0x04 0x00 0x00 0x00`, according to the specification of default argument values. After copying the generated seed to `input-dir`, afl-fuzz can be invoked directly:

```
$ afl-fuzz -i input-dir -o output-dir encrypt
```

which instantly reports crashes indicating leaks, according to constant-time. This is expected, since the memory locations accessed by the encrypt function depend on the secret contents of `msg`: secrets are used as offsets into the `book` buffer. Such secret-dependent memory accesses can lead to timing variations, and ultimately the leakage of `msg` contents.

To facilitate diagnostics, ct-fuzz includes a mechanism for logging and comparing the observations of the comparable yet distinguishable execution pairs, tracing observations and the source-file locations at which they occur. For example, comparing the traces generated for the leak exposed above:

```
[dbg] [0] [encrypt.c: 7, 14] [address, 403389]
...
[dbg] [1] [encrypt.c: 7, 14] [address, 403381]
```

we spot the divergence due to the dereference of the `book` buffer on Line 7, column 14.

C. Using Alternative Leakage Models

For some applications, the constant-time is too conservative. For example, since the `book` buffer in Fig 5 fits into a single cache line (with typical assumptions), the contents of `msg`

```
const uint8_t SE[256] = { 0x63, ... };

inline uint32_t SE_word(uint32_t x) {
  return make_uint32( SE[get_byte(0,x)],
      SE[get_byte(1,x)], SE[get_byte(2,x)], SE[get_byte(3,x)] );
}
```

Fig. 6. Secret-dependent array access in Botan AES encryption.

would not affect timing, since every access to book after the first hits the cache, independently of msg. Alternate leakage models can be selected in ct-fuzz with the `--memory-leakage` flag; currently we support two options: `address` and `cache`. The latter implements Sung et al.'s model [8] with fixed values for block size, associativity, and replacement policy. Extension to parametric and alternative models is straightforward. Using the cache model, our example is leak free.

VI. Experience and Case Studies

We have applied ct-fuzz to several popular cryptographic implementations — see §VII. To highlight one case, we consider the AES encryption functions of the Botan library. For simplicity, we consider potential leakage of a secret 16-byte key argument, fixing all other parameters.

A. Analysis of Constant-Time

Applying ct-fuzz immediately reveals a constant-time violation, witnessed by a pair of inputs differing only by their first byte: `0xaa` versus `0x2a`. Comparing their execution traces reveals leakage from the `SE_word` function shown in Figure 6: the single-byte input difference leads to different offsets into the `SE` table computed by `get_byte(3,x)`.

B. Precise Cache Modeling

Although Botan's AES implementation is not constant time, it is still considered secure against timing leaks due to its use of *cache preloading* countermeasures. Specifically, every entry in its lookup tables, e.g., the `SE` table, is accessed before the secret-dependent lookup-table accesses performed during encryption or decryption to ensure that subsequent secret-dependent accesses hit the cache. Previous versions neglected preloading in some functions, and were found insecure [9].

Applying ct-fuzz to insecure versions reveals a timing leak even with the cache-model monitor. Interestingly, the inputs for the first-reported leak are identical to the constant-time violation: one execution's access of `SE[get_byte(3,x)]` misses the cache, while the other's hits. The divergence arises because the addresses accessed in Fig. 6 are proximate in one execution, while the fourth is distant from the previous three in the other, placing them in different cache lines. Reapplying ct-fuzz to the secure version reveals no leaks.

VII. Empirical Evaluation

We evaluate ct-fuzz's ability to uncover timing leaks in a range of cryptographic implementations. We analyze multiple entry points from each with the constant-time and cache-model monitors described in Section IV. Besides the benchmarks used to evaluate SC-Eliminator [9], we have collected several libraries from their sources. Tables I and II summarize our results.[4] We run afl-fuzz 10 times with 10 second timeouts for each entry point. When crashes are reported, we re-run afl-fuzz 100 times with the same timeout, and report the mean and standard deviations until crash counting: time (in seconds), executions, and program paths explored. Otherwise, we re-run afl-fuzz 2 times for 100 seconds, and report the same types of measurements except time.

TABLE I
Wu et al.'s benchmarks [9] with ct-fuzz. Mean and standard deviation are reported for: time (seconds), executions, and path counts.

	Constant Time			Cache Model		
Function	Time	Execs	Paths	Time	Execs	Paths
Botan: https://github.com/randombit/botan/						
aes_key	1.1±0.07	69±0.87	1±0	2.1±0.19	61±2.3	1±0
cast128	0.35±0.08	140±9.3	1±0	0.38±0.07	74±4.2	1±0
des	0.36±0.07	190±4.8	1±0	0.36±0.07	100±7.4	1±0
kasumi	0.36±0.08	170±5.1	1±0	0.37±0.07	120±7.3	1±0
seed	0.36±0.07	160±9.8	1±0	0.37±0.08	99±8.3	1±0
twofish	0.6±0.07	88±3.1	1±0	0.59±0.07	69±1.4	1±0
ChronOS: http://www.chronoslinux.org						
aes	0.36±0.07	160±6.5	1±0	0.58±0.07	140±6.5	1±0
anubis	0.35±0.09	170±7.4	1±0	0.58±0.08	160±17	1±0
cast5	0.77±0.08	650±36	1±0	1.5±0.22	610±43	1±0
cast6	0.36±0.07	220±21	1±0	0.36±0.09	120±14	1±0
des	0.36±0.12	210±8	1±0	0.37±0.09	130±13	1±0
des3	0.36±0.07	180±12	1±0	0.36±0.08	110±8.9	2.6±0.6
fcrypt	0.36±0.07	280±21	1±0	0.37±0.07	200±17	1±0
khazad	0.34±0.07	270±20	1±0	0.37±0.1	130±9.4	1±0
Applied Cryptography Textbook: Schneier, 1996						
loki91	0.57±0.08	18±0.79	1±0	0.91±0.10	21±1.2	1±0
3way	0.35±0.07	190±3.6	1±0	0.36±0.07	150±18	1±0
FELICS: https://www.cryptolux.org/index.php/FELICS						
LBlock	0.36±0.07	150±5.2	1±0	–	5.4e4±890	1±0
Piccolo	0.36±0.08	130±9.6	1±0	–	4.4e4±1e3	1±0
PRESENT	0.37±0.07	140±7.6	1±0	–	4.3e4±65	1±0
TWINE	0.37±0.07	88±2.4	1±0	–	2.9e4±280	1±0
Libgcrypt: https://gnupg.org/software/libgcrypt/index.html						
camellia	0.35±0.07	270±32	1±0	0.36±0.08	190±27	1±0
des	0.35±0.07	210±19	1±0	0.35±0.08	150±19	1±0
seed	0.34±0.07	230±40	1±0	0.37±0.07	140±22	1±0
twofish	0.37±0.07	87±4.3	1±0	0.58±0.15	53±12	1±0
SUPERCOP: https://bench.cr.yp.to/supercop.html						
aes	0.35±0.08	290±3.2	1±0	0.36±0.07	220±5.8	1±0
cast	0.35±0.07	230±6.1	1±0	0.36±0.07	100±2.8	1±0

A. Analysis with the Constant-Time Monitor

ct-fuzz swiftly reports constant-time violations due to secret-dependent table lookups, e.g., all of Wu et al.'s benchmarks. OpenSSL's C implementation of AES encryption leverages substitution boxes. For supposedly constant-time implementations such as BearSSL's constant-time AES encryption (`aes_ct`) and libsodium's constant-time utility functions (`sodium_increment`, `sodium_is_zero`), ct-fuzz reports no violations. In our experience, running these benchmarks for several hours, all violations are first reported within seconds: within half of a second on average, exploring only hundreds of executions. Execution counts are approximate, since afl-fuzz's

[4] We run a 3.5GHz Intel i7 Ubuntu 16.04 desktop with 16GB DDR3 memory.

TABLE II
ANALYSIS OF OPEN-SOURCE CRYPTO IMPLEMENTATIONS WITH CT-FUZZ. MEAN AND STANDARD DEVIATION ARE REPORTED FOR: TIME (SECONDS), EXECUTIONS, AND PATH COUNTS. DECLASSIFIED INPUTS LISTED IN BOLDFACE.

Constant Time			Cache Model		
Time	Execs	Paths	Time	Execs	Paths
BearSSL: https://bearssl.org					
`aes_ct(iv,data,`**`data_len`**`,key,`**`key_len`**`)`					
–	9.4e4±3.6e3	22±0	–	9.3e4±4.4e3	30±2.8
`aes_small(`**`iv`**`,data,`**`data_len`**`,key,`**`key_len`**`)`					
0.35±0.071	1.7e2±0.2	1±0	1.5±0.13	7.5e2±51	15±6.8
`md5(data,`**`len`**`)`					
–	1.3e5±3.9e3	2±0	–	1.3e5±2.2e3	2±0
libsodium: https://download.libsodium.org/doc					
`crypto_chacha20poly1305_encrypt(c,clen,m,`**`mlen`**`,`**`ad`**`,`**`adlen`**`,`**`npub`**`,k)`					
–	4.2e4±1.5e3	31±0	–	3.5e4±3.6e2	46±2.1
`crypto_shorthash(out,in,`**`inlen`**`,k)`					
–	1.3e5±5.5e3	11±0	–	1.2e5±8.8e2	13±1.4
`sodium_increment(n,`**`nlen`**`)`					
–	1.4e5±7.2e3	12±0	–	1.4e5±8.8e2	16±0.7
`sodium_is_zero(n,`**`nlen`**`)`					
–	1.4e5±3.4e3	8±0	–	1.4e5±4.4e3	12±2.1
OpenSSL: https://www.openssl.org					
`EVP_aes_128_cbc(key,data,`**`iv`**`)`					
0.36±0.08	2.5e2±7.7	1±0	0.34±0.07	2e2±1.6	1±0
`ssl3_cbc_copy_mac(`**`orig_len_raw`**`,length)`					
–	2.8e5±5.3e3	7.5±0.7	–	2.7e5±1.1e3	8.5±0.7
`ssl3_cbc_copy_mac_modulo(`**`orig_len_raw`**`,length)`					
–	2.6e5±8.9e3	8.5±0.7	–	2.7e5±3.7e3	10±0
poly1305-donna: https://github.com/floodyberry/poly1305-donna					
`poly1305_auth(mac,m,`**`bytes`**`,key)`					
–	1.3e5±1.8e3	15±0	–	1.4e5±2.8e3	24±0.7
poly1305-opt: https://github.com/floodyberry/poly1305-opt					
`poly1305_auth(mac,in,`**`inlen`**`,key)`					
–	2.6e4±2.2e2	1±0	–	2.4e4±4.4e2	2±1.4
s2n: https://github.com/awslabs/s2n					
`s2n_hmac_digest(sekrit,`**`sekritlen`**`,msg,`**`msglen`**`)`					
–	9.3e4±8.7e2	4±0	–	8.2e4±2.5e3	4.5±0.7

`AFL_BENCH_UNTIL_CRASH` mode only guarantees termination *soon* after the first crash. Furthermore, path counts report the number of unique non-crashing control-flow paths discovered by afl-fuzz; paths are often unique since cryptographic implementations tend to use fairly straight-line code.

B. Analysis with the Cache-Model Monitor

We also run ct-fuzz with our cache-model monitor (§V-C). The cache model is identical to that used in the evaluation of SC-Eliminator [9]: a fully associative LRU cache of 512 64-byte lines. We report no timing leaks for Wu et al.'s FELICS benchmarks [9] which is expected since their lookup tables fit into one cache line. This is consistent with SC-Eliminator's static analysis, although our notion of leakage, framed as a two-safety property, is a more-precise and avoids false positives. Furthermore, we report zero leaks for constant-time functions like `aes_ct`, which is expected since constant-time is stricter. Differences in executions explored until crash are due to the approximation: since the constant-time monitor is more efficient, afl-fuzz squeezes in many more executions between the first crash and termination.

VIII. Conclusion and Future Work

This work demonstrates that testing-based software analysis methodologies can be effectively extended from safety properties to two-safety properties. While our initial application has targeted timing leaks in cryptographic implementations, there are several promising directions for future investigation. Since cryptographic code tends to follow rather simple straight-line control flow, coverage-guided greybox fuzzers offer relatively little over more simplistic random testing; exposing leaks in more complex input-processing applications such as JPEG encoders and decoders could better exploit two-safety-property fuzzers. Furthermore, ct-fuzz could be applied to other types of leakage, and even other classes of two-safety properties, by extending the instrumentation mechanism; in principle, our approach is capable of exposing violations to any two-safety property expressed as trace equalities.

Acknowledgements

This work was funded in part by the US Department of Homeland Security (DHS) Science and Technology (S&T) Directorate under contract no. HSHQDC-16-C-00034. The views and conclusions contained herein are the authors' and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DHS or the US government.

References

[1] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *ACM Conference on Computer and Communications Security*. ACM, 2016, pp. 1032–1043.

[2] Z. Manna and A. Pnueli, *Temporal verification of reactive systems - safety*. Springer, 1995.

[3] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[4] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *SAS*, ser. Lecture Notes in Computer Science, vol. 3672. Springer, 2005, pp. 352–367.

[5] J. McLean, "A general theory of composition for trace sets closed under selective interleaving functions," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1994, pp. 79–93.

[6] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," *Mathematical Structures in Computer Science*, vol. 21, no. 6, pp. 1207–1252, 2011.

[7] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *USENIX Security Symposium*. USENIX Association, 2016, pp. 53–70.

[8] C. Sung, B. Paulsen, and C. Wang, "CANAL: a cache timing analysis framework via LLVM transformation," in *ASE*. ACM, 2018, pp. 904–907.

[9] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *ISSTA*. ACM, 2018, pp. 15–26.