MicroWalk: A Framework for Finding Side Channels in Binaries

Jan Wichelmann
j.wichelmann@uni-luebeck.de
Universität zu Lübeck
Lübeck, Germany

Ahmad Moghimi
amoghimi@wpi.edu
Worcester Polytechnic Institute
Worcester, MA, USA

Thomas Eisenbarth
thomas.eisenbarth@uni-luebeck.de
Universität zu Lübeck & WPI
Lübeck, Germany

Berk Sunar
sunar@wpi.edu
Worcester Polytechnic Institute
Worcester, MA, USA

ABSTRACT

Microarchitectural side channels expose unprotected software to information leakage attacks where a software adversary is able to track runtime behavior of a benign process and steal secrets such as cryptographic keys. As suggested by incremental software patches for the RSA algorithm against variants of side-channel attacks within different versions of cryptographic libraries, protecting security-critical algorithms against side channels is an intricate task. Software protections avoid leakages by operating in constant time with a uniform resource usage pattern independent of the processed secret. In this respect, automated testing and verification of software binaries for leakage-free behavior is of importance, particularly when the source code is not available. In this work, we propose a novel technique based on Dynamic Binary Instrumentation and Mutual Information Analysis to efficiently locate and quantify memory based and control-flow based microarchitectural leakages. We develop a software framework named *MicroWalk* for side-channel analysis of binaries which can be extended to support new classes of leakage. For the first time, by utilizing *MicroWalk*, we perform rigorous leakage analysis of two widely-used closed-source cryptographic libraries: *Intel IPP* and *Microsoft CNG*. We analyze 15 different cryptographic implementations consisting of 112 million instructions in about 105 minutes of CPU time. By locating previously unknown leakages in hardened implementations, our results suggest that *MicroWalk* can efficiently find microarchitectural leakages in software binaries.

KEYWORDS

microarchitectural leakage, constant time, side channel, cache attacks, mutual information, binary instrumentation, cryptographic implementations, dynamic program analysis

ACSAC '18, December 3–7, 2018, San Juan, PR, USA

ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274741

ACM Reference format:
Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. *MicroWalk*: A Framework for Finding Side Channels in Binaries. In *Proceedings of 2018 Annual Computer Security Applications Conference, San Juan, PR, USA, December 3–7, 2018 (ACSAC '18),* 13 pages.
https://doi.org/10.1145/3274694.3274741

1 INTRODUCTION

Side-channel attacks exploit information leakage through physical behavior of computing devices. The physical behavior depends on the processed data. The resulting data-dependent patterns in physical signals such as power consumption, electromagnetic emanations or timing behavior can be analyzed to extract secrets such as cryptographic keys [19, 33, 50, 59]. Despite the physical proximity requirement for most physical attacks, there exist remotely exploitable side channels such as microarchitectural attacks [32].

Microarchitectural attacks exploit shared hardware features such as cache [13, 65, 67], branch prediction unit (BPU) [2], memory order buffer (MOB) [61] and speculative execution engine [49] to extract secrets from a process executed *on the same system*. These attacks can be mounted remotely or locally on systems where untrusted entities can execute code on a shared hardware, either because the system is shared or untrusted code is executed. Scenarios include but are not limited to cross-VM attacks in the cloud environment [44, 57], drive-by JavaScript trojans inside the browser sandbox [54], attacks originating from untrusted mobile applications [55] and system-adversarial attacks against Intel Software Guard eXtensions (SGX) [20, 62]. Microarchitectural leakage can be used to break software implementations of cryptographic schemes where the adversaries recover the secret key by combining the leaked partial information from key-dependent activities [12, 68, 83]. These side channels can be further exploited to violate user's privacy through activity profiling [37], or to steal user's keystrokes [35]. Memory protections such as Address Space Layout Randomization (ASLR) can be bypassed by exploiting microarchitectural side-channel leakages [30].

Defense against microarchitectural side channels have been proposed based on new hardware design [24, 47], systematic mitigation [56] and activity monitoring [18, 87]. However, the most widely-used protection against microarchitectural leakage is software hardening using constant-time programming techniques [17, 39]. In this context, constant-time programming implies using microarchitectural resources in a secret-independent fashion. Therefore,

timing, or trace-based leakages [26] in the hardware would not reveal any information about the secret. These techniques depend on the underlying microarchitecture and side-channel knowledge, i.e. software implementations are hardened to follow a constant-time behavior based on published attacks on the target microarchitecture. Consequently, a novel microarchitectural attack demands new changes to these software protections. While true constant-time code avoids such problems, manual verification of the software implementation for constant-time behavior is an error-prone task, and it requires extensive, and ever growing knowledge of side channels. Besides, what we observe in the source code is not always what is executed on the processor [72], and there are leakages in the program binary that remain unobserved in the source code [46]. The state of art tools and techniques for automated finding of side-channel leakages in software binaries fall short in practice, particularly when the source code is not available. As a result, commercial cryptographic products such as *Microsoft Cryptography API Next Generation (CNG)*, which is used everyday by millions of users, have never been externally audited for side-channel security.

1.1 Our Contribution

We propose a leakage detection technique, and develop a framework named *MicroWalk* to locate leakages within software binaries. We apply *MicroWalk* to analyze two commercial closed-source cryptographic libraries hardened toward constant-time protections and report previously unknown vulnerabilities, in summary:

- We propose a technique based on Dynamic Binary Instrumentation (DBI) and Mutual Information (MI) Analysis to locate memory based and control-flow based microarchitectural leakages in software binaries.
- We develop the *MicroWalk* framework to perform automated leakage testing and quantification based on our technique. Our framework can be extended to locate other and new types of microarchitectural leakages.
- We demonstrate the ease-of-use of *MicroWalk* by showing how it significantly eases the analysis of binary code even in cases where source code is not accessible to the analyst.
- We apply *MicroWalk* to cryptographic schemes implemented in *Microsoft CNG* and *Intel IPP*, which are both widely used, yet closed source crypto libraries. Our results include previously unknown leakages in these libraries.
- We perform analysis and quantification of the critical leakages, and discuss the security impact of these leakages on the relevant cryptographic schemes.

1.2 Analysis Setup and Targeted Software

Our machine for analysis is a Dell XPS 8920 machine with Intel(R) Core i7-7700 processor, 16 GB of RAM and a traditional hard disk drive running *Microsoft Windows 10*. The *MicroWalk* Framework uses *Pin* v3.6 as the DBI backend, and *IDA Pro* v6.95 for binary visualization and leakage analysis. The tested cryptographic modules are *Microsoft bcryptprimitives.dll* v10.0.17134.1 as part of *Microsoft CNG*, and *Intel IPP* v2018.2.185.

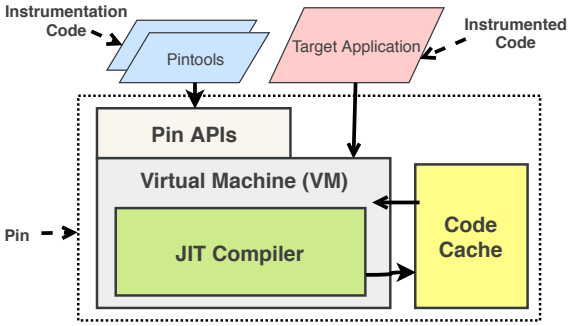

Figure 1: Pin: The JIT compiler combines application and instrumentation codes, and it stores the transformed binary in code cache. The virtual machine maintains and tracks program states, while it executes from the code cache.

2 BACKGROUND

2.1 Dynamic Binary Instrumentation

Dynamic program analysis is more accurate compared to static analysis due to availability of real system states and data [63]. Dynamic analysis requires instrumentation of the program binary, and it analyzes the program when it executes. The instrumentation code is added to the program binary without changing the normal logic and execution flow of the program under analysis, and it contains minimal instructions and subroutines for collecting metadata and measurements. The instrumentation code and the instrumented code execute at the same time following each other. Indeed, adding instrumentation is easier during the compilation phase and when the source code is available [53], but source code is not always available, and the analysis would not be as accurate due to compiler transformations. Thanks to Dynamic Binary Instrumentation (DBI) frameworks such as Pin [58], it is possible to instrument program binaries without source code.

Pin is a DBI framework based on just-in-time (JIT) compilation. In general, JIT compilers transform a source language to executable binary instructions at runtime. Figure 1 shows how an embedded JIT engine is part of *Pin* to recompile the binary instructions at runtime and combine the program's instruction with instrumentation codes, named *Pintools*. To avoid the performance pitfall of JIT compilation, *Pin* uses a `code cache` that stores the combined code, and re-execution of the same basic blocks occur from the `code cache`. Binary instrumentation using *Pintools* gives us an easy to use interface to collect runtime metadata about program states such as the accessed memory addresses, targets of indirect branches and memory allocations. *Pin* makes sure the instrumentation is transparent, i.e., it preserves the original application behavior [58]. These events can be measures as accurate as they occur on the OS and the processor and as it would be an uninstrumented execution. In terms of microarchitectural analysis, we can observe the program behavior and resource usage as they appear on the hardware, and this gives us the ability to model a known microarchitectural leakage based on the observation of states from a real system.

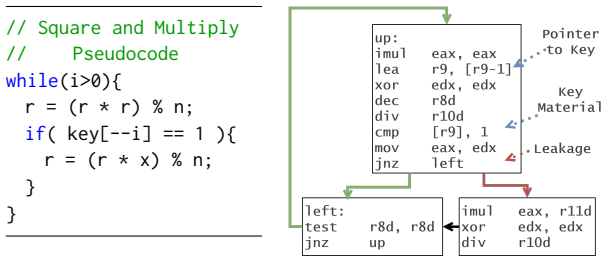

Figure 2: Montgomery `Square and Multiply` operations can leak information about the secret exponent. While `r9` points to the exponent in memory, comparison of a value from the exponent determines if the left jump should occur which leaves a key-dependent microarchitectural footprint.

2.2 Microarchitectural Leakage

Modern microarchitectures feature various shared resources, and these resources are distributed among malicious and benign processes with different permissions. A malicious process, sharing the same hardware, can cause resource contention with a victim and measure the timing of either the victim or herself to learn about the victim's runtime. In a cache attack, the adversary accesses the same cache set that the victim's security-critical memory accesses are mapped to, and she measures the memory accesses' timing. A slow memory access reveals some information about the address bits of the victim's memory access. As motivated by cache attacks on AES [13, 65], knowledge of secret-dependent memory accesses such as S-Box operations leaks information about the internal runtime state, and this information can be used for cryptanalysis and secret key recovery. In cache attacks, the size of each cache block is 64 B which stops adversaries from gaining information about the $\log_2(64) = 6$ least significant address bits. While some constant-time software countermeasures assume that the adversary cannot leak these bits, there are microarchitectural attacks on `cache banks` and `MOB` that leak beyond this assumption [61, 84]. In this work, we consider all secret-dependent memory accesses and treat them as memory-based leakages disregarding their spatial resolution.

Memory operations are not the only source of leakage. A conditional statement, or a processing loop that depends on a secret to choose an execution path can leak information about the secret. Each unique execution path operates on a different set of instructions, and it consumes the shared resources uniquely. Shared resources such as `instruction cache` and `BPU` leak information about the state of branches [1, 3]. Figure 2 resembles a classical side-channel leakage in RSA Montgomery modular exponentiation. This algorithm processes a secret exponent one bit at a time, and it performs an additional arithmetic operation when the secret bit is one. An adversary who is able to track the execution of the left branch is able to determine the secret value that affected the conditional jump decision. We treat all the attacks that are triggered due to secret-dependent branches as control-flow based attacks.

2.3 Mutual Information Analysis

Mutual information (MI) measures the mutual dependence of two random variables, and it can be used to quantify the average amount of obtainable information about one variable through observation of the second variable [36]. Mutual information using Shannon entropy is defined as

$$I(X,Y) = \sum_{x\in X}\sum_{y\in Y} p(x,y)\log_2\left(\frac{p(x,y)}{p(x)p(y)}\right),$$

where $p(x)$ and $p(y)$ are the probability distributions of random variables X and Y respectively. $p(x,y)$ is the joint probability of X and Y, and $I(X,Y)$ tells us the average amount of dependent information in bits[1] between the variables X and Y. MI has been utilized to quantify side-channel security [10, 43, 75, 86], or to mount side-channel attacks [34]. Redefining MI in the side-channel context, we can define variable X as the secret and variable Y as an internal physical state of a system leaked through a side channel. $I(X,Y)$ will measure the average amount of leakage from secret X, through observing the side-channel information Y.

2.4 Signing Algorithms

2.4.1 DSA. Digital Signing Algorithm (DSA) [69] is a signature scheme based on the discrete logarithm problem (DLP) [48]. Choosing a prime p, another prime q divisor of $p-1$, the group generator g, a secret key x, the public key $y = g^x \bmod p$, and the hash of the message to be signed z, the *DSA* signing operation is defined as

$$k \leftarrow RANDOM \mid 1 < k < q$$
$$r = (g^k \bmod p) \bmod q,\ s = k^{-1}(z + r\cdot x) \bmod q$$

where (r,s) are the output signature pairs.

2.4.2 ECDSA. Elliptic-Curve DSA (ECDSA), as an analogue of *DSA*, is a signature scheme based on elliptic curves [45], in which the subgroup of a prime p is replaced by the group of points on an elliptic curve over a finite field. Choosing an elliptic curve, a point on the curve G, the integer order n of G, a secret key d_A, the public key $Q_A = d_A \times G$, and the hash of message to be signed z, the ECDSA signing operation is defined as

$$k \leftarrow RANDOM \mid 1 < k < n-1$$
$$(x_1, y_1) = k \times G$$
$$r = x_1 \bmod n,\ s = k^{-1}(z + r\cdot d_A) \bmod n$$

Both *DSA* and *ECDSA* use an ephemeral secret k that needs to be chosen randomly for each operation.

2.4.3 Modified Elliptic Curve Signature. Elliptic-Curve Nyberg-Rueppel (ECNR) [64] and *SM2* [8], a standard signature scheme, are modified schemes based on *ECDSA* that allow signatures with message recovery. *ECNR* and *SM2* are widely used, and they are both supported by *Intel IPP*. Public parameters, the private/public key pair and the ephemeral secrets are chosen similar to *ECDSA*. The pair (x_1, y_1) is also calculated similarly, but the signature generation for *ECNR* is defined as

$$r = x_1 + z,\ s = k - r\cdot d_A,$$

and the signature generation for *SM2* is defined as

$$r = x_1 + z,\ s = (1 + d_A)^{-1}(k - r\cdot d_A)$$

[1] $\log_2$ measures the MI in *bit* unit.

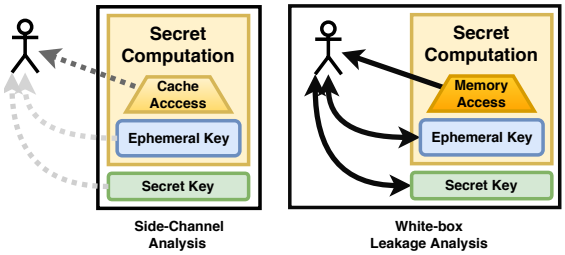

Figure 3: Left: side-channel analyst finds relationship between a real leakage such as cache access pattern and secrets such as cryptographic keys. Right: *MicroWalk* follows a white-box model where the security analyst has full access to runtime states such as memory accesses, and she can find dependencies between arbitrary secrets and internal states.

3 *MICROWALK* ANALYSIS TECHNIQUE

MicroWalk aims to find microarchitectural leakages in software binaries. A binary implementation is vulnerable to microarchitectural side-channel attacks when there is a dependency between a secret and internal computation states observable through the side channels. We expose such relationships and quantify the amount of observable leakage in these implementations. This helps security analysts **1)** to reveal whether an implementation has leakages, **2)** to locate the exact location of each leakage in the binary, and **3)** to measure the dependency between the secret and the internal state, i.e., it can give some confidence value on the severity of the leakage. In contrast to side-channel analysis model, we are able to perform this analysis in a white-box model.

3.1 Leakage Analysis Model

We assume a strong adversary with full access to runtime events such as memory accesses, execution path and even register values. Further, the adversary can choose and modify any secret input of the system. This strong adversary can define any internal computation state such as addresses of memory accesses and register values as a potential leakage vector, based on her knowledge of a category of side-channel attacks, e.g., memory based attacks (Section 2.2). The adversary executes the system under her full control, and feeds the system with arbitrary secrets while collecting runtime traces for the defined leakage vector. Figure 3 compares our leakage analysis model with the side-channel analysis model. As an example, if we try to analyze a binary implementation of AES, we need to define certain operations as our leakage vectors. Based on cache attacks, an adversary defines memory accesses as a leakage vector, and she collects all memory accesses during the execution of AES using arbitrary secret keys. If there is a dependency between different secret keys and the variation of memory accesses, the adversary can locate which instructions relate to any secret-dependent memory accesses, and identify potential leakages.

3.2 Capturing Internal States

We choose two common sources of leakage as our leakage vectors: **1)** execution path and **2)** memory accesses. A true constant-time implementation follows a linear execution path for any given secret input; each time a software performs a secret-dependent conditional branch, it leaks some amount of information about the secret. Defining execution path as a leakage vector helps us to check whether for any secret input the same operations are performed. The second common source of leakage are memory accesses. A constant-time implementation should follow a secret-independent memory access pattern. If, for example, an implementation of a cryptographic algorithm does key-dependent table look ups which can be exploited by measuring cache timings, an attacker will be able to extract parts of the secret key; we ensure that memory accesses are either invariant, or at least uncorrelated to the input (e.g. blinding in RSA [50]).

To be able to detect these two types of leakages, we need to collect the internal state for all memory accesses and branch operations. First of all, we generate a set of arbitrary inputs for a chosen secret. These inputs can be either random (e.g. plain texts for encryption) or have a special structure with some random components (e.g. private keys or ephemeral secrets). We then execute the target binary on each input and log the following events:

- memory allocations
- branches, calls and returns
- memory reads and writes
- stack operations.

Absolute memory addresses may vary even for constant-time programs, e.g., due to ASLR and dynamic heap allocation. We use the trace of memory allocations and stack operations to compute relative memory addresses; our meta data then consists of a list of relative addresses for memory accesses and the branches to, from and within the code we are analyzing. Note that one can define other leakage sources based on the underlying microarchitecture and collect the state of relevant instructions for analysis, e.g., the `multiplication` on some *ARM* platforms leaks information [7].

3.3 Preparing State Variables

We do not make any assumptions on the leakage granularity; compared to similar techniques, that stop at cache line level, we keep this parameter freely configurable. This has the advantage that the analysis can be restricted to leakage sizes that are actually relevant to the analyst: For example, as of writing this paper, on Intel processors the finest known attack has a leakage granularity of 4 bytes [61]. Applying our technique in 1-byte mode will give all positions where a leakage might occur, but if one only expects 4-byte leakages to be exploitable, this may yield some false positives. Instead, the security analyst can choose the leakage granularity that fits to the desired spatial resolution. After applying the chosen leakage granularity of $g \in \mathbb{N}$ bytes by discarding the lower $\log_2 g$ bits of each address, we can acquire an efficient representation of a specific execution state by computing a hash value of all or a subset of the trace entries; a truly constant-time program should have identical hashes of the full trace for every secret input. If we are only interested in analyzing individual instructions, e.g., memory access leakages of a specific subroutine, we can as well just compute the hash for the subset of traces for a single instruction.

3.4 Leakage Analysis

Our approach identifies any variations resulting from unique inputs and captured internal states per input. A naive approach is to compare the collected traces and divide them into classes. Observing

more than one class informs us about secret-dependent operations. One can also compare raw traces sequentially which outlines all positions where the program behaves input-dependent and thereby allows to isolate the problematic sections. In addition to these simple approaches, we use MI to detect/locate these leakages, and to quantify the observable information.

To simplify MI analysis, we assume that X is a set of unique uniformly distributed input test cases, which trigger deterministic behavior of the investigated program. If the program makes use of randomization (e.g. blinding in RSA [50]), the test cases $x \in X$ should contain the corresponding sources of randomness too.

Let Y be a set of possible internal states (e.g. hashes of execution traces). We then define the execution state $T_i \subset X \times Y$ of the analyzed program at time point i as

$$(x, y) \in T_i \wedge (x, y') \in T_i \Rightarrow y = y',$$

i.e. each test case $x \in X$ appears at most once in T_i. The probability of one observed state $y \in Y$ is

$$p_i(y) = \frac{|\{(x', y') \in T_i \mid y = y'\}|}{|T_i|}.$$

For the probability of pairs $(x, y) \in X \times Y$ we get

$$p_i(x, y) = \begin{cases} \frac{1}{|X|} & \text{if } (x, y) \in T_i, \\ 0 & \text{else,} \end{cases}$$

since each input and therefore each input/state tuple occur exactly once: $|T_i| = |X|$.

With this knowledge we can finally compute the mutual information between test cases X and the set of all occurring states $Y_i := \{y \mid (x, y) \in T_i\}$:

$$\begin{aligned} I_i(X, Y_i) &= \sum_{(x,y)\in T_i} \frac{1}{|X|} \log_2 \left(\frac{\frac{1}{|X|}}{\frac{1}{|X|} \cdot \frac{|\{(x', y') \in T_i \mid y = y'\}|}{|T_i|}} \right) \\ &= \sum_{(x,y)\in T_i} \frac{1}{|X|} \log_2 \left(\frac{|T_i|}{|\{(x', y') \in T_i \mid y = y'\}|} \right). \end{aligned}$$

3.5 Interpretation of MI Score

As mentioned before, we can compute the MI for the entire trace, or a single instruction. A non-zero score for whole-trace MI tells us that an implementation has leakages, but it cannot locate the leakage point, and an implementation that has multiple leakage points over the execution period will have an aggregated MI value. The MI for single instructions is more precise, in which we can locate the instructions with positive score. The MI score $I_i(X_i, Y_i)$ is bounded by the amount of input bits $\log_2 |X|$, and (for instruction MI) by the operand size: For example, an instruction that once accesses memory depending on 8 bits of the input will generate MI $\min\{8, \log_2 |X|\}$. If we only execute $|X| = 128$ test cases, we get MI score 7; for 256 or more test cases we get MI score 8. The analyzed MI score is an estimate of the average leakage over the given test cases. MI is the appropriate metric in cases where the analyzed inputs are not under the attackers control and commonly used in leakage quantification. Alternatively, the *worst case* leakage for any attacker-chosen input is given by the *min entropy*, which only considers the most likely guess. The use of min entropy instead of MI in *MicroWalk* is recommended if the adversary has full control over the inputs and specific high-leakage inputs exist [74].

4 *MICROWALK* FRAMEWORK

The *MicroWalk* framework is built as a pipeline with separate stages for test case generation, tracing and analysis (Figure 4). This modular design reduce the complexity, leading to easier extensibility: If one wants to implement additional analysis techniques apart from the ones that we already provide, she can directly add a new analysis stage, without needing to touch other parts like the trace generation. We will continue explaining these stages in more detail.

4.1 Investigated Binary

Although we are only interested in analyzing a specific function within a binary, we have to instrument and collect traces for the entire setup stage of the application before reaching to the analysis point, including the target library or executable itself, and parts of dependencies like system components. This process leads to an enormous decrease in analysis speed. A more efficient approach is to load and instrument the setup code only once, and then process the incoming test cases in a controlled loop. For libraries, we create a wrapper executable that executes an interface in a loop with new test cases. For executable applications, we can adopt in-memory fuzzing techniques where we inject hooks at the beginning and end of the target function and control the execution of the function to reset to the beginning with new test cases [11]. To separate traces of the different test cases and avoid that the loop code causes false positives, we place calls to two instrumented dummy functions `PinNotifyTestcaseStart` and `PinNotifyTestcaseEnd`, which mark the start and the end of the analyzed section. A similar approach is taken by some fuzzers like `WinAFL` [31], which use a built-in functionality of `DynamoRIO` [29] to exchange the argument list of `main` or a similar function.

4.2 Input Generation

The *MicroWalk* framework utilizes cryptographically secure pseudorandom number generators to create random test cases of any specified length. This performs well when analyzing cryptographic code, e.g. decrypting random ciphertexts. If a special input format is required, the test case generation code can be easily extended to produce such inputs, e.g. cryptographic keys in PEM format; this way, parts of the input can be kept constant while other parts are randomized, allowing to isolate the parameters which cause non-constant time behavior. Further, the framework supports passing a directory containing already generated inputs of any format.

4.3 Trace Generation

To trace the execution of individual test cases, we create a custom so-called *Pintool*, which is a client library making use of `Intel Pin`'s dynamic instrumentation capabilities. In summary, our *Pintool* logs the following events in a custom binary format on disk:

- module loads and the respective start and end addresses;
- calls to dummy functions in the instrumented executable, to identify start and end of a test case execution;

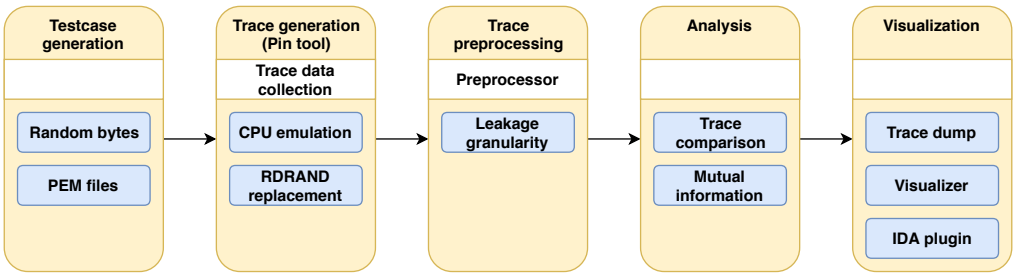

Figure 4: The *MicroWalk* pipeline: Given the software binary under test, the framework generates test cases using the selected source, that are then used to produce execution traces. These traces need to be preprocessed to extract important information. The resulting trace files can then be analyzed for leakages, which are shown to the user in the visualization stage. Each stage can be easily modified to add further functionality, that is used either interchangeably or in addition to existing features.

- sizes and addresses of allocated memory blocks through heap allocation functions such as `malloc` and `free` (Platform dependent), for resolving relative memory addresses;
- Stack pointer modifications, for resolving relative addresses;
- branches, calls and returns to and from all involved modules;
- memory reads and writes in investigated modules.

4.3.1 Instruction Emulation. Several cryptographic libraries use the `CPUID` instruction to detect the supported instructions for the respective processor and select a fitting implementation (that e.g. makes use of `AES-NI`). we enabled the *Pintool* to change the output of this instruction. This allows to test arbitrary subsets of the instruction set that is available on the computer running *Pin*.

As mentioned in Section 3, cryptographic implementations might use randomization techniques like blinding to hide correlations between secret inputs and execution, or use ephemeral secrets. Some of these rely on the `RDRAND` instruction, which provides random numbers seeded with hardware entropy [40]. We provide an option to override the output of this instruction with arbitrary fixed values to control the randomization of the program under investigation.

4.4 Trace Preprocessing

The resulting raw trace files now need some preprocessing: First we add the common trace prefix that is generated before running the first test case, and which contains allocation data from the setup phase. In a second step, we calculate relative offsets of memory addresses. This involves associating branch targets with instruction offsets in the respective libraries, and identify offsets of memory accesses, such that traces generated using the same test case but during different runs of the *Pintool* still match, regardless of the usage of randomized virtual addresses, e.g., *ASLR*. For accesses to heap memory, we need to maintain a list of all currently allocated blocks: We use a stack to match the allocation size with their respective returned memory addresses, since in some implementations the heap allocator tends to call itself to reserve memory for internal bookkeeping. Finally the resulting preprocessed trace file is much smaller than the raw one (which can be discarded after this step), saving disk space and speeding up the following analysis stage.

4.4.1 Applying Leakage Granularity. We apply the leakage granularity immediately before the analysis starts; this way the preprocessed trace files are not modified, so the analysis can be performed on the same traces with different parameters. An analysis granularity of $g = 2^b$ bytes ($b \in \mathbb{N}$) is introduced by discarding the b least significant bits of each relative address.

4.5 Leakage Analysis

We implemented three different analysis methods in our framework:

4.5.1 Analysis 1: Trace Comparison. The first analysis method implements the trace comparison technique; given two preprocessed traces, we compare them entry by entry to check whether they differ at all. This performs well for leakage detection of particularly small algorithms such as symmetric ciphers. Optionally, the user can use trace diffs to manually inspect varying sections.

4.5.2 Analysis 2: Whole-trace MI. For leakage detection of an entire logic and calculation of the average amount of input bits that might leak over arbitrary parts of the execution (assuming that the attacker has full access to the trace), we provide an option to estimate the MI between input data and resulting trace. Given a set X of unique test cases, we need to determine matching outputs for each trace prefix. Since we can compute the final MI only after waiting for completion of all test cases, it would be inefficient to store the entire trace; instead we reduce the trace data by encoding information like relative memory accesses and branch targets into 64-bit integers, and then compress them into one 64-bit integer $y \in \{0, \ldots, 2^{64} - 1\}$ using a hash function. We store the resulting tuples of inputs and hashes in sets $T_i \subset X \times \{0, \ldots, 2^{64} - 1\}$ for each prefix length i. We then apply the methods from Section 3 to measure the trace leakage.

4.5.3 Analysis 3: Single-instruction MI. The average amount of bits leaked by a single memory instruction is calculated analogously to the trace prefixes: Here, for a specific instruction i, $T_i \subset X \times \{0, \ldots, 2^{64} - 1\}$ contains hashes of the accessed memory addresses for each input x. These hashes change when the accessed addresses, their amount or their order vary, thus we get the maximum amount of information that is leaked by the respective instruction.

4.6 Manual Inspection and Visualization

To be able to manually inspect the preprocessed traces, the program has an option to convert binary traces into a readable text representation. If `MAP` files with function names are available (exported by some compilers or disassemblers), these can be used to symbolize memory addresses. We also created an IDA python plugin to import our single-instruction MI results as disassembly annotations. This helps further analysis on which parts of functions and loops leak.

Further we developed an experimental visualization tool, that renders function names and then draws an execution path. It also provides an option to render two traces simultaneously and highlight all sections where they have differences. This gives a quick overview of potential leakages and their structure.

5 CASE STUDY I: INTEL IPP

Intel's *Integrated Performance Primitives (IPP)* cryptographic library aims to provide high performance cryptographic primitives that are compatible with various generations of Intel's processor [41]. *Intel IPP* supports symmetric operations such as AES, as well as asymmetric signature and encryption schemes such as ECDSA. Intel IPP is used as the cryptographic backend for many of Intel's security products such as Intel SGX. Each of the implemented schemes in this library comes in variants optimized for different processors [42]. The dynamic library checks the supported instruction set at runtime and chooses the most optimized implementation. However, developers can statically link toward a specific implementation by choosing the proper architecture code, e.g., `n8_ippsAESInit` rather than `ippsAESInit`. In this case study, we test implementations for the variant optimized for processors supporting *Intel® Advanced Vector Extensions 2 (Intel® AVX2)* with architecture code `l9`.

5.1 Applying *MicroWalk* MI Analysis to IPP

To be able to test *Intel IPP* cryptographic implementations, we prepared wrappers that perform encryption and signing operations. For each tested implementation, we configured the wrappers for testing multiple test case scenarios: **1)** randomized plaintexts/ciphertexts to be encrypted/decrypted, or the message to be signed, **2)** randomized symmetric keys or private asymmetric keys, and **3)** random ephemeral secrets, when it is applicable, e.g., *DSA* and *ECDSA* as the input to MI Analysis. As suggested by chosen plaintext/ciphertext attacks, attacks on the cipher key and lattice attacks on ephemeral secrets [12], using these scenarios, we are able to detect leakages that are dependent on various types of secrets.

Table 1 shows the single-instruction MI analysis results, where symmetric ciphers: *(Triple) DES*, *AES* and *SM4* and asymmetric ciphers: *DSA*, *RSA*, *ECDSA*, *ECNR* and *SM2* have been tested. On our analysis setup, the total computational time to analyze 10 different implementations with about 92 million total instructions is 73 minutes of CPU time, highlighting the efficiency of our method. Note that we performed analysis with input size $2^7 = 128$ (7-bit *MI*) and input size $2^{10} = 1024$ (10-bit *MI*), for analysis of symmetric and asymmetric operations respectively. Although analysis with more iterations is possible, state-of-the art side-channel attacks on these implementations suggest that the random secret should show leakage behavior after this number of iterations. *Intel IPP* uses two separate interfaces for the key schedule, and ephemeral secret generation for most implementations (Table 1).

Table 1: Singe-instruction MI Analysis of Intel IPP cryptographic implementations v2018.2.185. All implementations are chosen from the *l9* architecture code.

Scheme	Interfaces	Executed / Unique Instructions	Analysis Time (ms)	Leakage Found
3DES/ECB	ippsDESInit ippsTDESDecryptECB	4074613 / 70205	11921	0
SM4/ECB	ippsSMS4Init ippsSMS4EncryptECB	4085517 / 68221	10004	0
AES/CTR	ippsAESInit ippsAESEncryptCTR	2138799 / 49181	27289	2
DSA (512)	ippsDLPGenKeyPair ippsDLPSignDSA	12245281 / 57423	1735153	2
RSA (512)	ippsRSA_Decrypt	43987943 / 55167	275090	1
ECDSA (SECP256R1)	ippsECCPGenKeyPair ippsECCPSignDSA	4085155 / 63785	358373	3
ECDSA (BN256)	ippsECCPGenKeyPair ippsECCPSignDSA	5383210 / 63699	750188	*
ECDSA (SM2)	ippsECCPGenKeyPair ippsECCPSignDSA	5158607 / 63741	353435	*
ECNR (SECP256R1)	ippsECCPGenKeyPair ippsECCPSignNR	4028592 / 62447	281937	2
SM2	ippsECCPSignSM2	6021005 / 64273	554035	3
Total		91208722 / 618142	73 minutes	13

* Different curves did not change the results for ECDSA.

(Triple) DES, *AES* and *SM4* are block ciphers that use table-based S-Box operations. The results suggest that these implementations are heavily protected against memory-based leakages. Our target architecture code uses the `AES-NI` instruction set for *AES* and *SM4* operations. `AES-NI` is inherently secure against known attacks. However, testing the *CTR mode* reveals some leakages. All asymmetric ciphers suffer from at least one leakage. For schemes that are based on elliptic curves such as *ECDSA*, *ECNR* and *SM2*, Intel IPP supports various standard curves. As some developers optimize curve arithmetic differently for various standard elliptic curves, we tested the *ECDSA* signing operation with three different curves: `SECP256R1`, `BN256` and `SM2`. However, the *MI* analysis results are exactly the same for different choices of elliptic curves. We found a total of 13 leakages in *Intel IPP*, while some of these leakages are triggered through calling the same subroutine, e.g., both *ECDSA* and *SM2* use the leaky subroutine for `scalar multiplication`. We will discuss these subroutines in more detail.

Table 2: Discovered leakage subroutines within Intel IPP cryptographic implementations v2018.2.185. Some of the subroutines expose critical and potentially exploitable leakages.

Subroutine	Affected	MI	Leakage Source
gfec_MulBasePoint	ECDSA, ECNR, SM2	0.86 / 10	Conditional Loop
cpMontExpBin	DSA	3.73 / 10	Conditional Loop
cpModInv	DSA, SM2, ECDSA	3.88 / 10	Conditional Loop
ExpandRijndaelKey	AES/CTR	7.00 / 7	Memory Lookup
ippsAESEncryptCTR	AES/CTR	0.13 / 7	Conditional Loop
gsMontExpWin	RSA	1.12 / 10 3.11 / 10	Conditional Loop Memory Lookup
alm_mont_inv	ECDSA, ECNR, SM2	5.33 / 10 9.98 / 10	Conditional Loop Memory Lookup

Algorithm 1 Bitmasked Montgomery Exponentiation

```
1:  procedure BinExp(base g, exponent k)
2:      A ← R mod p
3:      g̃ = MontMul(g, R^2 mod p)
4:      m ← 0
5:      i = 1
6:      while i < (BitLength(k) mod 64) do
7:          t ← A & ~m | g̃ & m
8:          A ← MontMul(A, t)
9:          m = ~m & k_i
10:         i = i + 1 − m
11:     end while
12:     for j ← 1 to BitLength(k)/64 do
13:         perform the same operations as above.
14:     end for
15:     return A
16: end procedure
```

5.2 Discovered leakages in Intel IPP

We have found 7 different subroutines that have leakages, i.e., perform data-depended memory accesses or branch decisions (Table 2). We performed an initial analysis of these leakages using our visualization tool and *IDA Pro*. The subroutine `gfec_MulBasePoint` performs scalar multiplication of a scalar and point on the elliptic curve, as a common operation in all curve-based signature schemes: *ECDSA*, *ECNR*, *SM2*. As defined by the signing algorithms Section 2.4, *gfec_MulBasePoint* leaks information about the ephemeral secret. This leakage occurs due to the dependability of the number of times the window-based multiplier loop processes the ephemeral secret. Further leakages exist in the curve operations after the scalar multiplication: The subroutine *alm_mont_inv* leaks information during the mapping of x coordinate of computed public point. As (x_1, y_1) are not secrets in the signing operation, this leakage is not critical, and we refrain from further root cause analysis. Similarly, the subroutine *cpModInv* has leakages with a relatively high MI score that is due to the secret-dependent loop count. *cpModInv* performs a modular inversion operation using Extended Euclidean Algorithm (EEA). In *ECDSA*, k^{-1} leaks information about the secret ephemeral, and in *SM2*, $(1 + d_A)^{-1}$ leaks information about the secret signing key. *ECNR* does not perform any modular inversion and is safe from leakages due to this subroutine. The existing leakage in *cpModInv* subroutine also applies to *DSA* where a modular inversion on ephemeral secret, k^{-1} can leak.

Intel IPP supports two distinct functions for performing Montgomery exponentiation. Exponentiation of big numbers is a common operation in schemes such as *RSA* and *DSA*. The *RSA* algorithm uses the *gsMonthExpWin* subroutine which is a window-based implementation of the Montgomery exponentiation. This function has leakages based on both memory lookup and conditional loop. The second Montgomery exponentiation subroutine `cpMontExpBin` is a protected binary implementation that has leakage due to the conditional loop count. *DSA* uses the latter, which leaks information about the ephemeral secret during computation of $(g^k \bmod p)$.

The only leakage exposed during testing of symmetric ciphers are due to *AES* key generation subroutine `ExpandRijndaelKey`, and calculation of the nonce length in *CTR mode*. `ExpandRijndaelKey` is called every time the `ippsAESInit` is used. As the high *MI* score shows, *AES* key schedule used during the CTR mode has full leakage. This leakage can be considered critical in scenarios such as the SGX environment where an adversary has a high resolution side channel [62, 78]. When the symmetric key is passed to the AES key schedule, a high resolution adversary can steal the secret key before any encryption/decryption. While *AES/CTR* encryption uses `AES-NI`, there is a loop within this implementation where calculating the length of nonce leaks about the leading zero bits.

5.2.1 Leakage of Scalar Multiplication. Scalar multiplication in *Intel IPP* uses a fixed-window algorithm with a window size of 5: for a 256-bit ephemeral secret, as defined by `SECP256R1`, the algorithm performs 51 iterations of the window operation. However, our dynamic analysis of the algorithm with various random ephemeral secrets shows that `gfec_MulBasePoint` skips the leading zero bits and applies fewer windows if there are leading zero bits in the beginning, as the multiple of the window size. In this case, the main loop performs 50 times for 2, 49 for 7 and 48 times for 12, etc, leading zero bits. *CacheQuote* [25] exploits a similar vulnerability used by *Intel EPID* signature scheme, but *EPID* uses a different function of *Intel IPP* for scalar multiplication `cpEcGFpMulPoint`. As our discovery suggests, this was a common issue in *Intel IPP* that was existed among other curve implementations. Although this implementation has countermeasure based on Scatter-Gather technique [17], this vulnerability can easily be exploited in high resolution settings using a lattice attack [25].

5.2.2 Bitmasked Montgomery Exponentiation. The Montgomery exponentiation in *Intel IPP* follows a bit-by-bit operation based on the Montgomery Reduction technique [38]. However, the implementation is protected by obfuscating the conditional statements as bit-masked operations. Therefore, the subroutine always executes the same `Montgomery multiplication (MontMul)` subroutine disregarding the value of the exponent bits. However, the exponent bits are used as a mask to choose the operand of the `MontMul` and to execute the `MontMul` two times with two different operands

when the exponent bit is one. Although this implementation looks secure at first sight, the exponent bits are used **1)** to calculate the exponent bit length, i.e., leading zero-bit leakage, and **2)** to decide the number of iterations of the loop. Based on Algorithm 1, the main loop executes two times if an exponent bit is one and once if the exponent bit is zero. This leaks the Hamming weight of the ephemeral secret to a microarchitectural adversary.

Further, the algorithm performs a similar operation with separate instructions for different parts of the key. For example, for a 160-bit DSA exponent, the algorithm first processes the first 32 bits, and then another code section processes the remaining 128 bits of exponent. This gives an adversary a local Hamming weight leakage of the first 32-bit of the secret exponent.

6 CASE STUDY II: MICROSOFT CNG

The *Cryptography API: Next Generation (CNG)* is the cryptography platform supplied with every Windows system beginning with Windows Vista, and replaces the older *CryptoAPI* as the default cryptographic stack. It includes many common algorithms, including *RSA*, *AES*, *ECDSA*. While the public API for *Microsoft CNG* resides in the `BCrypt.dll` system file, its cryptographic implementations themselves are located in another library file, `BCryptPrimitives.dll`. Microsoft does provide neither source code nor documentation for the internal functionality, but one can download `PDB` symbol files from Microsoft's symbol server, which contain most of the internal function names, helping to reduce the reverse engineering effort.

6.1 Applying *MicroWalk* MI Analysis to CNG

As we did with IPP, we again created wrapper executables to call the respective library functions of *RSA*, *DSA*, *ECDSA* and *AES/ECB*. For *AES*, the library uses the `CPUID` instruction to choose between two different implementations, one that uses `AES-NI` vector instructions, and a plain T-table based implementation. We tested both implementations by emulating the `CPUID` instruction, as explained in Section 4.3.1. The results are shown in Table 3. We analyzed a total of 21 million instructions in 31 minutes of CPU time, finding four different leakage points. For *RSA*, we discovered that Microsoft's implementation behaves truly constant-time. *ECDSA* and *DSA* implementations both suffer from leakage due to calling the same subroutine for modular inversion.

Table 3: Singe-instruction MI Analysis of some of the bcryptprimitives.dll v10.0.17134.1 cryptographic implementations.

Scheme	Interfaces	Executed / Unique Instructions	Analysis Time (ms)	Leakage Found
AES/ECB	SymCryptAesEcbEncrypt	2384298 / 55451	17546	0
AES/ECB	SymCryptAesEcbEncryptAsm	2324391 / 63179	26211	2
DSA (512)	MSCryptDsaSignHash	3586162 / 63748	223356	1
RSA (1024)	MSCryptRsaDecrypt	8073605 / 66454	760450	0
ECDSA (SECP256R1)	MSCryptEcDsaSignHash	4764783 / 64732	831136	1
Total		21133239 / 313564	31 minutes	4

Table 4: Discovered leakage subroutines within bcryptprimitives.dll v10.0.17134.1 cryptographic implementations.

Subroutine	Affected	MI	Leakage Source
SymCryptFdefModInvGeneric	DSA, ECDSA	10.00 / 10	Conditional Loop
SymCryptAesEncryptAsmInternal	AES	7.96 / 10	Memory Lookup

6.2 Discovered leakages in Microsoft CNG

Analyzing the aforementioned algorithms yielded two leakage candidates (see Table 4); the first one resides within the modular inversion function of *DSA* and *ECDSA* and is used for all processors. The MI returns full leakage for the modular inversion leakage, implying that the implementation is heavily unprotected. The second one is in the encryption function of *AES* and only used by processors not supporting `AES-NI`. As it is a table-based implementation, the leakage is expected.

6.2.1 Leakage of Modular Inversion. The modular inversion function that is used for *DSA* and *ECDSA* gives full *MI* on 1024 signing operations for random ephemeral secrets with fixed key and plaintext. This subroutine does not have any constant-time protection. However, while this is a non-constant time behavior and suggests that the ephemeral leaks, we considered this as not exploitable; Microsoft protects this implementation through a masking countermeasure. The masking countermeasure for modular inversion works as follow:

(1) A mask value m is generated randomly.
(2) The ephemeral secret k is multiplied by m before the modular inversion: $s = (k \cdot m)^{-1}(z + x) \bmod q$
(3) Then the signature s is multiplied again with m to produce the correct signature:

$$s = sm = (k \cdot m)^{-1}(z + x) \cdot m = k^{-1}(z + x)$$

Thus, the implementation leaks $k \cdot m$, where m is a random per-signature generated mask, effectively preventing extraction of useful information. Leakage of ephemeral keys is exploitable [12], the randomized product of ephemeral key and a random value is not.

6.2.2 Leakage of AES T-table Lookup. The non-vector version of *AES* uses a common lookup table implementation, where four so-called *T-tables* combine the steps `SubBytes`, `ShiftRows` and `MixColumns`. Each round consists of four of such lookups per table, leading to $16 \cdot r$ memory accesses per encryption, where $r \in \{10, 12, 14\}$ is the number of rounds. The 8-bit indices used for the table accesses depend on the plaintext and the key; since the *MI* is 7.96 for 1024 measurements, these indices can be considered fully leaking. Each table entry has 4 bytes size, thus each T-table has 1024 bytes, and therefore takes 16 cache lines on an Intel processor;

such implementations have already been shown to be exploitable with cache attacks [13].

7 RELATED WORK

Programming languages can support constant-time code generation and verification [16, 21, 73]. The general approach is to support annotation of security-critical variables and to generate instructions that operate obliviously on annotated secrets. Annotated secrets can be verified for constant-time behavior using SMT-based techniques [16]. Constant-time behavior can be enforced for some operations by using primitives such as oblivious RAM (ORAM) [76] and obfuscated execution [70]. Language-based approaches are not widely used, and annotation is an error-prone task.

Black-box testing approaches use statistical methods to quantify leakages of physical channels [23]. In particular, *Dudect* [71] performs black-box timing analysis, in which the timing of a target system with different inputs will be analyzed using the *t-test* [80], but these black-box techniques do not scale to microarchitectural attacks with a gray-box model. With an abstract model of the leakage channel, methods based on **Static Program Analysis** are proposed to analyze program code and to quantify leakages [4, 5, 9, 14, 51]. Similar to language-based approaches, these techniques are limited to correct annotation of the source code. While some of these approaches are limited to the source code and cannot find leakages that are potentially introduced by the compiler [5], others perform the analysis on the lower level LLVM bitcode [4] or the annotated machine code [9, 14]. However, they rely on the availability of the source code. *CacheAudit* [27, 28] is based on Static Binary Analysis (SBA). SBA approaches need to initially reconstruct the original basic blocks and control flow graph. Precise reconstruction of the program semantic and control flow graph is infeasible without the runtime information, by just using static disassembly [6]. As a result, while they give formal guarantees on the absence of leakages, they do not scale to accurately analyze large program binaries, e.g., *CacheAudit* approach has only been tested on rather simple algorithms such as sorting and symmetric encryption. Other proposals based on **Symbolic execution** quantify side-channel leakage by determining symbolic secret inputs that affect the runtime behavior [22, 66]. However symbolic execution is an expensive approach, and the proposed methods require access to the source code.

In this work, we leverage **Dynamic Program Analysis** techniques to accurately locate microarchitectural leakage in software binaries, as they execute on the processor. *ctgrind* [52] based on LLVM memcheck can check all branches and memory accesses to make sure that they do not have dependency on secret data. Irazoqui et al. instrument the source code to obtain and analyze cache traces using MI [43]. Sensitive code sections are identified by taint analysis. On binary-only approaches, *CacheD* [77] analyzes binaries based on symbolic execution and constraint solving. They initially use DBI to get execution traces for a set of input values; then, given the information which input values are considered secret, a taint analysis extracts all instructions that work with secrets, either directly or indirectly. These instructions are then analyzed using symbolic execution to detect whether cache leakages exist. In comparison, our method aims at maximum performance without too much loss of accuracy by only storing necessary information and using hash compression to get small execution states. The symbolic execution approach introduces a large bottleneck, as their analysis time suggest. This saving of computation time allows us to detect also other types of leakages like differing loop counts or byte-level memory access differences. Also, since *MicroWalk* is designed as a modular open source framework, one can implement arbitrary analysis stages for other types of leakages. Zankl et al. [85] use DBI to collect traces for instruction based leakage detection. They use *t-tests* for leakage analysis and only test for execution flow leakages. *STACCO* [81] is focused on differential trace analysis for Bleichenbacher attacks [15]. Independently, *DATA* [79] follows a similar approach based on DBI. They use trace differentiation and *t-tests* for leakage analysis. As of our knowledge, our work is the first that has been tested on actual closed-source binaries.

8 CONCLUSION

The lack of efficient and practical tools for leakage analysis of binaries leave the reliability of these untested deployed implementations a mystery. To be able to analyze the compiler outputs and closed-source libraries, we have created an extensible framework that supports various types of microarchitectural leakages based on `instruction and data cache`, `MOB`, `BPU`, etc. *MicroWalk* can be extended to analyze other and future side channels. Our framework leverages *DBI* to collect the internal state of a program under test, and it applies multiple analysis techniques based on trace comparison and *MI*. *MicroWalk* is open source and is publicly accessible: https://github.com/UzL-ITS/Microwalk. We used this framework to thoroughly analyze two widely used closed-source libraries, *Intel IPP* and *Microsoft CNG*. The tested implementations are optimized for the current generation of Intel processors. Our report shows that side-channel countermeasures for these implementations are still not fully leakage-free, e.g., all the curve-based signature schemes in *Intel IPP* suffer from at least one vulnerability. We have identified several leakages in symmetric and asymmetric ciphers, and reported them to the respective vendors. Our analysis shows that despite the existing efforts on protecting these implementations, some of them still suffer from security-critical leakages.

8.1 Future Work

8.1.1 Coverage-based Fuzzing. We use random test cases to get a uniform random distribution of potential memory accesses and execution paths; while this works well with cryptographic implementation, it would not scale to targets such as protocols or data structures. Coverage-based *Fuzzing*[60] is a technique to generate test cases with the aim of achieving maximum code coverage; while it was originally developed to find software bugs, e.g., memory corruption, the same approach can be applied for finding side-channel leakages, e.g., leakage in the *JPEG* library [82]. We have already implemented an experimental support for using `WinAFL`[31] as a test case generator; in that setting `AFL` helps to generate samples with higher coverage, while at the same time the test cases are sent to our framework for further processing. It is desirable to enhance this experimental feature and apply it to non-cryptographic implementations that are critical in terms of side-channel security.

8.1.2 Distinguishing leakages in call graph. We observed that in some cases control flow leakages in the higher level algorithm

residing at the top of the call chain hide leakages in the subroutines invoked in deeper levels. Also, if separate functions use a common subroutine, a positive MI result in this subroutine can not easily be assigned to its root cause. We therefore propose to add an option to *MicroWalk* to take the call graph into account when computing mutual information.

Responsible Disclosure We have informed the *Intel Product Security Incident Response Team (PSIRT)* and *Microsoft Security Response Center (MSRC)* of our findings. *MSRC* has not responded. After the initial report, we noticed that Intel have already patched `gfec_MulBasePoint` in Intel IPP v2018.3.240. Intel have acknowledged the receipt for the remaining vulnerabilities. Here is the time line for the responsible disclosure:

- **06/22/2018:** We informed our findings to the Intel Product Security Incident Response Team (Intel PSIRT) and the Microsoft Security Response Center.
- **06/25/2018:** Intel PSIRT acknowledged the receipt.
- **07/31/2018:** Intel PSIRT confirmed a work-in-progress patch for IPP 2018 update 4 (CVE-2018-12155, CVE-2018-12156).

Acknowledgements This work is supported by the National Science Foundation, under grant CNS-1618837.

REFERENCES

[1] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. 2010. New Results on Instruction Cache Attacks. In *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems (CHES'10)*. Springer, Berlin, Heidelberg, 110–124.

[2] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. On the Power of Simple Branch Prediction Analysis. In *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security (ASIACCS '07)*. ACM, New York, NY, USA, 312–320.

[3] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2006. Predicting Secret Keys via Branch Prediction. In *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA'07)*. Springer, Berlin, Heidelberg, 225–242.

[4] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 53–70.

[5] J. Bacelar Almeida, Manuel Barbosa, Jorge S. Pinto, and Bárbara Vieira. 2013. Formal verification of side-channel countermeasures using self-composition. *Science of Computer Programming* 78, 7 (2013), 796 – 812.

[6] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 583–600.

[7] ARM. [n. d.]. *Cortex-M3 Technical Reference Manual*. Chapter 18.4. Accessed: 2018-02-27.

[8] L. Bai, Y. Zhang, and G. Yang. 2012. SM2 cryptographic algorithm based on discrete logarithm problem and prospect. In *2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet)*. 1294–1297.

[9] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-level Non-interference for Constant-time Cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 1267–1279.

[10] Ali Galip Bayrak, Francesco Regazzoni, David Novo, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. 2015. Automatic application of power analysis countermeasures. *IEEE Trans. Comput.* 64, 2 (2015), 329–341.

[11] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. 2011. Finding software vulnerabilities by smart fuzzing. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. IEEE, 427–430.

[12] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. 2014. "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*. Springer, Berlin, Heidelberg, 75–92.

[13] Daniel J Bernstein. 2005. Cache-timing attacks on AES. (2005).

[14] Sandrine Blazy, David Pichardie, and Alix Trieu. 2017. Verifying constant-time implementations by abstract interpretation. In *European Symposium on Research in Computer Security*. Springer, Springer, 260–277.

[15] Daniel Bleichenbacher. 1998. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '98)*. Springer, London, UK, UK, 1–12.

[16] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 917–934.

[17] Ernie Brickell, Gary Graunke, and Jean-Pierre Seifert. 2006. Mitigating cache/-timing based side-channels in AES and RSA software implementations. In *RSA Conference 2006 session DEV-203*. RSA.

[18] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. 2018. CacheShield: Detecting Cache Attacks Through Self-Observation. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy (CODASPY '18)*. ACM, New York, NY, USA, 224–235.

[19] David Brumley and Dan Boneh. 2003. Remote Timing Attacks Are Practical. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Berkeley, CA, USA, 1–1.

[20] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1041–1056.

[21] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. 2017. FaCT: A Flexible, Constant-Time Programming Language. In *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017*. 69–76.

[22] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. 2017. Quantifying the Information Leak in Cache Attacks via Symbolic Execution. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '17)*. ACM, New York, NY, USA, 25–35.

[23] Jean-Sébasticn Coron, Paul Kocher, and David Naccache. 2001. Statistics and Secret Leakage. In *Financial Cryptography*. Springer, Berlin, Heidelberg, 157–173.

[24] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 857–874.

[25] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. 2018. CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018, 2 (2018), 171–191.

[26] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 51–67.

[27] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. USENIX, Washington, D.C., 431–446.

[28] Goran Doychev and Boris Köpf. 2017. Rigorous Analysis of Software Countermeasures Against Cache Attacks. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 406–421.

[29] DynamoRIO [n. d.]. DynamoRIO: Dynamic Instrumentation Tool Platform. http://dynamorio.org/. ([n. d.]). Accessed: 2018-02-27.

[30] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking Branch Predictors to Bypass ASLR. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 40, 13 pages.

[31] Ivan Fratric. [n. d.]. WinAFL. https://github.com/ivanfratric/winafl. ([n. d.]). Accessed: 2018-02-27.

[32] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* 8, 1 (01 Apr 2018), 1–27.

[33] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. 2015. Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, Springer, Berlin, Heidelberg, 207–228.

[34] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. 2008. Mutual Information Analysis. In *Cryptographic Hardware and Embedded Systems – CHES 2008*. Springer, Berlin, Heidelberg, 426–442.

[35] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 897–912.

[36] Silviu Guiaşu. 1977. *Information theory with new applications*. McGraw-Hill Companies.

[37] Berk Gulmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. 2017. PerfWeb: How to Violate Web Privacy with Hardware Performance Events. In *Computer Security – ESORICS 2017*. Springer, 80–97.

[38] Jae-Cheol Ha and Sang-Jae Moon. 1998. A common-multiplicand method to the montgomery algorithm for speeding up exponentiation. *Inform. Process. Lett.* 66, 2 (1998), 105 – 107.

[39] Mike Hamburg. 2009. Accelerating AES with Vector Permute Instructions.. In *CHES*, Vol. 5747. Springer, Springer, Berlin, Heidelberg, 18–32.

[40] Intel. [n. d.]. Intel Digital Random Number Generator (DRNG) Software Implementation Guide. https://intel.ly/1VNnVkE. ([n. d.]). Accessed: 2018-06-13.

[41] Intel. [n. d.]. Symmetric Cryptography Primitive Functions. https://intel.ly/2xwNvCM. ([n. d.]).

[42] Intel. [n. d.]. Understanding CPU Dispatching in the Intel® IPP Libraries. https://intel.ly/2QAcQo6. ([n. d.]). Accessed: 2018-02-27.

[43] Gorka Irazoqui, Kai Cong, Xiaofei Guo, Hareesh Khattri, Arun K. Kanuparthi, Thomas Eisenbarth, and Berk Sunar. 2017. Did we learn from LLC Side Channel Attacks? A Cache Leakage Detection Tool for Crypto Libraries. *CoRR* abs/1709.01552 (2017). arXiv:1709.01552 http://arxiv.org/abs/1709.01552

[44] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing – and Its Application to AES. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 591–604.

[45] Don Johnson, Alfred Menezes, and Scott Vanstone. 2001. The elliptic curve digital signature algorithm (ECDSA). *International journal of information security* 1, 1 (2001), 36–63.

[46] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. 2016. When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015. In *Cryptology and Network Security*. Springer, 573–582.

[47] Mehmet Kayaalp, Khaled N. Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. 2017. RIC: Relaxed Inclusion Caches for Mitigating LLC Side-Channel Attacks. In *Proceedings of the 54th Annual Design Automation Conference 2017 (DAC '17)*. ACM, New York, NY, USA, Article 7, 6 pages.

[48] S McCURLEY Kevin. 1990. The discrete logarithm problem. *Cryptology and computational number theory* 42 (1990), 49.

[49] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01203

[50] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology — CRYPTO '96*. Springer, Berlin, Heidelberg, 104–113.

[51] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2012. Automatic Quantification of Cache Side-Channels. In *Computer Aided Verification*. Springer, Berlin, Heidelberg, 564–580.

[52] A Langley. 2010. ctgrind: Checking that functions are constant time with Valgrind. (2010).

[53] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–.

[54] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. 2017. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In *Computer Security – ESORICS 2017*. Springer, 191–209.

[55] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 549–564.

[56] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. 2016. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 406–418.

[57] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 605–622.

[58] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200.

[59] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. 2007. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Springer, Berlin, Heidelberg.

[60] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. 2012. Fuzzing: The State of the Art. http://bit.ly/2DgUIrq. (2012).

[61] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations in SGX. In *Topics in Cryptology - CT-RSA 2018 - The Cryptographers' Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings*. 21–44.

[62] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *Cryptographic Hardware and Embedded Systems – CHES 2017*. Springer, 69–90.

[63] Nicholas Nethercote. 2004. *Dynamic binary analysis and instrumentation*. Technical Report. University of Cambridge, Computer Laboratory.

[64] Kaisa Nyberg and Rainer A. Rueppel. 1993. A New Signature Scheme Based on the DSA Giving Message Recovery. In *Proceedings of the 1st ACM Conference on Computer and Communications Security (CCS '93)*. ACM, New York, NY, USA, 58–61.

[65] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006*. Springer, Berlin, Heidelberg, 1–20.

[66] C. S. Pasareanu, Q. Phan, and P. Malacaria. 2016. Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. 387–400.

[67] Colin Percival. 2005. Cache missing for fun and profit. (2005).

[68] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. 2016. "Make Sure DSA Signing Exponentiations Really Are Constant-Time". In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 1639–1650.

[69] NIST FIPS PUB. 1993. Digital signature standard. (1993).

[70] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 431–446.

[71] O. Reparaz, J. Balasch, and I. Verbauwhede. 2017. Dude, is my code constant time?. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 1697–1702.

[72] Laurent Simon, David Chisnall, and Ross Anderson. 2018. What you get is what you C: Controlling side effects in mainstream C compilers. (2018).

[73] Rohit Sinha, Sriram Rajamani, and Sanjit A. Seshia. 2017. A Compiler and Verifier for Page Access Oblivious Computation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 649–660.

[74] Geoffrey Smith. 2009. On the Foundations of Quantitative Information Flow. In *Foundations of Software Science and Computational Structures*. Springer, 288–302.

[75] François-Xavier Standaert, Tal G. Malkin, and Moti Yung. 2009. A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. In *Advances in Cryptology - EUROCRYPT 2009*. Springer, Berlin, Heidelberg, 443–461.

[76] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. ACM, New York, NY, USA, 299–310.

[77] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 235–252.

[78] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, ACM, New York, NY, USA, 2421–2434.

[79] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. 2018. DATA – Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 603–620.

[80] Bernard L Welch. 1947. The generalization ofstudent's' problem when several different population variances are involved. *Biometrika* 34, 1/2 (1947), 28–35.

[81] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. 2017. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 859–874.

[82] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 640–656.

[83] Yuval Yarom and Naomi Benger. 2014. Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive* 2014 (2014), 140.

[84] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering* 7, 2

(2017), 99–112.

[85] Andreas Zankl, Johann Heyszl, and Georg Sigl. 2017. Automated Detection of Instruction Cache Leaks in Modular Exponentiation Software. In *Smart Card Research and Advanced Applications*. Springer, 228–244.

[86] Tianwei Zhang and Ruby B. Lee. 2014. New Models of Cache Architectures Characterizing Information Leakage from Cache Side Channels. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*. ACM, New York, NY, USA, 96–105.

[87] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. 2016. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In *Research in Attacks, Intrusions, and Defenses*. Springer, 118–140.