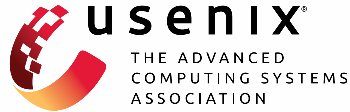

DATA – Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries

Samuel Weiser, *Graz University of Technology;* **Andreas Zankl**, *Fraunhofer AISEC;* **Raphael Spreitzer**, *Graz University of Technology;* **Katja Miller**, *Fraunhofer AISEC;* **Stefan Mangard**, *Graz University of Technology;* **Georg Sigl**, *Fraunhofer AISEC; Technical University of Munich*

https://www.usenix.org/conference/usenixsecurity18/presentation/weiser

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-939133-04-5

Open access to the Proceedings of the 27th USENIX Security Symposium is sponsored by USENIX.

DATA – Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries

Samuel Weiser[1], Andreas Zankl[2], Raphael Spreitzer[1],
Katja Miller[2], Stefan Mangard[1], and Georg Sigl[2,3]

[1]Graz University of Technology, [2]Fraunhofer AISEC, [3]Technical University of Munich

Abstract

Cryptographic implementations are a valuable target for address-based side-channel attacks and should, thus, be protected against them. Countermeasures, however, are often incorrectly deployed or completely omitted in practice. Moreover, existing tools that identify information leaks in programs either suffer from imprecise abstraction or only cover a subset of possible leaks. We systematically address these limitations and propose a new methodology to test software for information leaks.

In this work, we present DATA, a differential address trace analysis framework that detects address-based side-channel leaks in program binaries. This accounts for attacks exploiting caches, DRAM, branch prediction, controlled channels, and likewise. DATA works in three phases. First, the program under test is executed to record several address traces. These traces are analyzed using a novel algorithm that dynamically re-aligns traces to increase detection accuracy. Second, a generic leakage test filters differences caused by statistically independent program behavior, e.g., randomization, and reveals true information leaks. The third phase classifies these leaks according to the information that can be obtained from them. This provides further insight to security analysts about the risk they pose in practice.

We use DATA to analyze OpenSSL and PyCrypto in a fully automated way. Among several expected leaks in symmetric ciphers, DATA also reveals known and previously unknown leaks in asymmetric primitives (RSA, DSA, ECDSA), and DATA identifies erroneous bug fixes of supposedly fixed constant-time vulnerabilities.

1 Introduction

Side-channel attacks infer sensitive information, such as cryptographic keys or private user data, by monitoring inadvertent information leaks of computing devices. Cryptographic implementations are a valuable target for various side-channel attacks [11, 45, 77], as a successful attack undermines cryptographic security guarantees. Especially software-based microarchitectural attacks (e.g., cache attacks, DRAM attacks, branch-prediction attacks, and controlled-channel attacks) are particularly dangerous since they can be launched from software and, thus, without the need for physical access. Many of these software-based attacks exploit address-based information leakage to recover cryptographic keys of symmetric [6, 36] or asymmetric [28, 87] primitives.

Various countermeasures against address-based information leakage have been proposed on an architectural level [52, 62, 81]. However, these require changing the hardware, which prohibits fast and wide adoption. A more promising line of defense are software countermeasures, which remove address-based information leaks by eliminating key-dependent memory accesses to data and code memory. For example, data leakage can be thwarted by means of bit-slicing [43, 47, 66], and control-flow leakage by unifying the control flow [21]. Even though software countermeasures are already well studied, in practice their adoption to crypto libraries is often partial, error-prone, or non-transparent, as demonstrated by recent attacks on OpenSSL [27, 28, 88].

To address these issues, leakage detection tools have been developed that allow developers and security analysts to identify address-based side-channel vulnerabilities. Most of these tools, however, primarily focus on cache attacks and can be classified into static and dynamic approaches. Many static analysis methods use abstract interpretation [24, 25, 48, 57] to give upper leakage bounds, ideally proving the absence of information leaks in already secured implementations, e.g., the evaluation of Salsa20 [24]. However, these approaches struggle to accurately describe and pinpoint information leaks due to over-approximation [24, page 443], rendering leakage bounds meaningless in the worst case. Moreover, their approximations of the program's data plane fundamentally prohibit the analysis of interpreted code.

In contrast, dynamic approaches [41, 84, 89] focus on concrete program executions to reduce false positives. Contrary to static analysis, dynamic analysis cannot prove the absence of leakage without exhaustive input search, which is infeasible for large input spaces. However, in case of cryptographic algorithms, testing a subset of inputs is enough to encounter information leaks with a high probability, because crypto primitives heavily diffuse the secret input during processing. Thus, there is a fundamental trade-off between static analysis (minimizing false negatives) and dynamic analysis (minimizing false positives).

We aim for a pragmatic approach towards minimizing false positives, allowing developers to identify information leaks in real-world applications. Thus, we focus on dynamic analysis and tackle the limitations of existing tools. In particular, existing tools either focus on control-flow leaks or data leaks, but not both at the same time [80,89]; they consider the strongest adversary to observe cache-line accesses only [41], which is too coarse-grained in light of recent attacks (CacheBleed [88]); many of them lack the capability to properly filter program activity that is statistically independent of secret input [50, 80, 84]; and most do not provide any means to further assess the severity of information leaks, *i.e.*, the risk they bring and the urgency with which they must be fixed. Based on these shortcomings, we argue that tools designed to identify address-based information leaks must tackle the following four challenges:

1. *Leakage origin*: Detect the exact location of data and control-flow leaks in programs on byte-address granularity instead of cache-line granularity.
2. *Detection accuracy*: Minimize false positives, e.g., caused by non-determinism that is statistically independent of the secret input, and provide reasonable strategies to also reduce false negatives.
3. *Leakage classification*: Provide means to classify leaks with respect to the information gained by an adversary.
4. *Practicality*: Report information leaks (i) fully automated, *i.e.*, without requiring manual intervention, (ii) using only the program binary, *i.e.*, without requiring the source code, and (iii) efficiently in terms of performance.

In this work, we tackle these challenges with *differential address trace analysis* (DATA), a methodology and tool to identify address-based information leaks in application binaries. DATA is intended to be a companion during testing and verification of security-critical software.[1] It targets programs processing secret input, e.g., keys or passwords, and reveals dependencies between the secret and the program execution. Every leak that DATA identifies in a program is potentially exposed to side-channel attacks. DATA works in three phases.

Difference Detection: The first phase generates noiseless address traces by executing the target program with binary instrumentation. It identifies differences in these traces on a byte-address granularity. This accounts for all address-based side-channel attacks such as cache attacks [61,64,87], DRAM attacks [65], branch-prediction attacks [1], controlled-channel attacks [86], and many blackbox timing attacks [11].

Leakage Detection: The second phase tests data and control-flow differences for dependencies on the secret input. A *generic* leakage test compares the address traces of (i) a fixed secret input and (ii) random secret inputs. If the traces differ significantly, the corresponding data or control-flow differences are labeled as secret-dependent leaks. This minimizes false positives and explicitly addresses non-deterministic program behavior introduced by blinding or probabilistic encryption, for example.

Leakage Classification: The third phase classifies the information leakage of secret-dependent data and control-flow differences. This is achieved with *specific* leakage tests that find linear and non-linear relations between the secret input and the address traces. These leakage tests are a valuable tool for security analysts to determine the severity and exploitability of a leak.

We implement DATA in a fully automated evaluation tool that allows analyzing large software stacks, including initialization operations, such as key loading and parsing, as well as cryptographic operations. We use DATA to analyze OpenSSL and PyCrypto, confirming existing and identifying new vulnerabilities. Among several expected leaks in symmetric ciphers (AES, Blowfish, Camellia, CAST, Triple DES, ARC4), DATA also reveals known and previously unknown leaks in asymmetric primitives (RSA, DSA, ECDSA) and identifies erroneous bug fixes of supposedly resolved vulnerabilities.

Outline. The remainder of this paper is organized as follows. In Section 2, we discuss background information and related work. In Section 3, we present DATA on a high level. In Sections 4–6 we describe the three phases of DATA. In Section 7, we give implementation details. In Section 8, we evaluate DATA on OpenSSL and PyCrypto. In Section 9, we discuss possible leakage mitigation techniques. Finally, we conclude in Section 10.

2 Background and Related Work

2.1 Microarchitectural Attacks

Microarchitectural side-channel attacks rely on the exploitation of information leaks resulting from contention for shared hardware resources. Especially microarchitectural components such as the CPU cache, the DRAM,

[1] DATA is open-source and can be retrieved from `https://github.com/Fraunhofer-AISEC/DATA`.

and the branch prediction unit, where contention is based on memory addresses, enable powerful attacks that can be conducted from software only. For instance, attacks exploiting the different memory access times to CPU caches (aka cache attacks) range from timing-based attacks [11] to more fine-grained attacks that infer accesses to specific memory locations [61, 64, 87]. Likewise, DRAM row buffers have been used to launch side-channel attacks [65] by exploiting row buffer conflicts of different memory addresses. Also, the branch prediction unit has been exploited to attack OpenSSL RSA implementations [1]. Xu et al. [86] demonstrated a new class of attacks on shielded execution environments like Intel SGX, called controlled-channel attacks. They enable noise free observations of memory access patterns on a page granularity. For a detailed overview on microarchitectural attacks we refer to recent survey papers [29, 76].

2.2 Detection of Information Leaks

2.2.1 Terminology

We consider a program secure if it does not contain address-based information leaks. We distinguish between data and control-flow leakage. Data leakage occurs if accessed memory locations depend on secret inputs. Control-flow leakage occurs if code execution depends on secret inputs. We further distinguish between deterministic and non-deterministic programs. Latter include any kind of non-determinism such as randomization of intermediates (blinding) or results (probabilistic constructions). A *false positive* denotes an identified information leak that is in fact none. A *false negative* denotes an information leak which was not identified.

2.2.2 Blackbox Timing Leakage Detection

These techniques measure the execution time of implementations for different classes of inputs and rely on statistical tests to infer whether or not the implementation leaks information [23]. Reparaz et al. [67] use Welch's t-test [83] to identify vulnerable cryptographic implementations. More advanced approaches use symbolic execution to give upper leakage bounds [63]. However, these approaches fall short for more fine-grained address-based attacks such as cache attacks.

2.2.3 Address-based Leakage Detection

We distinguish between static and dynamic approaches.

Static Approaches. Well-established static approaches are CacheAudit [24, 48] and follow-up works [25, 57], which symbolically evaluate all program paths. Rather than pinpointing the leakage origin, CacheAudit accumulates potential leakage into a single metric, which represents an upper-bound on the maximum leakage possible. While a zero leakage bound guarantees absence of address-based side channels, a non-zero leakage bound could become rather imprecise (false positives) due to abstractions made on the data of the program. Abstraction also fundamentally prohibits analysis of interpreted code as it is encoded in the data plane of the interpreter.

Dynamic Approaches. Dynamic analysis relies on concrete program executions, which possibly introduce false negatives. Ctgrind [50] propagates secret memory throughout the program execution to detect its usage in conditional branches or memory accesses. However, ctgrind suffers from false positives as well as false negatives [4]. In contrast, Stacco [84] records address traces and analyzes them with respect to Bleichenbacher attacks [15], for which finding a single control-flow leak suffices. Stacco does not consider data leakage, and they do not consider reducing false negatives, *i.e.*, finding multiple control-flow leaks within the traces. If they did, they would suffer from false positives due to improper trace alignment (they use Linux diff tool).

None of the above approaches supports specific leakage models to further assess the information leak. Zankl et al. [89] analyze modular exponentiations under the Hamming weight model, but they do not consider other leakage models and only detect control-flow leaks.

Combined Approaches. CacheD [80] combines dynamic trace recording with static analysis introducing both, false negatives and false positives. They symbolically execute only instructions that might be influenced by the secret key. Since they only analyze a single execution, they miss leakage in other execution paths. Moreover, they do not model control-flow leaks.

Attack-based Approaches. These are dynamic approaches that conduct specific attacks but do not generalize to other attacks. For instance, Brumley and Hakala [19] as well as Gruss et al. [36] suggested to detect implementations vulnerable to cache attacks by relying on template attacks. Irazoqui et al. [41] use cache observations and a mutual information metric to identify control-flow and data leaks. Basu et al. [9] and Chattopadhyay et al. [20] quantify the information leakage in cache attacks.

Orthogonal Work. Other approaches analyze source code [14], which does not account for compiler-introduced information leaks or platform-specific behavior (cf. [4]). Yet others demand source-code annotations [4, 5, 7] or specify entirely new languages [16]. While they can prove absence of leakage for already secured code, they struggle to pinpoint leaks in vulnerable code. In contrast, DATA is designed to find and pinpoint leakage in *insecure*, unannotated programs. After mitigating leakage found by DATA, absence of leakage could be proven using [4, 5, 7, 16, 25].

Table 1: Comparison of leakage detection tools. ● means that the tool suffers from false positives/negatives. ○ means that the tool does not suffer from false positives/negatives. $\bigcirc_S$ denotes statistical guarantees.

Tool	Approach	Finest	Covered vulnerabilities		False positives		False	Output		Source code	Tool
		granularity	CF leak	Data leak	Deterministic	Non-deterministic	negatives	Leaks	Key dependency	required	available
CacheAudit [24]	Static analysis	Cache line	✓	✓	●	●	○	Leakage bound	✗	no	✓
CacheAudit 2 [25]	Static analysis	Byte address	✓	✓	●	●	○	Leakage bound	✗	no	✓
CacheD [80]	Combined	Cache line	✗	✓	●	●	●	Leak origin	✗	no	✗
ctgrind [50]	Dynamic	Byte address	✓	✓	●	●	●	Leak origin	✗	yes	✓
Stacco [84]	Dynamic (trace-based)	Byte address	✓	✗	○[a]	●	●	Leak origin	✗	no	✗
MI-Tool [41]	Dynamic (attack-based)	Cache line	✓	✓	$\bigcirc_S$	$\bigcirc_S$	●	Leak origin	generic	yes	✗
Zankl et al. [89]	Dynamic (trace-based)	Byte address	✓	✗	$\bigcirc_S$	$\bigcirc_S$	●	Leak origin	HW	no	✓
DATA	Dynamic (trace-based)	Byte address	✓	✓	$\bigcirc_S$	$\bigcirc_S$	◐	Leak origin	generic, HW, etc.	no	✓

[a] Only the first control-flow leak is reliably identified. Reporting multiple leaks could cause false positives.

2.3 Improvement Over Existing Tools

By addressing the identified challenges in Section 1, DATA overcomes several shortcomings of existing approaches, as shown in Table 1.

Leakage Origin. DATA follows a dynamic trace-based approach to identify both control flow and data leakage on byte-address granularity. This avoids wrong assumptions about attackers, e.g., only observing memory accesses at cache-line granularity [24, 41, 80], which were disproved by more advanced attacks [1, 88]. Nevertheless, identifying information leaks on a byte granularity still allows to map them to more coarse-grained attacks.

Detection Accuracy. Static approaches like CacheAudit suffer from false positives. In contrast, DATA filters key-independent differences with a high probability, thereby reducing false positives even for non-deterministic program behavior. However, as with all dynamic approaches, DATA could theoretically miss leakage that is not triggered during execution. Nevertheless, we found that in practice few traces already suffice, e.g., ≤ 10 for asymmetric algorithms, and ≤ 3 for symmetric algorithms, due to the high diffusion provided by these algorithms. Although without formal guarantee, this gives evidence that DATA reduces false negatives successfully. Compared to others, we take multiple measures to reduce false negatives in DATA. In contrast to CacheD and ctgrind, we analyze several execution paths. Compared to Stacco, which has improper trace alignment, we report all leaks visible in the address traces. Contrary to MI-Tool, we do not only focus on a specific attack technique (e.g., cache attacks). In contrast to Zankl et al. [89], we can detect generic key dependencies. This advantage is indicated by ◐ in Table 1.

Leakage Classification. While Zankl et al. [89] use the Hamming weight (HW) model only, DATA allows testing for various leakage models as well as defining new ones. Besides pinpointing the information leaks, this represents valuable information to determine key dependencies in the identified information leaks.

Practicality. DATA analyzes information leaks fully automatically. It does so on the program binary without the need for source code, allowing analysis of proprietary software. As will be outlined in our evaluation, we achieve competitive performance, support analysis of large software stacks and even interpreted code (PyCrypto and CPython), and DATA is open source.

3 Differential Address Trace Analysis

DATA is a methodology and a tool to identify address-based information leaks in program binaries.

Threat Model. To cover a wide variety of possible attacks, we consider a powerful adversary who attempts to recover secret information from side-channel observations. In practice, attackers will likely face noisy observations because side channels typically stem from shared resources affected by noise from system load. Also, practical attacks only monitor a limited number of addresses or memory blocks. For DATA, we assume that the attacker can accurately observe full, noise-free address traces. More precisely, the attacker does not only learn the sequence of instruction pointers [59], *i.e.*, the addresses of instructions, but also the addresses of the operands that are accessed by each instruction. This is a strong attacker model that covers many side-channel attacks targeting the processor microarchitecture (e.g., branch prediction) and the memory hierarchy (e.g., various CPU caches, prefetching, DRAM). A strong model is preferable here to detect as many vulnerabilities as possible. In line with [35], we consider defenses, such as address space layout randomization (ASLR) and code obfuscation, as ineffective against powerful attackers.

Limitations. DATA covers software side channels of components that operate on address information only, e.g., cache prefetching and replacement, and branch prediction. In contrast, the recent Spectre [44] and Meltdown [51] bugs exploit not only *address* information but actual *data* which is speculatively processed but insufficiently isolated across different execution contexts. In these attacks, sensitive data spills over to the address bus. These hardware bugs cannot be detected by analyzing software binaries with tools listed in Table 1.

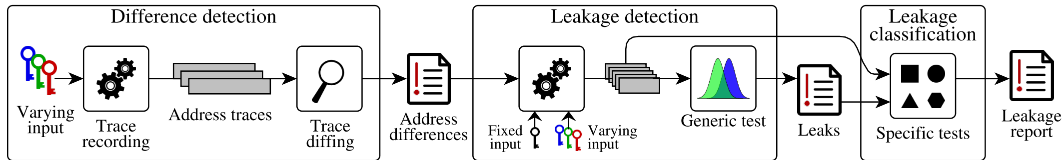

Figure 1: Overview of differential address trace analysis (DATA).

While software-only defenses exist for specific CPU models [22, 78], a generic solution should fix the hardware.

Methodology. DATA consists of three phases, the difference detection phase, the leakage detection phase, and the leakage classification phase, as depicted in Figure 1.

In the *difference detection phase*, we execute the target program multiple times with varying secret inputs and record all accessed addresses with dynamic binary instrumentation in so-called address traces. Thereby, we ensure to capture both, control flow and data leakages at their exact origin. The recorded address traces are then compared and address differences are reported.

The *leakage detection phase* verifies whether reported address differences are actually secret-dependent and filters all that are statistically independent. For this step, the program is repeatedly executed with one fixed secret input and a set of varying (random) secret inputs. In contrast to the previous phase, only the initially reported differences need to be monitored. The address traces belonging to the fixed input are then compared to those of the random inputs using a *generic* leakage test. Statistical differences are reported as true information leaks.

The *leakage classification phase* helps security analysts to assess the severity of previously confirmed leaks. This is done with *specific* leakage tests that find linear or non-linear relations between a given secret input and the previously recorded address traces. Such relations are formulated as so-called *leakage models*, e.g., the Hamming weight model. If a relation is found, the corresponding leakage model defines the information an attacker can learn about the secret input by observing memory accesses to the identified addresses. All detected relations are included in the final leakage report.

Relation to Similar Concepts. The idea of DATA is similar to differential power analysis (DPA) [46], which works on power traces. However, power traces are often noisy due to measurement uncertainty and the underlying physics. Hence, DPA often requires several thousand measurements and non-constant time implementations demand heavy pre-processing to correctly align power traces [55]. In contrast, address traces are noise-free, which minimizes the number of required measurements and allows perfect re-alignment for non-constant time traces (due to control-flow leaks).

DATA is also related to differential computation analysis (DCA) [17]. DCA relies on software execution traces to attack white-box crypto implementations. While DCA is conceptually similar to DATA, DCA attacks (white-box model) consider a much stronger adversary who can read the actual content of accessed memory locations.

4 Difference Detection Phase

We now introduce address-based information leaks and discuss the steps to identify them, namely recording of address traces and finding differences within the traces.

Notation. DATA analyzes a program binary P with respect to address leakage of secret input k. Let $P(k)$ denote the execution of a program with controllable secret input k. We write $t = \mathsf{trace}(P(k))$ to record a trace of accessed addresses during program execution. We define an address trace $t = [a_0, a_1, a_2, a_3 ...]$ as a sequence of executed instructions, augmented with memory addresses. For instructions operating on CPU registers, $a_i = ip$ holds the current instruction pointer ip. In case of memory operations, $a_i = (ip, d)$ also holds the accessed memory address d. Information leaks appear as differences in address traces. We develop an algorithm $\mathsf{diff}(t_1, t_2)$ that, given a pair of traces (t_1, t_2), identifies all differences. If the traces are equal, $\mathsf{diff}(t_1, t_2) = \varnothing$. A deterministic program P is leakage free if and only if no differences show up for any pair of secret inputs (k_i, k_j):

$$\forall k_i, k_j : \mathsf{diff}(\mathsf{trace}(P(k_i)), \mathsf{trace}(P(k_j))) = \varnothing \quad (1)$$

4.1 Address-based Information Leakage

Data leakage is characterized by one and the same instruction (ip) accessing different memory locations (d). Consider the code snippet in Listing 1, assuming line numbers equal code addresses. Execution with two different keys $key_A = [10, 11, 12]$ and $key_B = [16, 17, 18]$ yields two address traces $t_A = \mathsf{trace}(P(key_A))$ and $t_B = \mathsf{trace}(P(key_B))$, with differences marked bold:

$$t_A = [0, 18, 19, (17, 1), 20, (17, \mathbf{11}), 21, (17, \mathbf{12}), 22, (17, \mathbf{13}), 23]$$

$$t_B = [0, 18, 19, (17, 1), 20, (17, \mathbf{01}), 21, (17, \mathbf{02}), 22, (17, \mathbf{03}), 23]$$

The function 'transform' leaks the argument $kval$, which is used as index into the array LUT (line 17).

```
 0 program entry: call process with user-input
 1 unsigned char LUT[16] = { 0x52,
 2 0x19,
   ...
16 0x37};
17 int transform(int kval) { return LUT[kval%16]; }
18 int process(int key[3]) {
19   int val = transform(0);
20   val += transform(key[0]);
21   val += transform(key[1]);
22   val += transform(key[2]);
23   return val;
   }
```

Listing 1: Table look-up causing data leak.

Since the base address of LUT is 1, this operation leaks memory address $kval+1$. The first call to transform (line 19) with $kval = 0$ results in $a_1 = (17,1)$. Subsequent calls (line 20–22) leak sensitive key bytes. The differences in the traces—marked bold—reveal key dependencies.

To accurately report data leakage and to distinguish non-leaking cases (line 19) from leaking cases (line 20–22), we take the call stack into account. We formalize data leaks as tuples (ip, cs, ev) of the leaking instruction ip, its call stack cs, and the evidence ev. The call stack is a list of caller addresses leading to the leaking function. For example, the first leak has the call stack $cs = [0,20]$. The evidence is a set of leaking data addresses d. The larger the evidence set, the more information leaks. For example, $ev = \{11,01\}$ for the first leak, $ev = \{12,02\}$ for the second one, etc. Our diff algorithm would report:

$$\begin{aligned}\mathsf{diff}(t_A,t_B) = \{&(17,[0,20],\{11,01\}),\\ &(17,[0,21],\{12,02\}),\\ &(17,[0,22],\{13,03\})\}\end{aligned}$$

Control-flow leakage is caused by key-dependent branches. Consider the exponentiation in Listing 2, executed with two keys $k_A = 4 = 100_b$ and $k_B = 7 = 111_b$. This yields the following address traces, where R, P, and T denote the data addresses of the variables r, p, and t.

$$\begin{aligned}\mathsf{trace}(P(k_A)) = t_A = [&0,1,2,3,4,(7,R),(8,P),(9,R),\\ &2,3,\mathbf{5,(7,T),(8,P),(9,T)},\\ &2,3,\mathbf{5,(7,T),(8,P),(9,T)},2,6]\\ \mathsf{trace}(P(k_B)) = t_B = [&0,1,2,3,4,(7,R),(8,P),(9,R),\\ &2,3,\mathbf{4,(7,R),(8,P),(9,R)},\\ &2,3,\mathbf{4,(7,R),(8,P),(9,R)},2,6]\end{aligned}$$

There are two differences in the traces, both marked bold. The differences occur due to the `if` in line 3 which branches to line 4 or 5, depending on the key bit b, and causes operations in line 7 and 9 to be done either on the intermediate variable r or a temporary variable t.

```
0 program entry: call exp with user-input
1 function exp(key, *p) {
    ...
2   foreach (bit b in key)
3     if (b)
4       mul(r, p);
      else
5       mul(t, p);
6   return r;
  }
  function mul(*a,*b) {
7   tmpA = *a;
8   tmpB = *b;
    // calculate res = tmpA * tmpB
9   *a = res;
  }
```

Listing 2: Branch causing control-flow leak.

A control-flow leak is characterized by its branch point, where the control flow diverges, and its merge point, where branches coalesce again. In this example, the branch point is at line 3 and the merge point at line 2, when the next loop iteration starts. We model control-flow leaks as tuples (ip, cs, ev) of branch point ip, call stack cs, and evidence ev. For example, both differences occur at the same call stack $cs = [0]$. Hence, they are reported as the same leak. The evidence is a set of subtraces corresponding to the two branches. Our diff algorithm would report:

$$\begin{aligned}\mathsf{diff}(t_A,t_B) = \{(3,[0],\{&[4,(7,R),(8,P),(9,R)],\\ &[5,(7,T),(8,P),(9,T)]\}\ )\}\end{aligned}$$

4.2 Recording Address Traces

We execute the program on a dynamic binary instrumentation (DBI) framework, namely Intel Pin [54], and store the accessed code and data addresses in an address trace. To execute the program in a clean and noise-free environment, we disable ASLR and keep public inputs (e.g., command line arguments, environment variables) to the program fixed. As shown in Figure 1, we repeat this multiple times with varying inputs, causing address leaks to show up as differences in the address traces.

The concept of DATA is agnostic to concrete recording tools and, hence, could also rely on other tools [71] or hardware acceleration like Intel Processor Trace (IPT) [39]. Since the recording time is small compared to trace analysis, we did not investigate other tools.

4.3 Finding Trace Differences

The trace comparison algorithm (diff) in Algorithm 1 sequentially scans a pair of traces (t_A, t_B) for address differences, while continuously re-aligning traces in the same pass. Whenever ip values match but data addresses (d) do not, a data difference is detected (lines 4–6).

Algorithm 1: Identifying address trace differences (diff).

```
input : t_A, t_B ... the two traces
output: rep ... the report of all differences
1  rep = ∅, i = 0, j = 0
2  while i < |t_A| ∧ j < |t_B| do
3      a = t_A[i], b = t_B[i]
4      if a.ip = b.ip then
5          if a.d ≠ b.d then
6              rep = rep ∪ report_data_diff(t_A, t_B, i, j)
7          end
8          i++, j++
9      else
10         rep = rep ∪ report_cf_diff(t_A, t_B, i, j)
11         (i, j) = find_merge_point(t_A, t_B, i, j)
12     end
13 end
14 return rep
```

Algorithm 2: find_merge_point

```
input : t_A, t_B ... the two traces
input : i, j ... the trace indices of the branches
output: k, l ... the indices of the merge point
1  k = i, l = j, C_A = 0, C_B = 0, S_A = ∅, S_B = ∅
2  while k < |t_A| ∧ l < |t_B| do
3      if isCall(t_A[k]) then C_A++ ;
4      if isRet(t_A[k]) then C_A-- ;
5      if isCall(t_B[l]) then C_B++ ;
6      if isRet(t_B[l]) then C_B-- ;
7      if C_A <= 0 then S_A = S_A ∪ t_A[k].ip ;
8      if C_B <= 0 then S_B = S_B ∪ t_B[l].ip ;
9      M = S_A ∩ S_B
10     if M ≠ ∅ then
11         k = find(t_A[i...k], M)
12         l = find(t_B[j...l], M)
13         return (k, l)
14     end
15     if C_A >= 0 then k++ ;
16     if C_B >= 0 then l++ ;
17 end
18 error No merge point found
```

Control-flow differences occur when ip differs (line 9–11). Differences are reported using report_data_diff and report_cf_diff using the format specified in Section 4.1.

Trace Alignment. For control-flow differences, it is crucial to determine the correct merge points, as done by Algorithm 2. Starting from the branch point, it sequentially scans both traces, extending two sets S_A and S_B (lines 7–8) with the scanned instructions. If their intersection M becomes non-empty (lines 9–10), M holds the merge point's ip. We then determine the first occurrence of M in both branches using find (lines 11–12) and realign the traces before proceeding (Algorithm 1, line 11).

Context-Sensitivity. Since control-flow leaks could incorporate additional function calls (e.g., function mul in Listing 2), we need to exclude those from the merge point search. Therefore, we maintain the current calling depth in counters C_A and C_B (lines 3–6) and skip calling depths > 0 (lines 7–8). The functions isCall(a) and isRet(a) return `true` iff the assembler instruction at address $a.ip$ is a function call or return, respectively. If the calling depth drops below zero, the trace returned to the function's call-site. We stop scanning this trace (lines 15–17) and wait for the other trace to hit a merge point.

Our context sensitive alignment also works for techniques like retpoline [78] that aim to prevent Spectre attacks, since they just add additional call/ret layers. Code directly manipulating the stack pointer (return stack refill [78], setjmp/longjmp, exceptions, etc.) could be supported by detecting such stack pointer manipulations alongside calls and rets.

Comparison to Related Work. Trace alignment has been studied before as the problem of correspondence between different execution points. Several approaches for identifying execution points exist [74]. Instruction counter based approaches [58] uniquely identify points in one execution but fail to establish a correspondence between different executions. Using calling contexts as correspondence metric could introduce temporal ambiguity in distinguishing loop iterations [75]. Xin et al. [85] formalize the problem of relating execution points across different executions as execution indexing (EI). They propose structural EI (SEI), which uses taken program paths for indexing but could lose comprehensiveness by mismatching execution points that should correspond [74]. Other approaches combine call stacks with loop counting to avoid problems of ambiguity and comprehensiveness [74]. Many demand recompilation [74, 75, 85], which prohibits their usage in our setting. Specifically, EI requires knowledge of post-dominators, typically extracted from control flow graphs (CFGs) [30], which are not necessarily available (e.g., obfuscated binaries or dynamic code generation). Using EI, Johnson et al. [42] align traces in order to propagate differences back to their originating input. We use a similar intuition as Johnson et al. in processing and aligning traces in a single pass, however, without the need to make program execution indices explicit. By constantly re-aligning traces, we inherently maintain correspondence of execution points. Our set-based approach does not require CFG or post-dominator information.

In contrast to EI, we do not explicitly recover loops. This could cause imprecision when merging control-flow leaks embedded within loops. If the two branches are significantly asymmetric in length, we might match multiple shorter loop iterations against one longer iteration, thus introducing an artificial control-flow leak (false positive) when one branch leaves the loop while the other does not. Should such leaks occur, they would be dismissed as key independent in phase two. Note that correspondence (correct alignment) would be automatically restored as soon as both branches leave the loop. Also,

this is not a fundamental limitation of DATA, as other trace alignment methods could be implemented as well.

Combining Results. We run our diff algorithm pairwise on all recorded traces and accumulate the results in an intermediate report. Testing multiple traces helps capture nested leakage, that is, leakage which appears conditionally, depending on which branches are taken in a superordinate control-flow leak. Nested leakage would remain hidden when testing trace pairs which either take the wrong superordinate branch or exercise both branches.

5 Leakage Detection Phase

We implement a *generic* leakage test to reduce the number of false positives in case of (randomized) program behavior and events that are statistically independent of the secret input. The program is repeatedly executed with one fixed secret input and a set of random secret inputs. If the distributions of accessed addresses in these two sets can be distinguished, the corresponding address differences are marked as secret-dependent. A challenge that arises during this generic leakage test is that false negatives might occur if the fixed input is particularly similar to the average random case. We address this challenge by repeating the generic leakage test with multiple distinct fixed inputs and merging the results in the end. We introduce an appropriate leakage-evidence representation to compare distributions of accessed addresses.

5.1 Evidence Representation

We unify the representation of both data and control-flow evidences in so-called *evidence traces*. These traces hold a time-ordered sequence of memory addresses that a particular instruction accesses during *one* program execution. Note the difference to evidence sets used in Section 4.1, which are computed over *multiple* program executions. Evidence traces contain all essential information exploited in practical attacks, such as how often an address is accessed [11,45] and also when, *i.e.*, at which position an address is accessed in the trace [87].

Recording. Similar to the difference detection phase, the target program is executed to gather address traces. This time, however, we only monitor the previously detected differences, which significantly reduces trace sizes and instrumentation time. For each instruction that caused address differences in the first phase, we gather individual evidence traces. Addresses accessed in case of data differences are written to the trace in chronological order. For control flow differences, the branch target addresses taken at the branch points are written to the evidence trace, again in chronological order.

Building Histograms. As we execute the target program with multiple inputs, we accumulate the evidence traces

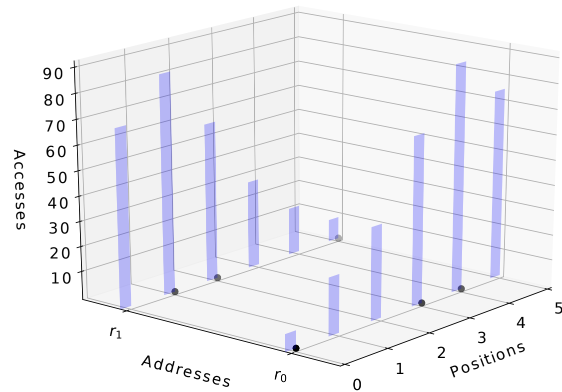

Figure 2: Histogram $\mathbf{H_{full}}$ over evidence traces.

of the *same* instruction in a two-dimensional histogram, as depicted in Figure 2. The y-axis contains the addresses accessed by the instruction, r_0 and r_1 in this case. The x-axis specifies their positions in the trace. A single evidence trace, e.g., $[r_0, r_1, r_1, r_0, r_0, r_1]$, would appear as dots in the x-y plane. When aggregating multiple traces, the z-axis accumulates all dots into bars, specifying the overall number of accesses for each address and position. This histogram, named $\mathbf{H_{full}}$, fully captures the characteristics of evidence traces, namely when and how often addresses are accessed. The downside of $\mathbf{H_{full}}$ is that a large number of traces is required to accurately estimate it. This would prolong the leakage detection phase and increase storage requirements. We therefore use two simplified histograms, each of which captures one characteristic of $\mathbf{H_{full}}$. The first one, $\mathbf{H_{addr}}$, tracks the total number of accesses per address, thus, collapsing the x-axis and omitting time information. The second one collapses the y-axis and counts the total number of accesses per position. This omits address information and is comparable to counting the length of evidence traces. Observe that counting the length of evidence traces equals the (negative) difference between consecutive positions. We therefore define $\mathbf{H_{pos}}$ as counting the length of evidence traces. We illustrate how $\mathbf{H_{addr}}$ and $\mathbf{H_{pos}}$ are compiled with the following example of three evidence traces:

$$ev_0 = [r_1, r_2], \quad ev_1 = [r_3, r_3, r_2, r_3, r_1], \quad ev_2 = [r_2, r_1, r_2]$$

$\mathbf{H_{addr}}$ contains one entry per address, counting how often each address occurs in the traces. Thus, $\mathbf{H_{addr}} = [3,4,3]$ for addresses $[r_1, r_2, r_3]$. $\mathbf{H_{pos}}$ records the length of the traces, which yields $\mathbf{H_{pos}} = [0,1,1,0,1]$ for lengths 1 to 5. For illustration purposes, counting the number of accesses per position would yield $[3,3,2,1,1,0]$ for positions 1 to 6. The (negative) differences between the positions are $[0,1,1,0,1]$, which is exactly $\mathbf{H_{pos}}$.

Implications. While the use of $\mathbf{H_{addr}}$ and $\mathbf{H_{pos}}$ reduces the measurement effort, we might miss leaks that only

show up in $\mathbf{H_{full}}$. Such a leak would occur, if the secret permutes the addresses in the evidence traces, e.g., $[r_1, r_2]$ and $[r_2, r_1]$, while the length of the evidence traces as well as the number of accesses per address remains the same. These special cases can still be detected with a multi-dimensional generic leakage test using $\mathbf{H_{full}}$.

5.2 Generic Leakage Test

We compile the evidence traces into two histograms, namely $\mathbf{H^{fix}_{addr}}$ and $\mathbf{H^{fix}_{pos}}$ for fixed secret inputs, and $\mathbf{H^{rnd}_{addr}}$ and $\mathbf{H^{rnd}_{pos}}$ for random inputs. If these histograms can be distinguished, the corresponding address difference constitutes a true information leak. In side-channel literature [33, 67], this fixed-vs-random input testing is typically done by applying Welch's t-test [83] to distributions of power consumption, electromagnetic emanation, or execution time measurements. For DATA, we cannot use the t-test, because it assumes normal distributions and evidence trace distributions are not necessarily normal. Instead, we use the more generic Kuiper's test [49], which does not make this assumption. The test essentially determines whether two probability distributions stem from the same base distribution or not. It is closely related to the Kolmogorov-Smirnov (KS) test but performs better when distributions differ in the tails instead of around the median. Since we do not assume anything about the tested distributions, we choose the increased sensitivity of Kuiper's test over the KS test at almost identical computational cost.

In preparation for Kuiper's test, we normalize our previously compiled histograms to obtain probability distributions. For the explanation of the test, assume two random variables X and Y, for which n_X and n_Y samples are observed. The first step of the test is to derive the empirical distribution functions $F_X(x)$ and $F_Y(x)$ as

$$F_X(x) = \frac{1}{n_X} \cdot \sum_{i=1}^{n_X} I_{[X_i,\infty]}(x) \ . \tag{2}$$

I is the indicator function, which is 1 if $X_i \leq x$, and 0 otherwise. $F_Y(x)$ is calculated accordingly. The Kuiper statistic V is then computed as

$$V = \sup_x [F_X(x) - F_Y(x)] + \sup_x [F_Y(x) - F_X(x)] \ . \tag{3}$$

The deviation of both distributions is significant if the Kuiper statistic V exceeds the significance threshold:

$$V_{st} = \frac{Q_{st}^{-1}(1-\alpha)}{C_{st}(n_X, n_Y)} \ . \tag{4}$$

C_{st} relates the threshold to the number of samples each empirical distribution is based on. This is important, as a larger number of samples increases the sensitivity of the Kuiper statistic. It is approximated as

$$C_{st}(n_X, n_Y) = \sqrt{\frac{n_X n_Y}{(n_X + n_Y)}} + 0.155 + \frac{0.24}{\sqrt{\frac{n_X n_Y}{(n_X+n_Y)}}} \ . \tag{5}$$

Q_{st} is derived from the asymptotic distribution of the Kuiper statistic. It links the test statistic to a certain confidence level and is defined as

$$Q_{st}(\lambda) = 2\sum_{i=1}^{\infty} \left(4i^2\lambda^2 - 1\right) e^{-2i^2\lambda^2}. \tag{6}$$

Its inverse, Q_{st}^{-1}, is calculated numerically. The value $(1-\alpha)$ determines the probability with which Kuiper's test produces false positives. For all tests performed in this work, this probability is set to 0.0001. If Kuiper's test statistic is significant, the corresponding data or control-flow difference is flagged as an information leak.
Accuracy. The probability of reporting false positives is sufficiently minimized by the choice of $(1-\alpha)$. False negatives can occur, if the histograms $\mathbf{H_{addr}}$ and $\mathbf{H_{pos}}$ are insufficient estimations of the underlying evidence distributions. This happens if the number of program executions for fixed and random inputs is too small. It is, however, a common problem of unspecific leakage testing to determine a required minimum number [55, 72]. Analysts using DATA should therefore add traces until the test results stabilize and no new leaks are detected.

6 Leakage Classification Phase

The leakage classification phase is based on a *specific* leakage test, which tests for linear and non-linear relations between the secret input and the evidences of information leaks. Finding these relations requires appropriate representations for both input and evidence traces, which are described in the following two sections.

6.1 Evidence Representation

Similar to the leakage detection phase, we collect evidence traces for multiple random secret inputs. Unlike before, however, we do not merge evidence traces into histograms, since this would dismiss information about which input belongs to which evidence trace. Instead, we aggregate evidence traces into *evidence matrices*, where each column represents a unique trace (and unique secret input). Since evidence traces might differ both in length and accessed addresses, we cannot store them directly in a matrix. Instead, we capture the characteristics of the evidence traces in two separate matrices, $\mathbf{M^{ev}_{addr}}$ and $\mathbf{M^{ev}_{pos}}$. The rows in both matrices correspond to the possible addresses in the traces. $\mathbf{M^{ev}_{addr}}$ stores the number

of accesses per address. If an address does not occur in a trace, the corresponding matrix entry is set to zero. $\mathbf{M}^{\mathbf{ev}}_{\mathbf{pos}}$ stores the position of each address in the evidence trace. If an address does not occur in a trace, the matrix entry is set to '-1'. This labels an absent address and has no negative impact on the statistical test. Any other negative value works as well, because all valid positions are non-negative integers. If an address occurs more than once in a trace, the matrix entry is set to the rounded median of the trace positions. The median adequately determines around which position in the evidence trace an address is accessed most frequently.

The following example illustrates how evidence matrices are compiled. We reuse the evidence traces ev_0 to ev_2 from Section 5.1 and insert one column for each trace. For each of the addresses r_1 to r_3, we insert one row. After adding the data, we obtain:

$$\mathbf{M}^{\mathbf{ev}}_{\mathbf{addr}} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 2 \\ 0 & 3 & 0 \end{bmatrix}, \quad \mathbf{M}^{\mathbf{ev}}_{\mathbf{pos}} = \begin{bmatrix} 0 & 4 & 1 \\ 1 & 2 & 1 \\ -1 & 1 & -1 \end{bmatrix}$$

6.2 Leakage Model

The transformation of the input is called *leakage model*. It defines which property or part of the secret input is compared to the evidence representations stored in $\mathbf{M}^{\mathbf{ev}}_{\mathbf{addr}}$ and $\mathbf{M}^{\mathbf{ev}}_{\mathbf{pos}}$. This serves two purposes. First, it confines the scope of the statistical test. This is important because the complete input space of a secret is often too large to handle in practice, e.g., $> 2^{128}$ for strong cryptographic keys. Second, this confinement implicitly quantifies the information an adversary could gain from observing evidences. A well-known leakage model is the Hamming weight model [55], which reduces a secret input to the number of its 1-bits. In [89], the Hamming weight model is used to find leaks in asymmetric cipher implementations. Another popular approach is slicing the secret input into smaller chunks [46], e.g., bytes or bits. While input slices are a good fit for byte- and bitwise operations in symmetric ciphers, they might not be the best fit for big-integer operations in asymmetric ciphers. Clearly, the choice of an appropriate leakage model is important, but ultimately depends on the target program. It requires some degree of domain knowledge, which we assume that analysts have. Our framework is designed to support a variety of leakage models, including Hamming weight and input slicing.

6.3 Specific Leakage Test

For the specific leakage test, the target binary is executed n times with random secret inputs. Instead of gathering new measurements, we reuse the (random input) traces from the leakage detection phase. In preparation for the test, we derive $\mathbf{M}^{\mathbf{ev}}_{\mathbf{addr}}$ and $\mathbf{M}^{\mathbf{ev}}_{\mathbf{pos}}$ from the traces. We also transform the secret inputs according to the chosen leakage model L and store the results in the input matrix $\mathbf{M}^{\mathbf{in}}_{\mathbf{L}}$. Similar to the evidence matrices, every input gets assigned a column in $\mathbf{M}^{\mathbf{in}}_{\mathbf{L}}$. The number of rows is defined by the model, e.g., the Hamming weight of the entire input requires one row. All rows in $\mathbf{M}^{\mathbf{in}}_{\mathbf{L}}$ are then compared to all rows in $\mathbf{M}^{\mathbf{ev}}_{\mathbf{addr}}$ and $\mathbf{M}^{\mathbf{ev}}_{\mathbf{pos}}$. For these comparisons, the selected rows are interpreted as pairwise observations of two random variables, X and Y, with length $n_X = n_Y = n$. We then use the Randomized Dependence Coefficient (RDC) [53] to determine the relation between the observations. The RDC detects linear and non-linear relations between random variables, its test statistic R is defined between 0 and 1, with $R = 1$ showing perfect dependency and $R = 0$ stating statistical independence. The parameters of the RDC are set to the values proposed in [53]: $k = 20$ and $s = \frac{1}{6}$. In contrast to mutual information estimators and similar metrics [68], which are also used in side-channel literature [31], the RDC can be calculated efficiently, especially for large sample sizes ($n > 100$). We precompute the significance threshold R_{st} for a given confidence level α by generating a sufficiently large number ($\geq 10^4$) of statistically independent sequences of length n (the same length as the rows in $\mathbf{M}^{\mathbf{ev}}_{\mathbf{addr}}$, $\mathbf{M}^{\mathbf{ev}}_{\mathbf{pos}}$, and $\mathbf{M}^{\mathbf{in}}_{\mathbf{L}}$) and estimating the distribution of R. Since the resulting distribution is approximately normal, we estimate the mean μ and the standard deviation σ. The significance threshold is then derived from $\Phi^{-1}(x)$, which is the inverse cumulative distribution function of the standard normal distribution, as follows:

$$R_{st} = \mu + \sigma \cdot \Phi^{-1}(\alpha) . \quad (7)$$

The value $(1 - \alpha)$ determines the probability with which the RDC produces false positives. For all tests performed in this work, it is set to 0.0001. If R exceeds R_{st}, the tested rows exhibit a significant statistical relation. This means that an adversary is able to infer the values and properties of the secret input that are defined by the leakage model from side-channel observations.

Accuracy. The probability of reporting false positives is sufficiently minimized by the choice of $(1 - \alpha)$. False negatives can occur if the number of observations n is too small. Similar to the discussion in Section 5, it is not possible to determine a required minimum number of observations that holds for arbitrary target programs. Naturally, simple and direct relations will be discovered with far less observations than faint and indirect ones.

7 Implementation and Optimizations

While the concept of DATA is platform independent, we implement trace recording on top of the Intel Pin frame-

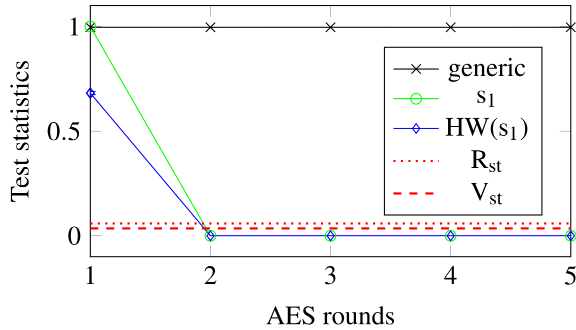

Figure 3: OpenSSL AES T-table leakage classification.

Table 2: Leakage summary of algorithms.

	Algorithm	Differences	Generic			Specific	
			Dismissed	CF	Data	Byte/Bit	HW
OpenSSL	AES-NI	0 (2)	0	0	0 (2)	0 (2)	-
	AES-VP	0	0	0	0	0	-
	AES bit-sliced	4	0	0	4	4	-
	AES T-table	20	0	0	20	20	-
	Blowfish	194	0	0	194	171	-
	Camellia	82	0	0	82	55	-
	CAST	202	0	0	202	133	-
	DES	138	0	0	138	63	-
	Triple DES	410	0	0	410	292	-
	ECDSA (secp256k1)	515	487	1	27	3	1
	DSA	781	354	160	267	19	33
	RSA	2248	1510	278	460	11	139
PyCrypto	AES	96	0	0	96	96	-
	ARC4	5	0	0	5	5	-
	Blowfish	384	0	0	384	384	-
	CAST	284	0	0	284	216	-
	Triple DES	108	0	12	96	101	-

work [38] for analyzing x86 binaries. We record address traces in separate trace files. To reduce their size, we only monitor instructions with branching behavior and their target branch as well as instructions performing memory operations. This suffices to detect control-flow and data leakage. To speed up recording of evidence traces in the second phase, we only record those instructions flagged as potential leaks in the first phase.

We implement the difference detection as well as the generic and the specific leakage tests in Python scripts, which condense all findings into human-readable leakage reports in XML format, structuring information leaks by libraries, functions, and call stacks.

Tracking Heap Allocations. Depending on the utilization of the heap, identical memory objects could get assigned different addresses by the memory allocator. During trace analysis, this could cause the same objects to be interpreted as different ones. We encountered such behavior for OpenSSL, which dynamically allocates big numbers on the heap and resizes them on demand. This causes frequent re-allocations and big numbers hopping between different heap addresses for different program executions. Our Pintool can therefore be configured to detect heap objects and replace their virtual address with its relative address offset. Currently, our analysis treats all heap objects equally, making the results more readable. More elaborate approaches like [73] are left as future work.

8 Evaluation and Results

We used Pin version 3.2-81205 for instrumentation and compiled glibc 2.24 as well as OpenSSL 1.1.0f[2] in a default configuration with additional debug information, using GCC version 6.3.0. Although debug symbols are not required by DATA, it incorporates available debug symbols in the final report. This allows to map detected leaks to the responsible functions and data symbols.

[2]Specifically, we tested commit `7477c83e15`.

8.1 Analysis Results

Table 2 shows the results of the three phases of DATA, namely address differences, generic and specific leaks.

OpenSSL (Symmetric Primitives). As summarized in the upper part of Table 2, AES-NI (AES new instructions [37]) as well as AES-VP (vector permutations based on SSSE3 extensions) do not leak. However, when using AES-NI (and other ciphers) via the OpenSSL command-line tool, the key parsing yields two data leaks, as indicated in brackets. Calling the AES-NI implementation without this command-line tool, as also done for the other three AES implementations, does not trigger these two data leaks. Besides, we identified four data leaks in the bit-sliced AES. While OpenSSL uses the protected implementation by Käspar and Schwabe [43] for the actual encryption, they use the same unprotected key expansion as used in T-table implementations.

All other tested symmetric implementations yield a significant number of data leaks since they rely on lookup tables with key-dependent memory accesses, which makes them vulnerable to cache attacks [11, 77]. These leaks have also been confirmed by the byte leakage model test. Figure 3 shows statistical test results of the vulnerable AES T-table implementation for the first five rounds, averaged over the 16 table lookups in each round. Phase two finds generic key dependencies, regardless of the round (values well above V_{st}), confirming its accuracy. The chosen byte leakage model detects linear dependencies to the first round state (s_1), which allows known-plaintext attacks [11]. For intermediate rounds, for which the chosen byte leakage model is *not* applicable, the test output is well below the threshold R_{st}. By adapting the leakage model to the last round state, one could also test for ciphertext-only attacks [60]. Moreover, one can see that the Hamming weight model on key bytes detects the same leakage but with a lower confidence, since it loses information about the key. This emphasizes the importance of choosing appropriate leak-

age models. We summarize results in Appendix A.

OpenSSL (Asymmetric Primitives). The asymmetric primitives show significant non-deterministic behavior, which is dismissed in the leakage detection phase. For example, OpenSSL uses RSA base blinding with a random blinding value. From 2248 differences in RSA, 1510 are dismissed, leaving 278 control-flow and 460 data leaks with key dependency. Among those, we found two constant-time vulnerabilities in RSA and DSA, respectively, which bypass constant-time implementations in favor of vulnerable implementations. This could allow key recovery attacks similar to [3, 82]. Moreover, DATA reconfirms address differences in the ECDSA wNAF implementation, as exploited in [10, 26, 79].

For asymmetric ciphers, we applied the Hamming weight (HW) model as well as the key bit model. The majority of leaks reported by the HW model are indicating that the length of the key or of intermediate values leaks (as the HW usually correlates with the length). For example, we detect leaks in functions that determine the length of big numbers, reconfirming the findings of [80]. Also, OpenSSL uses lazy heap allocation to resize objects on demand. This can cause different heap addresses for different key lengths, which will show up as data leakage. In contrast to the HW, the key bit model is more fine-grained and thus targets very specific leaks only, e.g., it reveals leaks that occur when the private key is parsed. This constitutes an insecure usage of the private key, and a very subtle bug to find. Details about leaking functions are given in Appendix A.

Python. We tested PyCrypto 2.6.1 running on CPython 2.7.13. The lower part of Table 2 summarizes our results. PyCrypto incorporates native shared libraries for certain cryptographic operations. From a side-channel perspective, this is desirable since those native libraries could be tightened against side-channel attacks, independently of the used interpreter. However, we found that all ciphers leak key bytes via unprotected lookup table implementations within those shared libraries, as indicated by the byte leakage model. We list the leaks in Appendix A.

Leakage-free Crypto. We analyzed Curve25519 in NaCl [13] as well as the corresponding Diffie-Hellman variant of OpenSSL (X25519) and found no address-based information leakage (apart from OpenSSL's key parsing), approving their side-channel security.

8.2 Discussion

Detection Accuracy. For symmetric algorithms in OpenSSL, we recorded up to 10 traces in the difference detection phase. We found that 3 traces are sufficient as more traces did not uncover additional differences. The low number of traces results from the high diffusion and the regular design of symmetric ciphers, which yields a

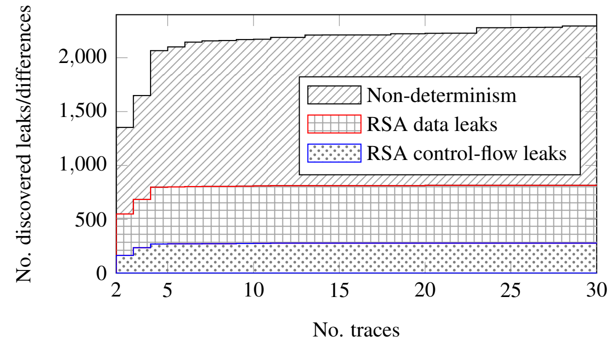

Figure 4: Dismissed non-deterministic differences and discovered leaks for OpenSSL RSA as stacked plot.

high probability for quickly hitting all variations in the program execution. This suggests that the difference detection phase achieves good accuracy for symmetric ciphers. Symmetric ciphers are typically deterministic, thus all differences are key-dependent. Indeed, Table 2 shows that the leakage detection phase confirms all differences as leaks.

To evaluate DATA's accuracy on non-deterministic programs, we tested OpenSSL asymmetric ciphers and collected up to 30 traces, as shown in Figure 4. While the address differences found in the difference detection phase do not settle within 30 traces (introducing false negatives), an important finding is that the majority of these differences are due to statistically independent program activity, e.g., RSA base blinding. These differences are characterized as key-independent and successfully filtered in the leakage detection phase. The number of actual data and control-flow leaks with key dependencies already settles at 4 traces. The few leaks observable with more traces are due to heap cleanup (these leaks were already discovered at heap allocation), leakage of the heap object's size, and exploring more paths of already discovered programming bugs. For example, DATA discovered the aforementioned RSA constant-time vulnerability, which was missed by other solutions, with only two traces. Analyzing more traces identifies more information leaks caused by the same programming bug. Hence, we recommend ≤ 10 traces for asymmetric primitives as a conservative choice. We observed similar behavior for DSA and ECDSA, but omit the details for brevity.

Performance. We ran our experiments on a Xeon E5-2630v3 with 386 GB RAM. DATA achieves good performance, adapting its runtime to the number of discovered leaks. Analysis of the leakage-free AES-NI and AES-VP took around 6 s, as only the first phase is needed. Finding leaks in the OpenSSL AES T-table implementation took 5 CPU minutes. Leakage classification took 8 CPU min. Asymmetric algorithms require more traces

and yield significantly more differences. Hence, the first phases took between 29.8 (for DSA) and 79.8 CPU minutes (for ECDSA). Running all three phases on RSA takes 233.8 CPU minutes with a RAM utilization of less than 4.5 GB (single core). By exploiting parallelism, the actual execution time can be significantly reduced, e.g., from 55 min to approximately 250 s for the first phase of RSA. Analyzing PyCrypto yields large address traces due to the interpreter (1GB and more), nevertheless DATA handles such large traces without hassle: The first phase discards all non-leaking instructions, stripping down trace sizes of the subsequent phases to kilobytes (see Appendix B).

Summary. The adoption of side-channel countermeasures is often partial, error-prone, and non-transparent in practice. Even though countermeasures have been known for over a decade [66], most OpenSSL symmetric ciphers as well as PyCrypto do not rely on protected implementations like bit-slicing. Also, the bit-sliced AES adopted by OpenSSL leaks during the key schedule, as the developers integrated it only partially [43] since practical attacks have not been shown yet. Moreover, we discovered two new vulnerabilities, bypassing OpenSSL's constant-time implementations for RSA and DSA initialization. Considering incomplete bug fixes of similar vulnerabilities identified by Garcia et al. [27, 28], this sums up to four implementation bugs related to the same countermeasure. This clearly shows that the tedious and error-prone task of implementing countermeasures should be backed by appropriate tools such as DATA to detect and appropriately fix vulnerabilities as early as possible.

We found issues in loading and parsing cryptographic keys as well as initialization routines. Finding these issues demands analysis of the full program execution, from program start to exit, which is out of reach for many existing tools. Also, analysis often neglects these information leaks because an attacker typically has no way to trigger key loading and other single events in practice. However, when using OpenSSL inside SGX enclaves (cf. Intel's SGX SSL library [40]), the attacker can trigger arbitrarily many program executions, making single-event leakage practically relevant, as demonstrated by the RSA key recovery attack in [82].

Responsible Disclosure. We informed the library developers as well as Intel of our findings. In response, OpenSSL merged our proposed patches upstream.

Security Implications. A leak found by DATA does not necessarily constitute an exploitable vulnerability. The leakage classification phase helps in rating its severity, however, an accurate judgment often demands significant effort in assembling and improving concrete attacks [12]. We argue that, unless good counter-arguments are given, any leak should be considered serious.

9 Mitigating Address-based Leaks

After using DATA to identify address-based information leaks in cryptographic software implementations, the following approaches could be applied as mitigation.

Software-based Mitigations. Coppens et al. [21] proposed compiler transformations to eliminate key-dependent control-flow dependencies. Similar approaches are followed by other program transformations [2, 56] and transactional branching [8]. Data leaks of lookup table implementations can be mitigated by bit-slicing [43, 47, 66]. Scatter-gather prevents data leaks on RSA exponentiation by interleaving data in memory such that cache lines are accessed irrespective of the used index. However, scatter-gather must be implemented correctly to prevent more sophisticated attacks [88]. Oblivious RAM [32, 77, 91] has been proposed as a generic countermeasure against data leaks by hiding memory access patterns. Hardened software implementations could then be proven leakage-free using [4, 5, 7, 16, 25].

Mitigations on Architectural/OS Level. Cache coloring [69] and similar cache isolation mechanisms [52] have been proposed to mitigate cache attacks. Others [90] proposed OS-level defenses against last-level cache attacks by controlling page sharing via a copy-on-access mechanism. Hardware transactional memory can be used to mitigate cache attacks by keeping all sensitive data in the cache during the computation [34]. Compiler-based tools aim to protect SGX enclaves against cache attacks [18] or controlled channel attacks [70].

10 Conclusion

In this work, we proposed differential address trace analysis (DATA) to identify address-based information leaks. We use statistical tests to filter non-deterministic program behavior, thus improving detection accuracy. DATA is efficient enough to analyze real-world software – from program start to exit. Thereby, we include key loading and parsing in the analysis and found leakage which has been missed before. Based on DATA, we confirmed existing and identified several unknown information leaks as well as already (supposedly) fixed vulnerabilities in OpenSSL. In addition, we showed that DATA is capable of analyzing interpreted code (PyCrypto) including the underlying interpreter, which is conceptually impossible with current static methods. This shows the practical relevance of DATA in assisting security analysts to identify information leaks as well as developers in the tedious task of correctly implementing countermeasures.

Outlook. The generic design of DATA also allows detecting other types of leakage such as variable time floating point instructions by including the instruction operands in the recorded address traces. DATA also

paves the way for analyzing other interpreted languages and quantifying the effects of interpretation and just-in-time compilation on side-channel security. Moreover, DATA could be extended to analyze multi-threaded programs by recording and analyzing individual traces per execution thread.

Acknowledgments

We would like to thank the anonymous reviewers, as well as Mario Werner and our shepherd Stephen McCamant for their valuable feedback and insightful discussions that helped improve this work.

This work was partially supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments", by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia, as well as the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No 681402).

A Leaking Functions

OpenSSL (Symmetric Primitives). To analyze AES, we implemented a wrapper that calls the algorithm directly. For other algorithms, we used the `openssl enc` command-line tool with keys in hex format. DATA identified information leaks in the code that parses these keys. In particular, the leaks occur in function `set_hex`, which uses stdlib's `isxdigit` function that performs leaking table lookups. Besides, `OPENSSL_hexchar2int` uses a switch case to convert key characters to integers. Although symmetric keys are usually stored in binary format, one should be aware of such leaks.

The bit-sliced AES implementation uses the vulnerable `_x86_64_AES_set_encrypt_key` function for key schedule. In addition, the unprotected AES leaks in function `_x86_64_AES_encrypt_compact`. Blowfish leaks at `BF_encrypt`, Camellia leaks the `LCamellia_SBOX` at `Camellia_Ekeygen` and `_x86_64_Camellia_encrypt`, CAST leaks the `CAST_S_table0` to 7 at `CAST_set_key` as well as `CAST_encrypt`, DES leaks the `des_skb` at `DES_set_key_unchecked` as well as `DES_SPtrans` at `DES_encrypt2`.

OpenSSL (Asymmetric Primitives). For the analysis of asymmetric ciphers, we use OpenSSL to generate keys in PEM format and then invoke the `openssl pkeyutl` command-line tool to create signatures with those keys.

```
1 int BN_MONT_CTX_set(BN_MONT_CTX *mont,
2                     BIGNUM *mod, BN_CTX *ctx) {
3   ...
4   BN_copy(&(mont->N), mod);
5   ...
6   BN_mod_inverse(Ri, R, &mont->N, ctx);
7   ...
8 }
```

Listing 3: OpenSSL RSA vulnerability.

Similar to symmetric ciphers, asymmetric implementations leak during key loading and parsing. We found leaks in `EVP_DecodeUpdate`, in `EVP_DecodeBlock` via lookup table `data_ascii2bin`, in `c2i_ASN1_INTEGER` that uses `c2i_ibuf` and in `BN_bin2bn`. Although the key is typically loaded only once at program startup, this has direct implications on applications using Intel SGX SSL.

DATA discovered two new vulnerabilities regarding OpenSSL's handling of constant-time implementations. The first one leaks during the initialization of Montgomery constants for secret RSA primes `p` and `q`. This is a programming bug: the so-called constant-time flag is set for `p` and `q` in function `rsa_ossl_mod_exp` but not propagated to temporary working copies inside `BN_MONT_CTX_set`, as shown in Listing 3, since the function `BN_copy` in line 3 does not propagate the consttime-flag from `mod` to `mont->N`. This causes the inversion in line 5 to fall back to non-constant-time implementations (`int_bn_mod_inverse` and `BN_div`). The second vulnerability is a missing constant-time flag for the DSA private key inside `dsa_priv_decode`. This causes the DSA key loading to use the unprotected exponentiation function `BN_mod_exp_mont`. Moreover, DATA confirms that ECDSA still uses the vulnerable point multiplication in `ec_wNAF_mul`, which was exploited in [10, 26, 79].

Finally, we found that the majority of information leaks reported for OpenSSL are leaking the length of the key or of intermediate variables. For example, we reconfirm the leak in `BN_num_bits_word` [80], which leaks the number of bits of the upper word of big numbers. There are several examples where the key length in bytes is leaked, e.g., via `ASN1_STRING_set`, `BN_bin2bn`, `strlen` of glibc as well as via heap allocation.

PyCrypto. PyCrypto symmetric ciphers leak during encryption, mostly via lookup tables. AES leaks the tables `Te0` to `Te4` and `Td0` to `Td3` in functions `ALGnew`, `rijndaelKeySetupEnc` and `rijndaelEncrypt`. Blowfish leaks in functions `ALGnew` and `Blowfish_encrypt`. CAST leaks the tables `S1` to `S4` in function `block_encrypt` and the tables `S5` to `S8` in `schedulekeys_half`. Triple DES leaks the table `des_ip` in function `desfunc` as well as `deskey`. ARC4 leaks in function `ALGnew`.

B Performance

Table 3 summarizes the performance figures of DATA for each phase.[3] Unless stated otherwise, all timings reflect the runtime in CPU minutes (single-core) and thus represent a fair and conservative metric. If tasks are parallelized, the actual runtime can be significantly reduced.

Difference Detection Phase. For OpenSSL, the trace size is < 30 MB for symmetric and < 55 MB for asymmetric ciphers. For PyCrypto, each trace has approximately 1 GB, because the execution of the interpreter is included. Regarding runtime, OpenSSL symmetric ciphers require less than a minute. PyCrypto ciphers finish in 5 minutes or less, despite large trace sizes. OpenSSL asymmetric ciphers need between 29.8 and 79.8 CPU minutes for two reasons. First, they require more traces. As we compare traces pairwise in the first phase, the runtime grows quadratically in the number of traces. Second, asymmetric ciphers yield significantly more differences that need to be analyzed. Especially control-flow differences demand costly re-alignment of traces. Yet, these results are quite encouraging, especially since the automated analysis of large real-world software stacks is out of reach for many existing tools. Also, we see possible improvements in further speeding up analysis times.

Leakage Detection Phase. We analyze three fixed and one random set à 60 traces, yielding 240 traces in total. Since this phase only analyzes address differences reported by the previous phase, the sizes of the recorded traces are significantly smaller. From several MB to over 1 GB in phase one, the traces are now several KB to around 1.3 MB for RSA. This makes recording and analyzing an even larger number of traces, e.g., more than 240, efficient. For example, the analysis of OpenSSL bit-sliced AES takes less than 5 CPU minutes. As expected, analyzing PyCrypto takes longer due to the instrumentation of the Python interpreter. Also, analysis of RSA is slower due to the high number of address differences to analyze. For example, RSA generates traces of up to 1343.9 KB to be analyzed. Nevertheless, phase two completes within less than 61 CPU minutes.

Leakage Classification Phase. The last phase records and analyzes 200 traces with random keys. To speed up recording, we reuse traces from the random input set of the previous phase. We benchmarked symmetric ciphers with the byte leakage model. Analysis times vary heavily between ciphers, because the performance critically depends on the number of reported address leaks and the size of the evidences, which need to be classified. For instance, most ciphers complete in less than 80 minutes, and AES bit-sliced in even 3.2 minutes. In contrast, PyCrypto Blowfish took almost 9 CPU hours because of a much larger number of evidences compared to PyCrypto AES, as can be seen from their trace sizes (271.8 kB for Blowfish versus 13.6 kB for AES). In general, testing the HW model is faster than the bit model because the HW cumulates all key bits into a single metric, while for the bit model we need to analyze multiple key bits independently. Table 2 shows that the cumulative runtime over both models is between 55 and 95 min. Also, the classification phase is generally slower than the leakage detection phase. This is because, first, DATA performs more specific leakage tests than generic ones ($\mathbf{H}_{\mathbf{addr/pos}}$ vs. $\mathbf{M}^{\mathbf{ev}}_{\mathbf{addr/pos}}$), and second, the RDC is more costly to compute than Kuiper's test. We believe significant performance savings are possible by pruning large evidence lists and by optimizing the RDC implementation.

Summary. The last two columns illustrate that the overall performance of DATA adapts to the amount of discovered leakage, which is desirable. Leakage-free implementations finish within 6 s, while leaky ones take up to 580 CPU minutes. In any of the phases, analysis requires less than 4.5 GB of RAM when executing on a single core. This is within the range of desktop computers and commodity laptops. When multi-core environments are available, one can exploit parallelism to greatly speed up analysis times. In fact, we parallelized phase one and reduced its runtime for RSA from 55 CPU minutes to approximately 250 real seconds. Similar optimizations could be implemented for phase two and three. Moreover, when doing frequent testing, software developers could not only omit the leakage classification phase intended for security analysts but also skip the leakage detection phase in case of deterministic algorithms.

[3] The overall performance might be higher than the sum of all phases because it includes the generation of final reports.

References

[1] Aciiçmez, O., Koç, Ç. K., and Seifert, J. Predicting Secret Keys Via Branch Prediction. In *Topics in Cryptology – CT-RSA 2007* (2007), vol. 4377 of *LNCS*, Springer, pp. 225–242.

[2] Agat, J. Transforming Out Timing Leaks. In *Principles of Programming Languages – POPL 2000* (2000), ACM, pp. 40–53.

[3] Aldaya, A. C., García, C. P., Tapia, L. M. A., and Brumley, B. B. Cache-Timing Attacks on RSA Key Generation. *IACR Cryptology ePrint Archive 2018* (2018), 367.

[4] Almeida, J. B., Barbosa, M., Barthe, G., Dupressoir, F., and Emmi, M. Verifying Constant-Time Implementations. In *USENIX Security Symposium 2016* (2016), USENIX Association, pp. 53–70.

[5] Almeida, J. B., Barbosa, M., Pinto, J. S., and Vieira, B. Formal Verification of Side-Channel Countermeasures Using Self-Composition. *Sci. Comput. Program. 78* (2013), 796–812.

[6] Apecechea, G. I., Eisenbarth, T., and Sunar, B. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES. In *IEEE Symposium on Security and Privacy – S&P 2015* (2015), IEEE Computer Society, pp. 591–604.

[7] Barthe, G., Betarte, G., Campo, J. D., Luna, C. D., and Pichardie, D. System-level Non-interference for Constant-

Table 3: Performance of DATA during the analysis of OpenSSL (top) and PyCrypto (bottom). Sizes are per trace. Time is in CPU minutes. Trace sizes for *Classification* are identical to *Leakage Detection*.

	Algorithm	**Difference Detection**			**Leakage Detection**			**Classification**		**Total**	
		Traces	Size (MB)	Time (min.)	Traces	Size (kB)	Time (min.)	Traces	Time (min.)	Time (min.)	RAM (MB)
OpenSSL	AES-NI	3	0.5	0.1	-	-	-	-	-	0.1	72.0
	AES-VP	3	0.5	0.1	-	-	-	-	-	0.1	72.2
	AES bit-sliced	3	0.5	0.4	240	0.2	4.6	200	3.2	8.4	77.1
	AES T-table	3	0.5	0.4	240	1.8	4.6	200	8.0	13.2	101.4
	Blowfish	3	28.2	0.8	240	264.8	13.7	200	79.1	96.0	717.8
	Camellia	3	27.3	0.6	240	2.5	9.0	200	17.5	27.3	146.8
	CAST	3	27.3	0.6	240	5.4	9.2	200	36.3	46.4	247.5
	DES	3	27.3	0.6	240	3.9	9.1	200	9.9	19.9	139.5
	Triple DES	3	27.3	0.7	240	13.9	10.5	200	49.2	60.9	351.7
	ECDSA (secp256k1)	10	54.1	79.8	240	387.9	18.3	200	55.3	161.2	1,316.3
	DSA	10	35.6	29.8	240	195.4	14.7	200	56.9	106.1	1,054.6
	RSA	10	44.2	55.0	240	1,343.9	60.9	200	94.3	233.8	4,414.0
PyCrypto	AES	3	1081.6	4.0	240	13.6	43.6	200	88.2	136.2	1,223.0
	ARC4	3	1081.5	3.9	240	6.4	43.1	200	60.3	107.6	1,222.7
	Blowfish	3	1082.3	5.0	240	271.8	47.9	200	526.5	582.2	2,302.6
	CAST	3	1081.6	4.0	240	11.8	44.0	200	76.7	125.1	1,223.0
	Triple DES	3	1082.4	4.2	240	65.8	45.0	200	63.3	113.3	1,223.8

time Cryptography. In *Conference on Computer and Communications Security – CCS 2014* (2014), ACM, pp. 1267–1279.

[8] Barthe, G., Rezk, T., and Warnier, M. Preventing Timing Leaks Through Transactional Branching Instructions. *Electr. Notes Theor. Comput. Sci. 153* (2006), 33–55.

[9] Basu, T., and Chattopadhyay, S. Testing Cache Side-Channel Leakage. In *Internaional Conference on Software Testing, Verification and Validation Workshops – ICST Workshops* (2017), IEEE Computer Society, pp. 51–60.

[10] Benger, N., van de Pol, J., Smart, N. P., and Yarom, Y. ”Ooh Aah... Just a Little Bit” : A Small Amount of Side Channel Can Go a Long Way. In *Cryptographic Hardware and Embedded Systems – CHES 2014* (2014), vol. 8731 of *LNCS*, Springer, pp. 75–92.

[11] Bernstein, D. J. Cache-Timing Attacks on AES, 2004. Technical report: `https://cr.yp.to/antiforgery/cachetiming-20050414.pdf`. Accessed: 2018-05-29.

[12] Bernstein, D. J., Breitner, J., Genkin, D., Bruinderink, L. G., Heninger, N., Lange, T., van Vredendaal, C., and Yarom, Y. Sliding Right into Disaster: Left-to-Right Sliding Windows Leak. In *Cryptographic Hardware and Embedded Systems – CHES 2017* (2017), vol. 10529 of *LNCS*, Springer, pp. 555–576.

[13] Bernstein, D. J., Lange, T., and Schwabe, P. NaCl: Networking and Cryptography library. `https://nacl.cr.yp.to/`. Accessed: 2018-05-29.

[14] Blazy, S., Pichardie, D., and Trieu, A. Verifying Constant-Time Implementations by Abstract Interpretation. In *European Symposium on Research in Computer Security – ESORICS 2017* (2017), vol. 10492 of *LNCS*, Springer, pp. 260–277.

[15] Bleichenbacher, D. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In *Advances in Cryptology – CRYPTO 1998* (1998), vol. 1462 of *LNCS*, Springer, pp. 1–12.

[16] Bond, B., Hawblitzel, C., Kapritsos, M., Leino, K. R. M., Lorch, J. R., Parno, B., Rane, A., Setty, S. T. V., and Thompson, L. Vale: Verifying High-Performance Cryptographic Assembly Code. In *USENIX Security Symposium 2017* (2017), USENIX Association, pp. 917–934.

[17] Bos, J. W., Hubain, C., Michiels, W., and Teuwen, P. Differential Computation Analysis: Hiding Your White-Box Designs is Not Enough. In *Cryptographic Hardware and Embedded Systems – CHES 2016* (2016), vol. 9813 of *LNCS*, Springer, pp. 215–236.

[18] Brasser, F., Capkun, S., Dmitrienko, A., Frassetto, T., Kostiainen, K., Müller, U., and Sadeghi, A. DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization. *CoRR abs/1709.09917* (2017).

[19] Brumley, B. B., and Hakala, R. M. Cache-Timing Template Attacks. In *Advances in Cryptology – ASIACRYPT 2009* (2009), vol. 5912 of *LNCS*, Springer, pp. 667–684.

[20] Chattopadhyay, S., Beck, M., Rezine, A., and Zeller, A. Quantifying the information leak in cache attacks via symbolic execution. In *International Conference on Formal Methods and Models for System Design – MEMOCODE 2017* (2017), ACM, pp. 25–35.

[21] Coppens, B., Verbauwhede, I., Bosschere, K. D., and Sutter, B. D. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *IEEE Symposium on Security and Privacy – S&P 2009* (2009), IEEE Computer Society, pp. 45–60.

[22] Corbet, J. The current state of kernel page-table isolation, 2018. `https://lwn.net/Articles/741878/`. Accessed: 2018-05-29.

[23] Coron, J., Kocher, P. C., and Naccache, D. Statistics and Secret Leakage. In *Financial Cryptography – FC 2000* (2000), vol. 1962 of *LNCS*, Springer, pp. 157–173.

[24] Doychev, G., Feld, D., Köpf, B., Mauborgne, L., and Reineke, J. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security Symposium 2013* (2013), USENIX Association, pp. 431–446.

[25] DOYCHEV, G., AND KÖPF, B. Rigorous Analysis of Software Countermeasures Against Cache Attacks. In *Programming Language Design and Implementation – PLDI 2017* (2017), ACM, pp. 406–421.

[26] FAN, S., WANG, W., AND CHENG, Q. Attacking OpenSSL Implementation of ECDSA with a Few Signatures. In *Conference on Computer and Communications Security – CCS 2016* (2016), ACM, pp. 1505–1515.

[27] GARCÍA, C. P., AND BRUMLEY, B. B. Constant-Time Callees with Variable-Time Callers. In *USENIX Security Symposium 2017* (2017), USENIX Association, pp. 83–98.

[28] GARCÍA, C. P., BRUMLEY, B. B., AND YAROM, Y. "Make Sure DSA Signing Exponentiations Really are Constant-Time". In *Conference on Computer and Communications Security – CCS 2016* (2016), ACM, pp. 1639–1650.

[29] GE, Q., YAROM, Y., COCK, D., AND HEISER, G. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *J. Cryptographic Engineering 8* (2018), 1–27.

[30] GEORGIADIS, L., WERNECK, R. F. F., TARJAN, R. E., TRIANTAFYLLIS, S., AND AUGUST, D. I. Finding Dominators in Practice. In *European Symposium on Algorithms – ESA 2004* (2004), vol. 3221 of *LNCS*, Springer, pp. 677–688.

[31] GIERLICHS, B., BATINA, L., TUYLS, P., AND PRENEEL, B. Mutual Information Analysis. In *Cryptographic Hardware and Embedded Systems – CHES 2008* (2008), vol. 5154 of *LNCS*, Springer, pp. 426–442.

[32] GOLDREICH, O., AND OSTROVSKY, R. Software Protection and Simulation on Oblivious RAMs. *J. ACM 43* (1996), 431–473.

[33] GOODWILL, G., JUN, B., JAFFE, J., AND ROHATGI, P. A Testing Methodology for Side Channel Resistance Validation, 2011. `http://csrc.nist.gov/news_events/non-invasive-attack-testing-workshop/papers/08_Goodwill.pdf`. Accessed: 2018-05-29.

[34] GRUSS, D., LETTNER, J., SCHUSTER, F., OHRIMENKO, O., HALLER, I., AND COSTA, M. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security Symposium 2017* (2017), USENIX Association, pp. 217–233.

[35] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *Conference on Computer and Communications Security – CCS 2016* (2016), ACM, pp. 368–379.

[36] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium 2015* (2015), USENIX Association, pp. 897–912.

[37] GUERON, S. White Paper: Intel Advanced Encryption Standard (AES) Instructions Set, 2010. `https://software.intel.com/file/24917`. Accessed: 2018-05-29.

[38] INTEL. Pin - A Dynamic Binary Instrumentation Tool. `https://software.intel.com/en-us/articles/pintool/`. Accessed: 2018-05-29.

[39] INTEL. Intel 64 and IA-32 Architectures Software Developers Manual, 2016. Reference no. 325462-061US.

[40] INTEL. Intel SgxSSL Library User Guide, 2018. Rev. 1.2.1. `https://software.intel.com/sites/default/files/managed/3b/05/Intel-SgxSSL-Library-User-Guide.pdf`. Accessed: 2018-05-29.

[41] IRAZOQUI, G., CONG, K., GUO, X., KHATTRI, H., KANUPARTHI, A. K., EISENBARTH, T., AND SUNAR, B. Did we learn from LLC Side Channel Attacks? A Cache Leakage Detection Tool for Crypto Libraries. *CoRR abs/1709.01552* (2017).

[42] JOHNSON, N. M., CABALLERO, J., CHEN, K. Z., MCCAMANT, S., POOSANKAM, P., REYNAUD, D., AND SONG, D. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *IEEE Symposium on Security and Privacy – S&P 2011* (2011), IEEE Computer Society, pp. 347–362.

[43] KÄSPER, E., AND SCHWABE, P. Faster and Timing-Attack Resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems – CHES 2009* (2009), vol. 5747 of *LNCS*, Springer, pp. 1–17.

[44] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution. *meltdownattack.com* (2018).

[45] KOCHER, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO 1996* (1996), vol. 1109 of *LNCS*, Springer, pp. 104–113.

[46] KOCHER, P. C., JAFFE, J., AND JUN, B. Differential Power Analysis. In *Advances in Cryptology – CRYPTO 1999* (1999), vol. 1666 of *LNCS*, Springer, pp. 388–397.

[47] KÖNIGHOFER, R. A Fast and Cache-Timing Resistant Implementation of the AES. In *Topics in Cryptology – CT-RSA 2008* (2008), vol. 4964 of *LNCS*, Springer, pp. 187–202.

[48] KÖPF, B., MAUBORGNE, L., AND OCHOA, M. Automatic Quantification of Cache Side-Channels. In *Computer Aided Verification – CAV 2012* (2012), vol. 7358 of *LNCS*, Springer, pp. 564–580.

[49] KUIPER, N. H. Tests concerning random points on a circle. *Indagationes Mathematicae (Proceedings) 63*, Supplement C (1960), 38–47.

[50] LANGLEY, A. ctgrind: Checking that Functions are Constant Time with Valgrind. `https://github.com/agl/ctgrind`. Accessed: 2018-05-29.

[51] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown. *meltdownattack.com* (2018).

[52] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C. V., HEISER, G., AND LEE, R. B. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *High Performance Computer Architecture – HPCA 2016* (2016), IEEE Computer Society, pp. 406–418.

[53] LÓPEZ-PAZ, D., HENNIG, P., AND SCHÖLKOPF, B. The Randomized Dependence Coefficient. In *Neural Information Processing Systems – NIPS 2013* (2013), pp. 1–9.

[54] LUK, C., COHN, R. S., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, P. G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. M. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming Language Design and Implementation – PLDI 2005* (2005), ACM, pp. 190–200.

[55] MANGARD, S., OSWALD, E., AND POPP, T. *Power Analysis Attacks – Revealing the Secrets of Smart Cards*. Springer, 2007.

[56] MANTEL, H., AND STAROSTIN, A. Transforming Out Timing Leaks, More or Less. In *European Symposium on Research in Computer Security – ESORICS 2015* (2015), vol. 9326 of *LNCS*, Springer, pp. 447–467.

[57] MANTEL, H., WEBER, A., AND KÖPF, B. A Systematic Study of Cache Side Channels Across AES Implementations. In *Engineering Secure Software and Systems – ESSoS 2017* (2017), vol. 10379 of *LNCS*, Springer, pp. 213–230.

[58] MELLOR-CRUMMEY, J. M., AND LEBLANC, T. J. A Software Instruction Counter. In *Architectural Support for Programming Languages and Operating Systems – ASPLOS 1989* (1989), ACM Press, pp. 78–86.

[59] MOLNAR, D., PIOTROWSKI, M., SCHULTZ, D., AND WAGNER, D. A. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Information Security and Cryptology – ICISC 2005* (2005), vol. 3935 of *LNCS*, Springer, pp. 156–168.

[60] NEVE, M., AND SEIFERT, J. Advances on Access-Driven Cache Attacks on AES. In *Selected Areas in Cryptography – SAC 2006* (2006), vol. 4356 of *LNCS*, Springer, pp. 147–162.

[61] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006* (2006), vol. 3860 of *LNCS*, Springer, pp. 1–20.

[62] PAGE, D. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. *IACR Cryptology ePrint Archive 2005* (2005), 280.

[63] PASAREANU, C. S., PHAN, Q., AND MALACARIA, P. Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *Computer Security Foundations – CSF 2016* (2016), IEEE Computer Society, pp. 387–400.

[64] PERCIVAL, C. Cache Missing for Fun and Profit, 2005. Technical report: `http://www.daemonology.net/hyperthreading-considered-harmful/`. Accessed: 2018-05-29.

[65] PESSL, P., GRUSS, D., MAURICE, C., SCHWARZ, M., AND MANGARD, S. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium 2016* (2016), USENIX Association, pp. 565–581.

[66] REBEIRO, C., SELVAKUMAR, A. D., AND DEVI, A. S. L. Bitslice Implementation of AES. In *Cryptology and Network Security – CANS 2006* (2006), vol. 4301 of *LNCS*, Springer, pp. 203–212.

[67] REPARAZ, O., BALASCH, J., AND VERBAUWHEDE, I. Dude, is my code constant time? In *Design, Automation & Test in Europe – DATE 2017* (2017), IEEE, pp. 1697–1702.

[68] RESHEF, D. N., RESHEF, Y. A., SABETI, P. C., AND MITZENMACHER, M. M. An Empirical Study of Leading Measures of Dependence. *CoRR abs/1505.02214* (2015).

[69] SHI, J., SONG, X., CHEN, H., AND ZANG, B. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Dependable Systems and Networks Workshops – DSNW* (2011), IEEE, pp. 194–199.

[70] SHIH, M., LEE, S., KIM, T., AND PEINADO, M. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Network and Distributed System Security Symposium – NDSS 2017* (2017), The Internet Society.

[71] SONG, D. X., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *International Conference on Information Systems Security – ICISS 2008* (2008), vol. 5352 of *LNCS*, Springer, pp. 1–25.

[72] STANDAERT, F. How (not) to Use Welch's T-test in Side-Channel Security Evaluations. *IACR Cryptology ePrint Archive 2017* (2017), 138.

[73] SUMNER, W. N., AND ZHANG, X. Memory indexing: canonicalizing addresses across executions. In *Foundations of Software Engineering – FSE 2010* (2010), ACM, pp. 217–226.

[74] SUMNER, W. N., AND ZHANG, X. Identifying execution points for dynamic analyses. In *Automated Software Engineering – ASE 2013* (2013), IEEE, pp. 81–91.

[75] SUMNER, W. N., ZHENG, Y., WEERATUNGE, D., AND ZHANG, X. Precise calling context encoding. In *International Conference on Software Engineering – ICSE 2010* (2010), ACM, pp. 525–534.

[76] SZEFER, J. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *IACR Cryptology ePrint Archive 2016* (2016), 479.

[77] TROMER, E., OSVIK, D. A., AND SHAMIR, A. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology 23* (2010), 37–71.

[78] TURNER, P. Retpoline: a software construct for preventing branch-target-injection, 2018. `https://support.google.com/faqs/answer/7625886`. Accessed: 2018-05-29.

[79] VAN DE POL, J., SMART, N. P., AND YAROM, Y. Just a Little Bit More. In *Topics in Cryptology – CT-RSA 2015* (2015), vol. 9048 of *LNCS*, Springer, pp. 3–21.

[80] WANG, S., WANG, P., LIU, X., ZHANG, D., AND WU, D. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *USENIX Security Symposium 2017* (2017), USENIX Association, pp. 235–252.

[81] WANG, Z., AND LEE, R. B. New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks. In *International Symposium on Computer Architecture – ISCA 2007* (2007), ACM, pp. 494–505.

[82] WEISER, S., SPREITZER, R., AND BODNER, L. Single Trace Attack Against RSA Key Generation in Intel SGX SSL. In *ASIA Conference on Information, Computer and Communications Security – AsiaCCS 2018* (2018), ACM.

[83] WELCH, B. L. The generalization of student's problem when several different population varlances are involved. *Biometrika 34*, 1-2 (1947), 28–35.

[84] XIAO, Y., LI, M., CHEN, S., AND ZHANG, Y. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *Conference on Computer and Communications Security – CCS 2017* (2017), ACM, pp. 859–874.

[85] XIN, B., SUMNER, W. N., AND ZHANG, X. Efficient Program Execution Indexing. In *Programming Language Design and Implementation – PLDI 2008* (2008), ACM, pp. 238–248.

[86] XU, Y., CUI, W., AND PEINADO, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy – S&P 2015* (2015), IEEE Computer Society, pp. 640–656.

[87] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium 2014* (2014), USENIX Association, pp. 719–732.

[88] YAROM, Y., GENKIN, D., AND HENINGER, N. CacheBleed: A Timing Attack on OpenSSL Constant-time RSA. *J. Cryptographic Engineering 7* (2017), 99–112.

[89] ZANKL, A., HEYSZL, J., AND SIGL, G. Automated Detection of Instruction Cache Leaks in Modular Exponentiation Software. In *Smart Card Research and Advanced Applications – CARDIS 2016* (2016), vol. 10146 of *LNCS*, Springer, pp. 228–244.

[90] ZHOU, Z., REITER, M. K., AND ZHANG, Y. A Software Approach to Defeating Side Channels in Last-Level Caches. In *Conference on Computer and Communications Security – CCS 2016* (2016), ACM, pp. 871–882.

[91] ZHUANG, X., ZHANG, T., AND PANDE, S. HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. In *Architectural Support for Programming Languages and Operating Systems – ASPLOS 2004* (2004), ACM, pp. 72–84.