ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures

Ben Gras*†, Cristiano Giuffrida*, Michael Kurth*, Herbert Bos*, and Kaveh Razavi*

*Vrije Universiteit Amsterdam †Intel Corporation

***Abstract*—The past decade has seen a plethora of side-channel attacks on various CPU components. Each new attack typically follows a *whitebox* analysis approach, which involves (i) identifying a specific shared CPU component, (ii) reversing its behavior on a specific microarchitecture, and (iii) surgically exploiting such knowledge to leak information (e.g., by actively evicting shared entries to monitor victim accesses). This approach requires lengthy reverse engineering, repeated for every component and microarchitecture, and does not allow for attacking unknown shared resources.**

In this paper, we present ABSynthe, a system that takes a target program and a microarchitecture as inputs and automatically synthesizes new side channels. The key insight is that by limiting ourselves to (typically on-core) contention-based side channels, we can treat the target CPU microarchitecture as a black box, enabling automation. To make ABSynthe possible, we have automatically generated *leakage maps* for a variety of x86_64 microarchitectures. These leakage maps show a complex picture of interaction between different x86_64 instructions and justify a black box approach to finding the best sequence of instructions that cause information to leak from a given software target, which we also treat as a black box. To recover the secret information using the optimized sequence of instructions, ABSynthe relies on a recurrent neural network to craft practical side-channel attacks that recover a secret bit stream. Our evaluation shows that ABSynthe can synthesize better attacks by exploiting contention on multiple components at the same time compared to state of the art contention-based attacks that focus on a single component. Furthermore, the automation made possible by ABSynthe allows us to synthesize cross-thread attacks for a variety of microarchitectures (from Intel, AMD and ARM) on four different cryptographic software targets, in both native and virtualized environments. The results show that ABSynthe can recover cryptographic key bit streams with high accuracy. As an example, ABSynthe recovers a full 256-bit EdDSA key from just a single trace capture with 100% success rate on one of our test beds.

I. INTRODUCTION

Modern processors provide strong isolation guarantees between distrusting execution contexts at the architectural level of abstraction. These guarantees are unfortunately not enforced at the microarchitectural level. A plenitude of existing side-channel attacks show one can leak secret information (e.g., cryptographic keys) by examining changes made by a victim's execution to the state of shared microarchitectural components such as caches [1, 2, 3, 4, 5], cache directories [6], TLBs [7], and branch predictors [8, 9]. Such attacks are typically based on *whitebox* side-channel analyses which require heroic reverse engineering efforts to gain a deep understanding of the target component and then craft component-specific exploitation primitives (e.g., the ability to track victim cache accesses by actively forcing evictions). Such manual efforts need to be repeated for each new component and each (micro)architecture, in search of new, dedicated exploitation primitives.

In this paper, we present ABSynthe, an automatic, *black box* approach towards synthesizing microarchitectural side channels in a more general and sustainable fashion, by exploiting contention on shared resources. Its blackbox analysis requires no reverse engineering effort on the part of the attacker, and eschews complicated eviction strategies that require deep understanding of the dimensions, organization and policies that govern certain wide-exploited shared resource such as CPU caches. Instead, ABSynthe exploits the insight that the mere presence or absence of contention on shared resources often leads to measurable performance differences.

For efficiency reasons, microarchitectures today include a large number of shared resources that may, potentially, serve in side channels attacks. Examples include per-core caches, execution units, and execution ports, but there are many others. Since any of these resources *may* harbor a side channel, we designed ABSynthe as a generic solution to measure and optimize the information leakage—for any component, software, and (micro)architecture. Interestingly, during our analysis we also found new sources of leakage, including some that we cannot easily associate with a single component. By focusing exclusively on contention-based side channels, ABSynthe can treat the target CPU microarchitecture and its components as black boxes, while synthesizing side channels that are stealthy [10, 11], that are easy to regenerate across different architectures, and that may even combine multiple microarchitectural components to boost the signal and outperform state-of-the-art side-channel attacks.

Microarchitectural side-channel attacks Existing microarchitectural side-channel attacks rely on leakage primitives derived from the reverse engineering of a specific microarchitectural component. Such whitebox side-channel strategies often take the form of *eviction-based* attacks—attacks that exploit knowledge of the target component to measure modifications made to the microarchitectural state by the victim's secret operation. At their core, these measurements involve

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.23018
www.ndss-symposium.org

the eviction of certain state from the component of interest. For example, in FLUSH+RELOAD, the attacker evicts a shared cache line and later checks whether it is reloaded by the victim in a secret operation. While attacks on CPU caches are the most common, attackers may equally target other microarchitectural components, including cache directories [6], TLBs [7], and branch predictors [9]. Some recent efforts, such as PortSmash [10] and SMoTherSpectre [11], show that one can also leak secret information by crafting *contention-based* attacks that exploit contention on execution ports. The high-level strategy in contention-based attacks is to replace the *active evictions* of prior efforts with *passive monitoring*, which also vastly improves the stealthiness of the attack [10]. However, these efforts still rely on exploitation primitives specific to a particular microarchitectural component, in particular execution ports.

ABSynthe In this paper, we present the first complete Synchronous Multithreading (SMT) *leakage maps* for different microarchitectures implementing the x86_64 ISA. These *leakage maps* show complex interactions between different x86_64 instructions and allow us to make a number of observations. First, there are many different microarchitectural components that leak secret information and allow for practical contention-based attacks. Second, by testing different instructions, we can create contention on arbitrarily different microarchitectural components *without any knowledge of the contended component(s) or of the microarchitecture*, opening the door for attack automation. Third, the instructions that create observable contention on one microarchitecture do not necessarily do so on others. This means that contention-based attacks are not always portable across different processors with different microarchitectures. Building on these insights, we present ABSynthe, the first system to automatically synthesize new side-channel attacks on a given microarchitecture and a given software target.

To build ABSynthe, we combine a number of novel techniques. First, to automatically detect secret-dependent control flows in a given software target, we employ taint analysis similar to DATA [12]. We further refine the analysis by relying on performance monitoring counters to identify the target branches. Once ABSynthe identifies the target branches, it tries to find a sequence of instructions from the leakage maps that *maximizes the information leakage* from the target branches. ABSynthe relies on a genetic algorithm to find a combination of instructions that create the best contention to leak the maximum amount of information from the target software. The result is a highly optimized target-specific sequence of instructions that performs better than any single instruction used in recent work [10, 11]. Using a number of different cryptographic functions and commodity CPU architectures (Intel, AMD, ARM), we show that ABSynthe is effective in synthesizing practical cross-thread attacks in native and virtualized environments. Lastly, ABSynthe employs a Recurrent Neural Network (RNN) for complete cryptographic key bit stream recovery using the synthesized attacks. As we later show with an example, an analyst armed with ABSynthe's results can then recover the final secret key with basic post processing techniques.

Like fuzzers and other testing techniques, ABSynthe may primarily serve as a powerful *regression testing framework* for both hardware vendors to find new side channel leakages in their microarchitectures and software developers to determine if their, say, crypto algorithm is vulnerable to contention-based side channels on a target CPU.

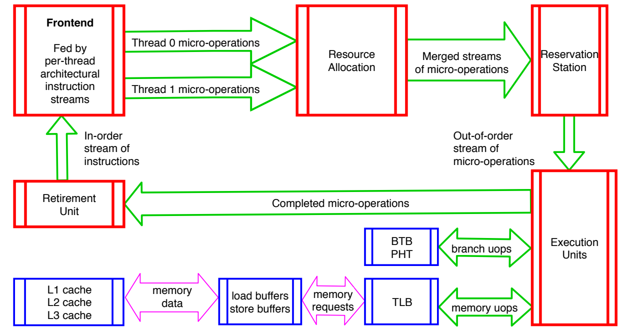


Fig. 1: Simplified diagram of a superscalar, out-of-order execution pipeline showing the stream of instructions being dispatched and retired (green, thick arrows), units that hold and process them (red, thick boxes), the memory request and data stream (purple, thin arrows), and their supporting microarchitectural caches and data structures (blue, thin boxes).

Contributions This paper makes the following contributions:

- We present the first complete *leakage maps* for a number of x86_64 microarchitectures in SMT settings which additionally provide us with new insights for building contention-based side-channel attacks.
- We present ABSynthe, a fully automated approach for synthesizing side-channel attacks on arbitrary microarchitectures and software targets (by treating both as blackboxes)—eliminating the need for per-component reverse engineering and enabling a portable, multi-component side-channel analysis on commodity CPUs.
- We show that ABSynthe can synthesize side-channel attacks on a variety of cryptographic routines on different architectures (Intel, AMD, ARM) and environments (native, virtualized) by creating contention on different microarchitectural components. Surprisingly, these attacks perform even better than creating contention on a single specific component as done in prior work [10, 11].

II. Background

We discuss the microarchitectures of modern processors to get an overview of shared components across different execution contexts. We then discuss how some of these components are prone to contention-based side channels.

A. Microarchitectural components

CPUs implement their Instruction Set Architecture (ISA) using microarchitectural components. Such components increase in number, size and complexity with each new CPU generation. Given that these components are not visible to software, their low-level details can only be uncovered by careful reverse engineering. Furthermore, the properties of each of these components can change between CPU revisions, without posing compatibility problems for the software, but

prompting new, lengthy reverse engineering efforts every time for side channel researchers.

Figure 1 shows the high-level overview of some of these microarchitectural components in a modern CPU, using Intel terminology as a reference. The micro-architecture can be divided in two main parts: (a) the frontend, which decodes the architectural instruction stream operating on architectural registers from multiple, in this case two, logical processors into micro-operations (micro-ops) operating on microarchitectural registers (physical registers), and (b) the backend, which operates on a stream of micro-ops merged from the 2 thread streams, allocates the necessary resources, and schedules them for execution in an out-of-order fashion. We focus on components in the CPU backend. While late in the pipeline, these components operate at the stage where the instruction streams of the multiple hardware threads have merged and resources are mostly shared between the threads. Significantly, these threads can be executing in different security domains.

We will not go into detail for all micro-architectural components depicted in the figure. After all, the objective of this paper is to exploit resource sharing without relying on the knowledge of the workings, or even the existence, of these components. We nevertheless briefly discuss the use of execution ports and units by micro-ops, as dispatched from the reservation station. In particular, the Reservation Station (RS) holds a collection of in-flight micro-ops which have allocated resources (e.g., physical registers) and are ready to execute once their operands are available. Micro-ops in the RS are mixed from both threads already. Meanwhile, the execution units run the actual micro-ops that are dispatched from the RS. After execution, micro-ops are sent to the retirement unit and retired in-order. Execution units are reached through execution *ports* that are typically numbered in a fairly intuitive manner ($P0$, $P1$, $P5$, etc.) and sometimes one micro-op can be executed on any of a set of ports (expressed as $P06$). As we shall see in Section IV, execution ports and units will play a large role in contention on shared resources.

B. Simultaneous Multi-Threading (SMT)

SMT is an architectural technology that is primarily intended to enable cross-thread sharing of on-core resources that would otherwise be unused [13, 14, 15]. When transitioning from a single-threaded to a multi-threaded core design, on-core resources are owned in one of three ways [16]:

1) *Replicated*: there is one instance per thread for private use. This happens for microarchitectural state such as the architectural register file and the instruction pointer.
2) *Partitioned*: there is a static assignment of ownership of half of the resource to each thread. Examples include the iTLB [7] and the Physical Register File (PRF) [17].
3) *Competitive*: there is a full resource pool available to all threads. Examples include execution slots in the reservation station, CPU caches, load/store buffers, L1 dTLB, Shared TLB (STLB) [7], and execution units [10].

As we will see in Section IV, measurable interference can occur across security boundaries on resources that are competitively shared, possibly leading to information leaking.

C. Eviction- vs. contention-based attacks

Existing side-channel attacks on microarchitectural components are largely *eviction-based*. That is, they use evictions to bring the target component to a known state. After the secret operation, they can then examine the state to infer any changes that leak information about the secret operation. As an example, the PRIME+PROBE attack primes the CPU cache by accessing a set of memory addresses (evicting other addresses). It then waits for the victim operation to execute. Finally, it probes the cache by checking whether any of the previously accessed addresses has been evicted from the cache. These attacks are powerful, as an attacker can actively control the microarchitectural behavior of the victim, but also hinge on (i) intimate knowledge of the target microarchitectural component, and on (ii) an active eviction strategy that reduces the stealthiness of the attack and is amenable to mitigations [18].

To address the latter, *some* recent attacks [10, 11] rely on the available bandwidth to execution ports to stealthily leak information across threads. The attacker simply measures the bandwidth (i.e., operations per second) over time and, by observing its fluctuations, can infer information about the victim's operation. This *contention-based* attack relies on passive monitoring rather than active evictions, improving the stealthiness and mitigation-resistance of the attack. However, the exploitation strategy is still targeted to a specific component and requires assumptions on the underlying microarchitecture and its interactions with the target software.

This paper's contribution is twofold. First, we show there are many more components amenable to contention-based attacks and it is not clear a-priori which component (or set of components) is the most effective on a given software target. Second, we show we can automatically synthesize contention-based side-channel attacks on any given microarchitecture. We discuss the first in Section IV and the second in Section V.

III. THREAT MODEL

We assume an attacker who has code execution on the victim machine. The aim of the attacker is to leak sensitive data, such as cryptographic keys, from a victim process or VM. Similar to existing contention-based attacks [10], we primarily focus on a victim executing on a (sibling) hardware thread on the same core as the attacker. In Section VIII-A, we discuss how we can generalize our analysis to non-SMT settings. We further assume that all state-of-the-art side-channel protections are enabled, but the target software is vulnerable to side channels (e.g., due to vulnerable coding practices [19]). The attacker seeks to automatically synthesize contention-based side-channel attacks against the given vulnerable software and microarchitecture. We focus our analysis on recovering the control flow of a target and use common examples with secret-dependent control flow.

IV. SIDE CHANNELS ON CONTENDED COMPONENTS

To motivate our work, we systematically study the possibility of creating contention-based attacks on different shared microarchitectural components normally accessed by victim software. We take a fully black-box, microarchitecture-agnostic

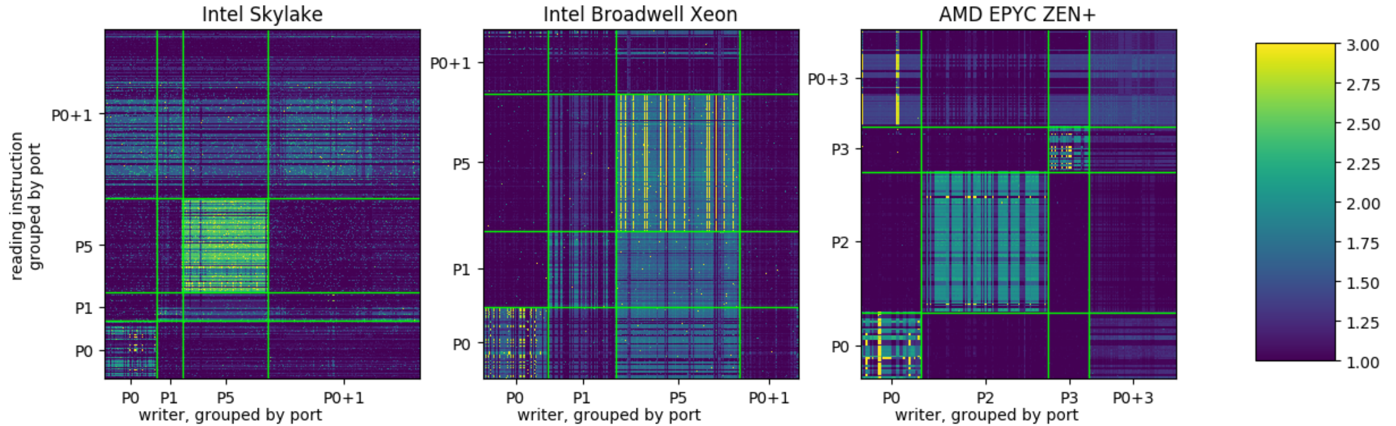


Fig. 2: Exhaustive map of instruction interference grouped by 4 different execution port sets on 3 different microarchitectures. For each microarchitecture, we visualize the matrix $CC_{B,A}$ as constructed using Algorithm 1.

approach by measuring instruction interference of every instruction vs. every other instruction, and find significant unexplained sources of contention.

A. Creating Contention on CPU Components

As any shared resource can potentially lead to an exploitable side channel, we wish to map out as many different sources of contention as possible within a given CPU core. This later allows us to find the best performing side channel for a given a software target.

Similar to the Covert Shotgun's blog post [20], our approach tries to find whether two instructions create observable contention on various CPU components. Unlike Covert Shotgun's handful of carefully picked instructions, however, ABSynthe's blackbox strategy covers the entire x86_64 ISA and synthesizes side-channel attacks rather than much simpler covert channels. For this purpose, we need to find instruction sequences that create the largest possible observable contention. To approach this in a principled way, we investigate measurable contention caused by *any single* instruction. We design an experiment that runs on a single physical core and measures interference $CC_{B,A}$, which can be read as "Interference factor that instruction B experiences under influence of instruction A." We call B the reader instruction (we observe its latency), A the writer instruction (it causes the latency difference, if any), and the $CC_{B,A}$ matrix the *leakage map* for a certain microarchitecture.

To make this possible for all instructions, we build on the XML file containing an exhaustive list of the instructions available of each microarchitecture of the `uops.info` [21] project to generate an implementation of Algorithm 1 automatically. Our reasoning is that any element $CC_{B,A} > 1.0$ is evidence of contention generated by A and experienced by B. To visualize the results, we group the rows and columns of CC by execution port usage of the corresponding instruction, again obtained from [21]. There are two reasons for this. First, since execution ports can be a source of contention, we wish to group this influence in contiguous bands in the visualization. Second, this ordering serves to group together instructions with similar functions and utilizing the same sets of execution units. We construct the $CC_{B,A}$ matrix by measuring the latency of instruction B while instruction A

Data: List of x86 instructions
Result: Matrix $CC_{B,A}$, the latency increase of each instruction B under the influence of A compared to **nop**.
On a core with SMT1 and SMT2:
for *Every x86 instruction* A **do**
  On SMT1: start a loop of **sequence** A;
  for *Every x86 instruction* B **do**
    On SMT2: $LAT_{B,A} \longleftarrow$ **rdtscp**(**sequence** B) ;
  end
end
for *Every x86 instruction pair* (A, B) **do**
  Compute $CC_{B,A} \longleftarrow LAT_{B,A}/LAT_{B,\mathbf{nop}}$
end

Algorithm 1: Constructing Covert Channel matrix $CC_{B,A}$

is executing (denoted $LAT(B, A)$) concurrently on a sibling thread, and express this latency as a factor of NOP executing concurrently: $CC_{B,A} = LAT(B, A)/LAT(B, NOP)$. The full procedure is described in Algorithm 1.

We perform Algorithm 1 on two different Intel microarchitectures, Skylake and Broadwell Xeon, and one AMD microarchitecture, ZEN+, using the EPYC platform. Figure 2 shows the leakage maps for these microarchitectures. A column represents the signature of a single instruction A running on SMT1, composed of many observations of different instructions B running on SMT2.

B. Discussion of the Results

Our measurements show that while execution ports are indeed a source of contention [10, 11], they are hardly unique and also interfere in more intricate ways than previously thought. Firstly, the contention pattern is not purely a function of execution port sets. In particular, instructions that share execution port sets may equally well show no contention or high contention. Secondly, instructions that do not share execution ports show overall low contention, but there *are* clusters and streaks of high contention patterns here also—implying shared microarchitectural resources between SMTs that are not purely execution ports or even execution units.

Significantly, we note that the contention signature of

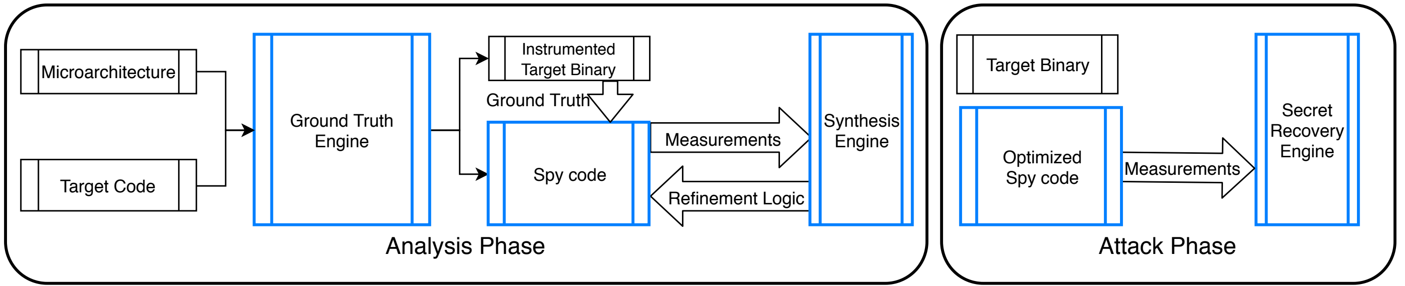


Fig. 3: High-level overview of ABSynthe.

every instruction (shown as a single column in the CC matrix), is different from that of other instructions. In other words, the CC matrix does not lend itself for simplification by grouping together instructions with identical signatures. Even when already grouped together according to execution ports, the interference signature is generally different each time, implying that there are many different causes for contention.

Lesson 1: There are multiple independent sources of contention.

Examining the CC matrices further, we notice the structure of resource contention is very different for each leakage map. Even microarchitectures from the same manufacturer (Intel) have a different leakage map. This suggests that a contention-based side-channel attack in one microarchitecture may not work well in another and this process may need to be repeated, unless it can be done in a fully black-box manner.

Lesson 2: A contention-based side channel on one microarchitecture may not work as well on another.

We further see clusters which correspond to shared resources used by one set of instructions interfering with other sets of instructions. For instance, the Skylake results exhibit a small number of clusters in the P0,P0 cell, whereas Broadwell Xeon shows a large number of clusters in the P0,P0 cell, and smaller clusters in other cells. Every row (corresponding to one *reader* instruction) with a small number of high contention values on *writer* instructions, will show reliable detection for those writer instructions. These findings suggest that the best possible (within the possibilities of ABSynthe) side channel may be obtained by creating contention on many different resources and possibly on multiple resources at the same time.

Lesson 3: The best contention-based side channel may require contention from multiple instructions.

We conclude that finding a sequence of instructions that optimally distinguishes a particular subset of target instructions, may require particular and non trivial combinations of reader instructions. Furthermore, the leakage maps for different microarchitectures do not generalize, and we have to repeat our black-box synthesis for each microarchitecture separately. We use these insights in the design of ABSynthe discussed next.

V. Automated Side-channel Synthesis

Given our earlier observations, we designed ABSynthe, an automated system that synthesizes, within parameters, the best possible contention-based side channel for a given software target by trying different sequences of instructions and creating the appropriate contention on different microarchitectural components. Figure 3 presents a high-level overview of ABSynthe.

In the *analysis phase*, ABSynthe takes a given microarchitecture and the target software as input. It then automatically generates an instrumented binary that synchronizes with a spy program whenever it performs a secret operation. The spy code is initially based on instructions from the target microachitecture's leakage map. For every well-performing instruction in the leakage map, ABSynthe communicates the raw contention-based measurements to the synthesis engine. The synthesis engine aims to improve the quality of the signal by generating new sequences of instructions based on the contention-based measurements. These new instruction sequences repeatedly refine the spy code until the synthesized side channel can detect the secret information with sufficient confidence. We find in our evaluation that in many cases single instructions can achieve an acceptable performance for side-channel synthesis. In other cases, however, refining the instruction sequence significantly improves the results.

After the analysis phase, in which ABSynthe uses synchronized steps to classify secret bits, the attack phase uses the spy code and ABSynthe's secret recovery engine to leak the secret information with *no synchronization* with the victim software (making it suitable for practical attacks). To realize these two phases in ABSynthe, we need to address three challenges:

- **C1** In the analysis phase, ABSynthe needs to automatically instrument the target software to synchronize the measurements with the spy code for collecting ground truth.
- **C2** In the analysis phase, ABSynthe needs to automatically refine the side channel for a given microarchitecture.
- **C3** In the attack phase, ABSynthe needs to recover secret information with a non-cooperating victim binary using the refined side channel.

Section VI describes how we addressed these challenges in the design of ABSynthe. In summary, to address **C1**, we use a combination of taint analysis and a novel technique that makes use of performance counters. To address **C2**, we use a differential evolutionary genetic algorithm relying on a Gaussian Naive Bayes classifier as its fitness function. Finally, to address **C3**, we use an RNN classifier for unsynchronized key bit stream recovery.

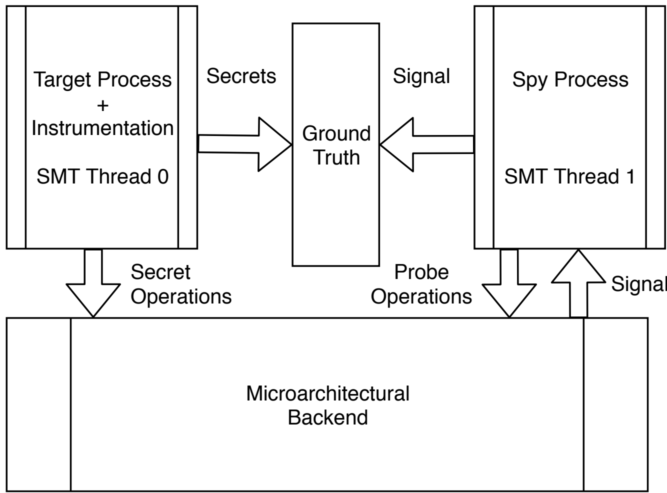


Fig. 4: Collecting ground truth information. The spy process is connected to the target through shared memory and is running on the same physical core.

```
1  for (j=nbits-1; j >= 0; j--) {
2   CRYPTLOOP_START(0); // pattern start, record '0'
3   _gcry_mpi_ec_dup_point (result, result, ctx);
4   if (mpi_test_bit (scalar, j)) {
5     CRYPTLOOP_VALUE(1); // bit == 1? pattern: '01'
6    _gcry_mpi_ec_add_points(result,result,point,ctx);
7   }
8  }
```

Fig. 5: Secret-dependent control flow: EdDSA 25519 elliptic curve multiplication, annotated for use in ABSynthe. If the ABSynthe pattern is ‘01’, the bit value was 1, and if it is ‘0’, the bit value was 0). The code is simplified.

VI. ABSynthe

At a high level, ABSynthe works by finding secret-dependent control flow in the target program during the analysis, and then detecting those secret-dependent code paths being executed, using a tuned measurement algorithm in a spy process. To do so, we first collect signals with known ground truth and use them to classify unknown signals. Specifically, as shown in Figure 4, we collect the ground truth by automatically instrumenting the target program at places where it processes a secret bit in order to collect the value of that bit, and then combine these values with measurements of the contention-based signal to determine how differently the signal looks for different values. In this section we discuss how we find the secret-dependent branches and collect the ground truth (Section VI-A), synthesize a tuned instruction sequence for measurements (Section VI-B), and finally map the measured signals to the secret (Section VI-C).

A. **C1**: *Automatically finding secret-dependent branches*

While executing the target program during the analysis phase, ABSynthe’s Ground Truth Engine explicitly signals the ground truth about the secret to the spy program. For this purpose, ABSynthe needs to instrument the target program at relevant secret-dependent program points. In the common case,

```
 for (j=nbits-1; j >= 0; j--) {
  CRYPTLOOP_START(0);
  _gcry_mpi_ec_dup_point(result,result,ctx);
  _gcry_mpi_ec_add_points(&tmppnt,result,point,ctx);
   if (mpi_test_bit (scalar, j)) {
      CRYPTLOOP_VALUE(1); // bit == 1? pattern: '01'
      point_set (result, &tmppnt);
   }
```

Fig. 6: Secret-dependent control flow of hardened ED25519 algorithm. We present a simplified sketch.

```
{
  CRYPTLOOP_START(0);
  _gcry_mpih_sqr_n (xp, rp, rsize, tspace);
  if ((mpi_limb_signed_t)e < 0)
    {
      CRYPTLOOP_VALUE(1);
       _gcry_mpih_mul ( xp, rp, rsize, bp, bsize );
    }
}
```

Fig. 7: Secret-dependent control flow of RSA. We present a simplified sketch of the RSA code.

such program points will be provided by the analyst (a mode of operation explicitly supported by ABSynthe), who may want to attack a very specific secret-dependent computation. However, assuming the analyst wants to target *any* computation dependent on a given secret, ABSynthe can automatically find the interesting victim instructions by determining branch instructions in the victim that are secret-dependent (and thus interesting/vulnerable) and executed often (and thus more side channel prone). Automatically locating such instructions by means of information flow tracking and profiling is straightforward, as we shall see. Moreover, while we make no claims that we can guarantee that such instructions are *always* the most “interesting” from an analyst point of view, this is very often the case in practice—especially for cryptographic software (See also Section VIII-B).

In particular, the two properties which guide ABSynthe to the instrumentation points in a blackbox fashion commonly

```
for (i=loops-2; i > 0; i--) {
  CRYPTLOOP_START(0);
  _gcry_mpi_ec_dup_point (result, result, ctx);
  if(mpi_test_bit(h,i)==1 && mpi_test_bit(k,i)==0) {
      CRYPTLOOP_VALUE(1); // NAF == 1? pattern: '01'
      point_set(&p2,result);
      _gcry_mpi_ec_add_points(result,&p2,&p1,ctx);
  }
  if(mpi_test_bit(h,i)==0 && mpi_test_bit(k,i)==1) {
      CRYPTLOOP_VALUE(2); // NAF == -1? pattern: '02'
      point_set(&p2, result);
      point_set(&p1inv, &p1);
      ec_subm(p1inv.y, ctx->p, p1inv.y, ctx);
      _gcry_mpi_ec_add_points(result,&p2,&p1inv,ctx);
  }
}
```

Fig. 8: Secret-dependent control flow: elliptic curve multiplication NIST P-256. The scalar is represented with NAF, causing 3 cases instead of 2. The code is simplified.

TABLE I: ABSynthe branch analysis. Number of unique branches executed by the target code (Executed), selected to be analyzed using Data Flow Analysis (DFA) for tainting, found to be tainted (Taint), and finally selected for ABSynthe instrumentation (Instrumented).

Target	Executed	DFA	Taint	Instrumented
ED25519	353	70	1	1
ED25519-hardened	353	70	1	1
RSA	472	86	6	1
ECDSA P-256	596	114	3	2

hold for crypto software. First, we assume that the operation executes the secret-dependent branches for a substantial part of the execution time in a number of iterations that is related to the key size. Second, the branch instructions should depend on the secret. As the public-key algorithms that we target iterate in multiple rounds over the bits that make up the key, taking up significant time, and few other algorithms will do so in a secret-dependent way, these properties hold for most of the crypto code that is not explicitly designed to not have secret dependent branches. ABSynthe finds the instrumentation points by profiling using dynamic taint analysis (in our current implementation by means of LLVM's DFSan [22]), while just requiring an analyst to taint the secret once in the source code.

For example, let us assume, that our target program executes the popular EdDSA 25519 elliptic curve cryptography algorithm from `libgcrypt`. In the implementation, the secret key is represented by a variable called `scalar` and Figure 5 shows the relevant loop with a secret-dependent branch in line 4. Lines 2 and 5 are part of the ABSynthe instrumentation and should be ignored for now. Figure 9 shows a flame graph breakdown for the EdDSA 25519 algorithm, representing the occurrences of stack traces, of the target execution as gathered using `perf record`. The key intuition behind branch selection revolves around the transition from `gcry_mpi_ec_mul_point`, which has nearly full cumulative execution time, to `gcry_mpi_ec_add_points`, and `gcry_mpi_ec_dup_point`. These two functions split the execution time in a way that depends on the secret key (as confirmed by the snippet in Figure 5). While we illustrated this for EdDSA 255519, it is a typical pattern: secret-dependent branches divide the execution time between themselves. We show annotated examples for the hardened version of EdDSA 25519 (Figure 6), RSA (Figure 7), and the NIST-P256 curve (Figure 8).

We now combine the flame graph with taint analysis to find the secret-dependent branches automatically, as follows:

1) We first build the code using LLVM with DFSan enabled and let ABSynthe taint all the data in the key file.
2) ABSynthe profiles the target program using `perf record` to find all functions with significant cumulative execution time, and instruments them with code that tests if the condition of the branch is tainted.
3) For the final ground-truth instrumentation, ABSynthe selects all branches that are tainted and that executed a significant number of times with respect to the key size (and did not have the same branch outcome every time).
4) Finally, ABSynthe also instruments the top of the parent loops of instrumented branches to let the spy differentiate between a varying number of non-taken secret branches.

An overview of the analyzed branches for different target programs can be seen in Table I. We see the total number of branches in the binary, and the selection process that leads to the final number of secret-dependent, instrumented branches to collect the ground truth. We verified that the found branches are indeed exactly the desired secret-dependent branches.

This approach generalizes to secret values that are larger than a single bit. In the general case, the ground-truth instrumentation records a unique pattern for each secret value. We give the spy code all the information needed to reconstruct the control flow, and with that, reconstruct the secret values.

As an example of the instrumentation for one of the EdDSA 25519 target, consider again Figure 5. The instrumentation in the top of the loop, `CRYPTLOOP_START()`, signals the start of the processing of a key bit by writing a marker to shared memory. Whenever a secret-dependent branch is taken, we signal that with a `CRYPTLOOP_VALUE(1)` by simply writing the corresponding value to shared memory. The spy code will read these values and collect them together with the side channel signal for training and evaluating.

Collecting the Ground Truth Given the instrumentation, ABSynthe extracts additional ground truth, namely how differently the contention-based signal looks when the target software is processing a particular secret bit value.

In the previous section, we explained how we instrument the target software to report when the target is taking or not taking secret-dependent branches. Our spy process, in turn, stores the contention-based *measurements* during the processing of these secret bits.

Synchronizing when the target is processing a particular secret value with the spy code is challenging since the synchronization itself can introduce noise into the measurements. To address this challenge, instead of using hard synchronization, ABSynthe's Ground Truth Engine relies on a *soft synchronization* strategy between the software target and the spy code. In particular, our design uses an efficient shared memory channel and signaling protocol between the target software and the spy code. To implement the shared memory channel, we simply allocate a single shared memory page. We design our signaling protocol to be asynchronous and minimalistic.

The spy code constantly monitors the target shared memory location and tags each latency measurement with the sampled value. This simple mechanism provides us with a reliable way to derive the ground truth of what a side channel signal looks like for the processing of the different secret key bits.

Virtualization support Our prototype implementation expands support to virtual environments by running both the target binary and the spy code in VMs. Instead of the native shared memory, we use a guest-accessible shared memory implementation, IVSHMEM [23], on KVM/Linux to facilitate the communication between the instrumented target software and the spy code. Our end-to-end strategy addresses **C1**.

B. **C2**: *Side-channel Refinement*

We seek to synthesize, within parameters, the best performing side-channel attack by creating contention on potentially

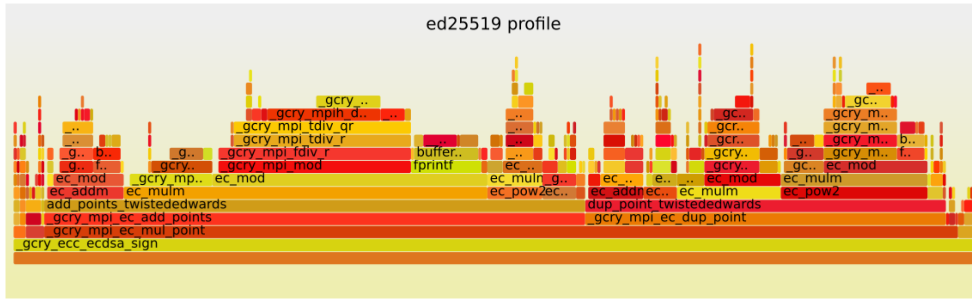


Fig. 9: Secret-dependent branches divide the execution time between themselves. This flame graph of execution of ed25519 shows this property which we found to be common across other cryptographic targets.

REPEAT NUMBER	BARRIER BEFORE START	BARRIER BEFORE END	INSTR1	INSTR2	INSTR3	INSTR4	COMBINE MODE
11	True	False	4	3	6	5	INTER LEAVE

Fig. 10: Recipe for a combination of side channel primitives. The four best performing single instructions are INSTR1 through INSTR4. The recipe states that 4 INSTR1 instructions are used, 3 INSTR2 instructions, and so on. That they are combined in an interleaving fashion. Further a memory fence instruction should be emitted before the code starts, and the whole snippet should be repeated 11 times.

multiple resources at the same time. To do this, we need to find the right sequence of instructions for a software target that we instrumented in the previous step. To find such a sequence, we choose a guided search algorithm that uses classification reliability (i.e., detecting correct information) as its optimization metric. The input to this algorithm are instructions that can create contention on various microarchitectural components. We use the best performing instructions as seeding set.

We wish to give the search algorithm freedom in choosing the final instruction sequence. The only requirement is that the instructions remain valid. Meeting both of these criteria, we design *recipes* that the system uses for synthesizing instruction sequences. Our evolutionary search algorithm mutates these recipes when looking for the best-performing solution. Before providing more details on the search algorithm, we first discuss the format of the recipes shown in Figure 10. Each recipe consists of a short string of integer parameters, each with a specific range that the search algorithm must respect.

Repeat number This parameter defines the number of iterations (between 1 and 20) the complete code needs to run. The execution time of the entire loop provides the raw signal for detecting the secret operation. We wish to strike a balance between observing a meaningful stretch of time and having a high-resolution measurement. The optimal value for this parameter depends on the synthesized code and on the target program.

Barriers These parameters define whether or not there is a memory barrier 1) before executing the instruction sequence or 2) after the instruction sequence is executed. Memory-dependent side channels benefit from these barriers, since they prevent the memory traffic from creating noise on the instructions that measure the execution time due to out-of-order execution. Barriers, however, may come at a cost of lowering the temporal resolution, due to the incurred latency of draining the memory pipeline. Hence, they can have adverse effects on the instructions that do not exercise the memory subsystem, and we don't want to force them, and leave their inclusion as a parameter.

Instruction blocks These parameters are each aimed at creating contention. Currently, for each parameter, we use an architecture-dependent sequence of instructions that yields a covert channel, as discussed in Section IV. Each parameter defines the number of instructions that is desired from a specific covert channel.

Combine mode This parameter specifies the method for combining the instruction blocks. We identify three possibilities: concatenation, interleaving, and a random shuffle after concatenation. With concatenation, we simply concatenate the instruction blocks in the recipe. With interleaving, we interleave instructions from different blocks. Finally, with random shuffling, we first concatenate different instruction blocks and then do a random shuffle of these instructions.

Evolutionary search algorithms require a fitness function that evaluates the fitness of the current population of solutions. We use a Gaussian Naive Bayes (GNB) classifier to decide whether a given sequence of instructions gives a signal measurement that is able to differentiate between the various code paths the target is executing.

To do this, we train the classifier on the signal values, a vector of 220 latency measurements. We label them with the ground truth obtained from the target instrumentation. The ground truth is the code path being executed by the target when the signal was measured. As an example we label these code paths 0 and 1.

To increase accuracy, we first apply a normalization step on the raw measurements from a given instruction sequence. We subtract the mean latency from all measurements, and then divide each measure in order to give the signal unit variance.

Finally, we then train the GNB using 75 target executions (providing empirically accurate results). Typically each execution gives us hundreds of code path samples, giving us a training set of e.g. 19200 '0' values and 9600 '1' values. We

then and observe how well the trained GNB classifier performs on a separate training set of 25 target executions. From this test, we compute the f1 score. We use this f1 score as the fitness function for the genetic algorithm. As an example of the raw signals, and of the effect of the signal preprocessing step, see Figure 11.

We chose the GNB classifier since it performs well without tuning parameters, and its training and evaluation have linear-time complexity which allows for our evolutionary algorithm to quickly progress.

We select Differential Evolution [24] (DE) for our evolutionary search algorithm. This choice is motivated by DE's ability of treating the optimized function as a black box, for not relying on a well-defined gradient to exist, and also for its resistance to a noisy fitness function, which is the case for our GNB classifier. The algorithm works by assembling a population of candidate solutions, described by a set of parameters for each candidate, and continuously trying to improve the candidates' fitness functions by combining their parameters into a new generation of the population. We define the discussed recipes as candidates which the DE strives to optimize using the GNB classifier.

We will show the effectiveness of our search strategy and a sample of refined sequence of instructions in Section VII.

C. **C3***: Secret Recovery*

The ground truth captured in the training phase includes synchronization information. This means we can train and test on signals with known equal starting points and length. Under such assumptions, the classifier can easily tells us the corresponding secret value. However, a realistic attack requires processing a captured signal with many consecutive such sub-signals starting at unknown position. In order to obtain secret recovery in practical settings, we need to recognize the sub-signals of many samples that correspond to a particular secret value and extract just the single value. A 2-label classifier, while fast to train and evaluate, is not equipped for this purpose, nor can it easily be adapted to reliably do so. The primary reason is the fact that we express a time series of samples, as a vector, to which the classifier assigns the same meaning in each position. However, the samples may easily desynchronize over time when analyzing a long capture.

Unsynchronized key recovery To detect signals corresponding to secret values in absence of synchronization, we turn to a sequence classification algorithm that is intended for time series data and is robust in the face of imperfect synchronization: a Long Short-Term Memory [25] (LSTM) Recurrent Neural Network (RNN). Not only does this network improve the synchronized classification significantly, it is also robust in face of small time shifts in the signal, allowing unsynchronized secret recovery.

LSTM models For robustness reasons that we will discuss later in this section, we design two different LSTM models, each with different detection characteristics. Model 1 consists of 3 stacked LSTM layers with decreasing number of cells (400, 300, 200). In order to avoid over-fitting, we have set a *dropout* of 0.2 on each layer. The first 2 LSTM layers propagate their hidden state to the subsequent layer. As we have a 3-label classification, we use a *softmax* layer to calculate the output probabilities. Model 2 is more complex in comparison to model 1, totaling six layers. We engineered the model so that after an LSTM layer, a fully connected layer with *RELU* activation follows, which we find to work well in practice. We use three such layer packets, again with decreasing numbers of cells (400, 256, 300, 128, 200). The LSTM layers also propagate their hidden state to the *RELU* layers. Therefore the *RELU* layers have input dimension 3. Similar to model 1, we use *dropout* to encourage resistance to over-fitting and a *softmax* layer for classification. The two models are the result of engineering various models of different shapes and parameters and selecting the ones that provided the best test-score. In most cases, model 1 performs slightly better than model 2. However, we use the two deep neural networks in a stacked ensemble, which results in a better performing classifier. We implement our models using the Keras [26] interface to Tensorflow [27].

We combine the results from two different models in a stacked ensemble in order to be more resistant to miss-guessing secret values, as not guessing any secret value is much less damaging than guessing the wrong value. We fully detail this design decision when discussing the brute force tradeoff, below. We train the weights on the models as follows.

1) We assemble a labeled training set. The features of the training samples are latency values, which have been processed for normalization, in the same way as in the Gaussian Naive-Bayes classifier described above.
2) We include 2 types of samples in the training set. The first type is a signal of latencies, where the start time coincides with the moment a code path is being processed by the target according to the ground truth data. The second type is a signal that starts somewhere between the start of one path and the start of the next. We label this with a special, extra label that is not used by the instrumentation.
3) As with the Gaussian classifier, we train the models on 75 executions of the target, typically giving 384 code paths each execution of the first, synchronized type. Added to these is a equal number of special training samples that are not synchronized.

Brute force tradeoff A recovered secret is frequently not exactly correct. A reasonable amount of brute-force search performed by the analyst is acceptable in order to get to the correct key from a recovered key bit stream. We assume we can verify a candidate secret, which we can do if the secret is, say, a cryptographic key that is used in a signing operation. This leads to a crucial observation.

The secret recovery algorithm will return a sequence of $(time, secret)$ pairs. Whenever a secret is miss-detected as a wrong value, brute forcing means we have to try hundreds of positions to find the wrong bit. For a 384-position guess and N wrong guesses, brute forcing leads to a 384^N work factor for every bit. The approach quickly becomes infeasible.

However, we can reliably detect whenever a secret value is missing (deletion), or whenever a spurious value was inserted (insertion), because of the time series information. Deleted bits leave a large gap, and inserted bits create very narrow gaps.

For deletions, it is clear where to insert a trial bit, or not, giving 3 possibilities: ignore, insert 0, insert 1. For insertions, assume one of the three bits forming the two narrow gaps is to blame, and try to delete each of them, also leading to 3 possibilities. We can tolerate quite a few insertions and deletions before brute force becomes infeasible—for a total of N insertions plus deletions, we have to do 3^N trials.

This means we should design our detection algorithm to be conservative in secret value predictions. We can compensate missing values with brute force to a large degree. This is the key reason we train 2 different LSTM models and only accept a secret value prediction when they both predict the same one.

This approach results in a more reliable classification system, so robust in classifying time series data that, given a sufficiently high-quality underlying side channel, a practical secret recovery system can be built on it. This strategy addresses **C3**.

VII. Evaluation

In this section, we quantify the effectiveness of ABSynthe in synthesizing practical side-channel attacks. We first discuss our evaluation environment and our example software targets. We aim to answer six questions in our evaluation of ABSynthe.

1) Can aligned snippets of execution traces corresponding to single secret bits be reliably classified into the correct secret bit by ABSynthe's spy process? We show that this the case by presenting our results in Section VII-C.
2) Can a complete capture of a secret key operation, without any alignment, reliably be mapped back to the original secret bits? This is a more challenging task than the previous one. We show how ABSynthe can make predictions without any alignment information in Section VII-D. We show how these predictions can easily be turned into a fully automated secret recovery using a case-specific heuristic in one of our software targets.
3) How does black-box generation of instruction sequences in ABSynthe compare against the state-of-the-art contention-based attacks such as PortSmash [10] and SMoTherSpectre [11]? We show that ABSynthe's DE algorithm manages to automatically find better sequence of instructions for leaking information in Section VII-E.
4) Is ABSynthe robust to signal capturing larger than the execution region of interest? Can we detect where the region of interest is, namely where secret key bits (which we have trained our classifiers to detect) are being processed? We evaluate this using a crypto-as-a-service scenario in a larger application (i.e., GnuPG) in Section VII-F1.
5) How robust are the synthesized attacks against noise? We detail the impact of interference from concurrently executing other processes on key recovery performance in Section VII-F2.
6) Can we generalize our system to not only detect secret-dependent code accesses, but also secret-dependent data accesses? We show the results of extending ABSynthe to synthesize cache attacks in Section VII-G.

A. Experimental Environments

We use four different testbeds. Two are based on Intel processors, one with a desktop processor, a 4-core 2-way SMT Intel Skylake i7-6700K running Ubuntu 18.04 and another with a server processor, an 8-core 2-way SMT Intel Broadwell E5-2620 running Debian Buster. We also use a system with a 24-core 2-way SMT AMD EPYC Zen 7401P running Ubuntu 18.04.1. Finally, we report partial results on ARM for a two sockets machine, each with a 56-core 4-way ARM Cavium Thunder X2 running Ubuntu 16.04.6 LTS. Note that the results presented in this paper are the first exploration of SMT-based side channels on ARM platforms, but since we do not have a machine-readable ISA for ARM64[1] for creating its leakage map, we only apply ABSynthe on hand-written instruction sequences and report its effectiveness.

To study the effectiveness of ABSynthe in virtualized environments, we create VMs on the Broadwell and EPYC testbeds. We run both the spy process and the victim process in different VM instances locked to different threads on the same physical core in order to assess cross-VM information leakage with ABSynthe.

Unless otherwise mentioned, we report median figures. In all our experiments, ASLR is turned on, which means that contention-based attacks synthesized by ABSynthe are not affected by this defense.

B. Software Targets

We use four versions of cryptographic functions in libgcrypt 1.6.3 and libgcrypt 1.8.5 as targets: EdDSA 25519, EdDSA 25519-hardened, EdDSA 25519-secure (1.8.5 only), RSA, and ECDSA P-256. EdDSA 25519-hardened has a rudimentary side channel mitigation, while the side-channel mitigation in EdDSA 25519-secure is considered state-of-the-art. We emphasize that we are not interested in the susceptibility of these algorithms to side channels *per se*, but rather in showing that ABSynthe is a generic testing solution to evaluate any such function. Next, we provide further information on these software targets.

EdDSA 25519 Figure 5 shows a point-scalar multiplication with secret-dependent code path. This is an elliptic curve secret key operation, where the secret is the `scalar` variable. The ABSynthe instrumentation has been automatically applied as explained in Section VI-A. EdDSA 25519 operates on the Curve25519 elliptic curve. A point-scalar multiplication proceeds in a series of point doublings and additions, under control of the secret scalar k. The doublings are unconditional, but the additions happen only for 1-valued bits in k. As a result, the control flow can be used to infer the pattern of secret key bits in the scalar.

EdDSA 25519-hardened This is a variant of EdDSA 25519 which explicitly mitigates against secret-dependent control flow side-channel attacks. Specifically, it performs the duplication and addition operations unconditionally, and only conditionally uses the result of the addition. While not a state-of-the-art mitigation anymore (a more secure version is discussed next), this is an illustrative example of a side channel mitigation that is less secure than it would seem at first glance. As we shall see, ABSynthe is able to detect

[1]Since doing this work, the authors have been made aware of the ARM Machine Readable Architecture specification [28], and we include a short discussion of this limitation in Section VIII-B.

the conditional exchange with high reliability *automatically*, proving the value in incorporating ABSynthe into an analysis cycle when designing side channel mitigations. In our results, we designate this variant `ED25519-hardened`.

EdDSA 25519-secure This is a variant of EdDSA 25519 with state-of-the-art side-channel mitigations for secret-dependent computations. This variant also does the duplication and addition operations unconditionally, and conditionally uses the result of the addition, but without control flow-level conditionals. We expect ABSynthe not to be to able to synthesize a successful attack against this target. As there is no secret dependent control flow, we manually annotate the loop with the secret key bit being processed.

RSA We use the simplified RSA code in libgcrypt that follows the familiar square-and-multiply pattern while processing a secret key exponent, usually referred to as d. The squaring is unconditional, but the multiply is conditional on a 1-bit in d. Thus, the control flow of the modular exponentiation can be directly mapped to the secret key bits in d.

ECDSA NIST P-256 Our fifth example target is also ECC point-scalar multiplication code, but of a different type from EdDSA 25519. The NIST P-256 curve follows a similar multiplication procedure, but first converts k to Non-Adjacent Form (NAF). This is a representation where each position can be valued 0, -1 or 1, and on average, only one third of the digits will be non-zero. This allows a multiplication to be evaluated with fewer point additions (and, for -1 values, subtractions). Control flow for this multiplication has 3 cases: the k digit is 0, -1 or 1. In the case of 0, only a doubling is performed. In the case of 1, a point doubling and addition is performed. In the case of -1, a point doubling and subtraction. This means that recovering the control flow of a target gives us the representation of k in NAF. If desired, it can be trivially transformed into a binary representation. We show that this 3-label detection case too can be done in ABSynthe with high reliability. For more information on NAF, see [29, 30].

The target code is automatically instrumented as explained in Section VI-A.

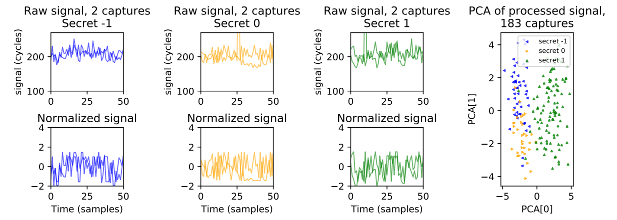
C. Classification Reliability

As mentioned earlier, we use the Gaussian Naive Bayes (GNB) classifier to assess the quality of the synthesized side channels by ABSynthe. GNB allows fast training and testing and gives a baseline of reliability for the RNN which we use to aid complete key recovery later. RNNs take much longer to train and test, but they typically perform better than the GNB classifier and do not require synchronization with the target.

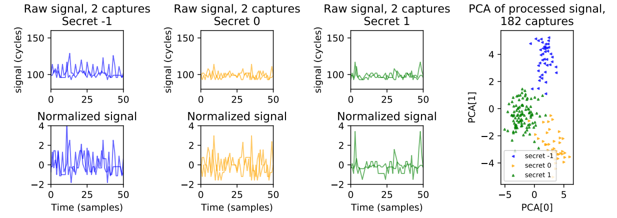
We design experiments to obtain contention-based measurements for *all* instructions available on a processor. For each instruction, we instantiate a side channel, and collect traces in the spy program with the associated ground truth. All of this can be parallelized and since each capture is quite short, this one-time process just takes around an hour for each software target and machine combination.

On Intel Broadwell and AMD EPYC Zen, we further experiment with virtualized environments, where target and spy are running in separate KVM instances. We collect 100 traces, give the GNB model 75 training traces (providing empirically accurate results), and test the reliability on the remaining 25 traces. It takes around 10 seconds to train the model for each scenario. Furthermore, we seek to enhance the performance of the side-channel primitives by combining instructions using the evolutionary algorithm as discussed in Section VI-B. We first show the complete results for this analysis before showing how the evolutionary algorithm can improve the quality of the signal.

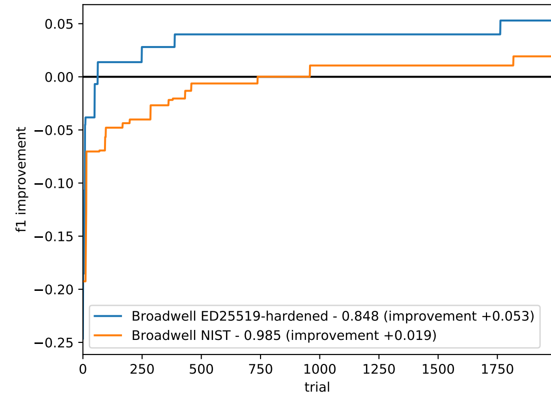
The spy traces that we collect are aligned and correspond to a small number of secret ground truth values, which are



(a) Signal of best-performing single-instruction side channel on the Broadwell-NIST-P256 target (no clear separation in PCA).



(b) Signal of evolved side channel (with clear separation).



(c) DE algorithm progressively finding better side channels.

Fig. 11: Raw signal processing and classification on Intel Broadwell, applied to the NIST P-256 target. In (a) and (b), we use a 2-dimensional PCA to show that the DE algorithm can better discriminate between 0 and 1 key bits than the best-performing side-channel primitive (as evidenced by the clear separation in the PCA plot). The bottom figure shows how DE can progressively find a better side channel through mixing different side-channel primitives together.

TABLE II: F1 score reliability results for best-performing primitive side channels, as well as our evolved side channels. We test on 4 different microarchitectures (Intel Skylake, Intel Xeon Broadwell, AMD Zen and ARM Cavium Vulcan). For the x86_64 microarchitectures, we test all possible instructions and show the 4 best performing ones (Instr4, Instr3, Instr2 and Instr1, in order). DE corresponds to our evolved side channel, which is an improvement in some cases. On ARM, we only use hand-written side channel snippets that stress memory (e.g., loads, but no cache eviction) and ALU operations (e.g., XORs). These classifiers operate on aligned signals and classify into secrets. The classifier in the LSTM column operates on unaligned secrets (discussed in Section VII-D.)

Intel Xeon Broadwell	**Software Target**	**Instr4 F1**	**Instr3 F1**	**Instr2 F1**	**Instr1 F1**	**DE F1**	**LSTM F1**
native	gcrypt 1.6.3 EDDSA 25519	0.97	0.98	0.98	0.98	**0.98**	0.95
native	gcrypt 1.6.3 EDDSA 25519-hardened	0.76	0.77	0.79	0.79	**0.84**	0.88
native	gcrypt 1.6.3 ECDSA P-256	0.94	0.94	0.94	**0.97**	0.97	0.91
native	gcrypt 1.6.3 RSA	0.81	0.81	0.83	**0.88**	0.88	0.91
Cross-VM (Debian Stretch)	gcrypt 1.6.3 EDDSA 25519	0.96	0.97	0.98	**0.99**	0.99	0.98
Cross-VM (Debian Stretch)	gcrypt 1.6.3 EDDSA 25519-hardened	0.66	0.66	0.68	0.68	**0.70**	0.76
Cross-VM (Debian Stretch)	gcrypt 1.6.3 ECDSA P-256	0.97	0.98	0.98	**0.98**	0.98	0.95
Cross-VM (Debian Stretch)	gcrypt 1.6.3 RSA	0.72	0.72	0.73	**0.80**	0.80	0.88
Intel Skylake	**Software Target**	**Instr4 F1**	**Instr3 F1**	**Instr2 F1**	**Instr1 F1**	**DE F1**	**LSTM F1**
native	GnuPG 2.2.17/gcrypt 1.6.3 EDDSA 25519	0.98	0.99	0.99	0.99	**1.00**	0.99
native	gcrypt 1.8.5 ECDSA P-256	0.99	0.99	0.99	1.00	**1.00**	0.99
native	gcrypt 1.8.5 EDDSA 25519	0.99	0.99	0.99	0.99	**1.00**	0.99
native	gcrypt 1.8.5 EDDSA 25519-hardened	0.77	0.79	0.79	0.81	**0.90**	0.92
native	gcrypt 1.8.5 EDDSA 25519-secure	0.53	0.53	0.53	0.53	**0.53**	0.66
native	gcrypt 1.8.5 RSA	0.73	0.73	0.73	0.74	**0.82**	0.82
native	gcrypt 1.6.3 ECDSA P-256	0.98	0.98	0.98	0.98	**0.99**	0.95
native	gcrypt 1.6.3 EDDSA 25519	0.99	0.99	0.99	0.99	**1.00**	0.99
native	gcrypt 1.6.3 EDDSA 25519-hardened	0.78	0.78	0.80	0.81	**0.91**	0.93
native	gcrypt 1.6.3 RSA	0.74	0.74	0.75	0.76	**0.79**	0.81
AMD EPYC Zen+	**Software Target**	**Instr4 F1**	**Instr3 F1**	**Instr2 F1**	**Instr1 F1**	**DE F1**	**LSTM F1**
native	gcrypt 1.6.3 EDDSA 25519	0.98	0.98	0.98	0.98	**0.99**	0.97
native	gcrypt 1.6.3 EDDSA 25519-hardened	0.73	0.73	0.74	0.74	**0.85**	0.80
native	gcrypt 1.6.3 ECDSA P-256	0.95	0.95	0.95	0.95	**0.96**	0.91
native	gcrypt 1.6.3 RSA	0.66	0.67	0.68	0.68	**0.75**	0.79
Cross-VM (Debian Stretch)	gcrypt 1.6.3 EDDSA 25519	0.83	0.84	0.84	**0.86**	0.86	0.80
Cross-VM (Debian Stretch)	gcrypt 1.6.3 EDDSA 25519-hardened	0.82	0.83	0.83	**0.89**	0.89	0.78
Cross-VM (Debian Stretch)	gcrypt 1.6.3 ECDSA P-256	0.67	0.67	0.68	**0.70**	0.70	0.60
Cross-VM (Debian Stretch)	gcrypt 1.6.3 RSA	0.53	0.53	0.56	0.57	**0.71**	0.66
ARM Thunder X2 Vulcan	**Software Target**			**ALU F1**	**LOADS F1**	**DE F1**	**LSTM F1**
native	gcrypt 1.6.3 EDDSA 25519			0.79	0.74	**0.85**	0.84
native	gcrypt 1.6.3 EDDSA 25519-hardened			0.51	0.52	**0.56**	0.66
native	gcrypt 1.6.3 RSA			0.51	0.43	**0.57**	0.68
native	gcrypt 1.6.3 ECDSA P-256			0.77	0.33	**0.84**	0.86

known from our instrumentation code in the training phase. When evaluating the quality of this side channel, we use the synchronization information for alignment and run the classifier on the unknown, aligned trace. If the classifier guesses the right secret most of the time, we have a signal. The quality of guessing is combined in an $F1$ score, an accuracy score corrected for testing set size, where 1.00 denotes a perfect score.

Given that we exhaustively measured the effect of all instructions on all architectures on all targets, we are able to give an interesting overview of these results grouped in different ways. Figure 13 shows these F1 scores grouped both by architecture and by software target.

Table II shows how well the 2-label classifier works (or in the case of NIST P-256, 3-label) when using instruction sequences found with ABSynthe's DE algorithm on our evaluation platforms in different settings (i.e., native and virtualized). Each classifier distinguishes aligned samples into one of the secret values. We also include the F1 scores for the four best-performing instructions we found in our exhaustive, black-box test. As the results show, ABSynthe successfully synthesizes side channels on different platforms and software targets. Furthermore, the ABSynthe's DE algorithm can in certain cases synthesize a better side channel by creating contention at multiple resources at the same time. As expected, there is not much to gain for EdDSA 25519-secure which was designed explicitly with side channels in mind, but for the other algorithms, the signal is significantly improved by DE, sometimes even across VMs. These results show the value of using ABSynthe's automated pipeline for testing the susceptibility of cryptographic functions against contention-based side-channel attacks.

On the ARM platform where we currently do not have a leakage map, we write some snippets by hand that we expect to generate contention: one snippet that does XOR operations and exercises the ALU unit and another snippet that does memory loads and exercises the memory subsystem (without cache eviction). Clearly, non of this is exhaustive and the F1 scores are lower in general. However, even without a full instruction set, ABSynthe's DE algorithm can synthesize a significantly better side channel from these primitives than either snippet.

Figure 11 provides more detail on the improvements made to the synthesized side channel using ABSynthe's DE algorithm. As an example, we show the improved signals collection on the NIST P-256 target on an Intel Broadwell machine. NIST P-256 has 3 secret values due to the NAF representation (see Figure 8 for more information). We visualize the improvement using Principal Component Analysis (PCA) on the normalized signal. This is a more rudimentary technique than our classifier (GNB), but lends itself better to visualizing the ability for the

```
# DB1
DIV CL
IDIV R8
PUSHFQ
DIV CL
IDIV R8
PUSHFQ
DIV CL
IDIV R8
PUSHFQ
DIV CL
IDIV R8
PUSHFQ
IDIV R8
PUSHFQ
IDIV R8
PUSHFQ
IDIV R8
PUSHFQ
PUSHFQ
```

Fig. 12: An example of an evolved side channel snippet, combining a varying number of 3 different primitive instructions. The `DB1` annotation is a signal to the code synthesis system that a memory barrier (`mfence`) must be emitted before the first `rdtscp`.

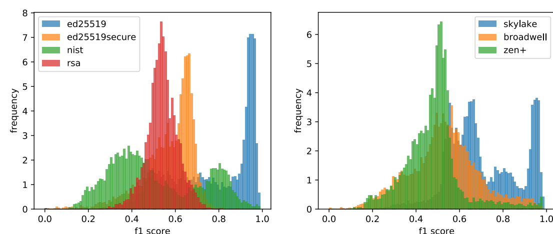


Fig. 13: Histogram of side channel reliability organized by software target and by platform.

signals to be separated. We visualize the signal by forcing PCA to express them using just 2 components, and plot these 2 components in a scatter-plot, showing whether or not the signal can be separated this way.

Figure 11a shows that with the best-performing instruction it is difficult to distinguish between -1, 0 and 1 bits, while Figure 11b shows that the distinction is very clear with our DE-refined side channel. The improvement is the result of ABSynthe creating contention on different resources at the same time. Finally, Figure 12 shows what a sequence of instructions found by DE may look like, clearly illustrating that it is difficult (if not impossible) to manually create such sequences.

D. Unaligned Secret Bit Sequence Recovery

We next show an analyst armed with ABSynthe's results is capable of recovering secret bit sequences without synchronization with the victim. We train the LSTM models using 75 training traces, and then evaluate the performance of secret recovery using 7 additional testing traces. It takes roughly a one-time session of 15 minutes to train the model for each scenario.

TABLE III: Unaligned secret bit sequence recovery. These are all performed on Intel Skylake on the EdDSA 25519 target. We show the number of trials, success rate, and median brute force attempts needed. The GnuPG case uses the same software target, but the full execution trace of GnuPG is processed, and the secret-dependent region of interest is automatically identified and analyzed without external cues.

Platform	Target	Instr	Trials	Success	Med. BF (2^N)
Skylake	ED25519	DE1	7	1.00	7.9
Skylake	ED25519	Instr2	7	1.00	15.8
Skylake	ED25519	Instr1	7	1.00	15.8
Skylake	GPG/ED25519	DE1	7	0.71	29.7
Skylake	GPG/ED25519	Instr2	7	0.86	22.5
Skylake	GPG/ED25519	Instr1	7	1.00	17.4

Our LSTM classifiers are expected to classify a signal into a certain secret (implying alignment), or a special blank label, implying no alignment. We use the predictions from the synchronized classifiers as the ground-truth for the unsynchronized classifications. The results of this experiment can be found in the LSTM F1 column in Table II. In many cases, the LSTM classifiers achieve a very good F1 score implying a strong signal for the secret key bits.

To recover the actual secret (key) in an example scenario, we pick the EdDSA 25519 target on the Skylake architecture, for its high reliability score in the aligned secret classification scenario. At this point, an analysis can perform post processing of the signal to recover the actual secret key bits. We exemplify this with a basic heuristic. This heuristic assumes that secret bits are processed in key bit order and uses the density of the label predictions by the LSTM models as an indication of the secret key bits. In cases, where the heuristic is not confident with the predictions, the target secret key bit will be left for brute-forcing. Given the key value guesses that the LSTM models made, we needed to do a modest amount of brute forcing to reach the exact original key. We limit this to 2^{40} brute-forcing trials. If the key guess requires more brute forcing than that, we call that trial a failure. Otherwise, the trial is successful and we report the median of how many brute force trials were needed.

Table III shows the results, including the number of trials performed in each scenario, the success rate, and the median number of brute force attempts needed before we could guess the correct key in the successful cases. ABSynthe's synthesized end-to-end side channels were 100% successful in aiding secret key recovery, using only a single trace capture and a modest amount of brute forcing, even when the cryptographic function is embedded inside a full application.

These results demonstrate that a simple case-specific heuristic is effective for the recovery of arbitrary secret keys using ABSynthe's unaligned secret recovery analysis. We leave the exploration of other signal processing heuristics (e.g., cross correlation) for other software targets as future work.

TABLE IV: Classification performance of ABSynthe classifier using PortSmash and SMoTherSpectre instruction sequences. `ror` and `popcount` are unique to SMoTherSpectre. All except `popcount` are used by PortSmash. On Skylake microarchitecture in the native environment (non-virtualized).

Software Target	`add`	`paddb`	`ror`	`andn`	`crc32`	`vpermd`	`popcnt`	**ABSynthe**
gcrypt 1.6.3 ECDSA P-256	0.83	0.84	0.79	0.80	0.54	0.82	0.59	**0.99**
gcrypt 1.6.3 EDDSA 25519	0.95	0.93	0.95	0.95	0.87	0.95	0.89	**1.00**
gcrypt 1.6.3 EDDSA 25519-hardened	0.64	0.67	0.65	0.65	0.61	0.70	0.64	**0.91**
gcrypt 1.6.3 RSA	0.64	0.64	0.59	0.61	0.55	0.65	0.56	**0.79**

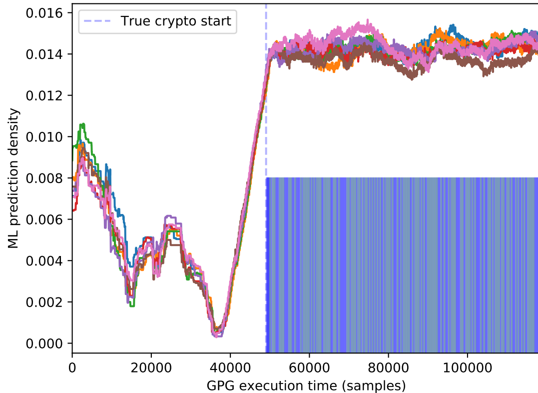


Fig. 14: Noise resistance heuristic in full GnuPG execution. We show the heuristic operating correctly in 7 different trials.

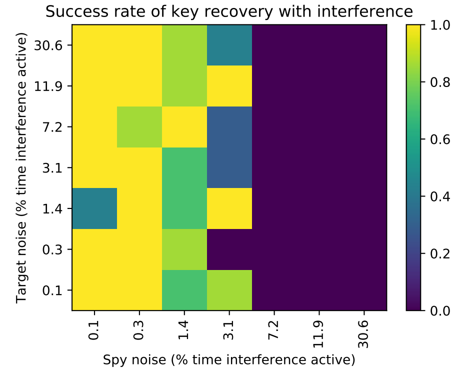


Fig. 15: Interference noise resistance in key recovery. We show all combinations of 7 different artificially induced interference levels. We execute an interference process on the same logical processor as both the target and the spy process, and vary the amount of CPU time that the interference process claims. We show the resulting key recovery success rate for each combination.

E. Comparison with manually discovered sequences

In recent work, PortSmash [10] and SMoTherSpectre [11] suggested exploiting contention-based side channels using manually discovered instruction sequences. We now compare such sequences against the best sequences found by ABSynthe on our target.

PortSmash uses instructions `add`, `paddb`, `ror`, `andn`, `crc32`, and `vpermd`, as evidenced from the original source code repository[2]. SMoTherSpectre uses `ror` and `popcnt` (the latter missing from PortSmash's list). For each selected instruction, we compare its classification performance on each of our targets to the performance of the best sequence that ABSynthe found. Table IV presents our results. According to our classifier, ABSynthe's automated DE algorithm outperforms all the other sequences in terms of classification reliability by a wide margin by finding instructions that create the maximum contention for a given target.

F. Robustness

In this section, we evaluate two robustness aspects of ABSynthe. Firstly, if we capture the side channel signal during the execution of a target program, can we automatically identify the region of interest, i.e., the region during which the secret key bits are processed? Secondly, if either the spy process, the target process, or both, are periodically interrupted by concurrent computation, can we still perform key recovery on the resulting signal? We detail, visualize and quantify these two aspects next.

1) Automatically finding the region of interest: While it is straightforward to record our side channel signal, it is less straightforward to determine when the execution region of interest, namely the processing of the cryptographic key, occurs. The detection algorithm will be processing data it is not trained to handle, and spurious bit predictions may occur. To show that our algorithm can adequately handle this situation and detect when the region of interest starts, we record side-channel samples on a full execution of GnuPG 2.2.17 linked with libgcrypt 1.8.3, using the non-sidechannel-safe EdDSA 25519 algorithm. We find that, while there are many spurious cryptographic bit predictions, the density of predictions in the region of interest is significantly higher. We can use this pattern as a reliable heuristic to detect the region of interest without external synchronization. This data is illustrated with 7 different executions of GnuPG in Figure 14.

2) Target and spy executing with interference: We wish to quantify the effect of imperfect measurement conditions. What is the effect of other processes executing concurrently with either the spy process or the target process? To quantify this, we start two interference processes. One executes on the same logical processor as the target, while the other executes on the same logical processor as the spy. The interference process can be configured to ask for a varying amount of CPU time, by executing a computation loop and a `usleep` period of configurable lengths. We vary the desired CPU time that

[2]https://github.com/bbbrumley/portsmash

TABLE V: Full, unaligned key recovery with ABSynthe with increasing amount of noise (concurrent computation) to both the target and the spy processes.

Platform	Target	Instr	Trials	Success	Med. BF (2^N)
Skylake	ED25519	DE1+N01	7	1.00	18.1
Skylake	ED25519	DE1+N02	7	0.71	18.1
Skylake	ED25519	DE1+N03	7	0.57	25.8
Skylake	ED25519	DE1+N04	7	0.14	36.1
Skylake	ED25519	DE1+N05	7	0.14	38.0
Skylake	ED25519	DE1+N06	7	0.00	-
Skylake	ED25519	DE1+N07	7	0.00	-
Skylake	ED25519	DE1+N08	7	0.00	-

the interference process asks for from 0.1% to 30.6% in 7 steps, and execute the measurement, training and key recovery procedures in each possible combination.

The results are visualized in Figure 15 with numbers in Table V. In Figure 15 we see that only interfering with the spy process has any effect, as this interferes with signal acquisition, but interfering with the target process does not have any significant effect, as the key recovery procedure is robust to noisy insertions in the signal. We show the numbers in Table V, where we interfere in steps with both the target and the spy process in equal amounts in increasing steps. We see that even minimal interference in the spy has an effect, and the success-rate gradually decreases as more interference is added. In summary, assuming the spy has complete control over its process, ABSynthe can successfully recover the secret keys even if the victim's execution is noisy.

G. Secret-Dependent Data Accesses

To demonstrate that the ABSynthe analysis pipeline is flexible enough to distinguish secrets based on secret-dependent data accesses, we briefly forego our purely blackbox philosophy (designed to target secret-dependent code accesses) and integrate in ABSynthe an active component that specifically targets different TLB data cache sets on the Skylake microarchitecture. We try all these measurement functions on all our targets with ASLR disabled this time. In this experiment, we seek to observe data accesses from the profiled secret-dependent branches (hence making the data accesses secret-dependent as well). This allows ABSynthe's dynamic taint analysis logic to find explicit secret-dependent data accesses (memory loads/stores with a tainted address) to extend the scope of the analysis. Table VI presents our results.

Our results show that, purely by observing cache accesses and having basic ground truth information, ABSynthe can distinguish between secrets. ABSynthe's performance is dependent on the target and its cache set number.

As expected, for the side channel safe implementation of EdDSA 25519 in libgcrypt 1.8.5, marked ED25519-secure in the table, the $f1$ score of at most 0.53 indicates ABSynthe cannot distinguish between different secrets.

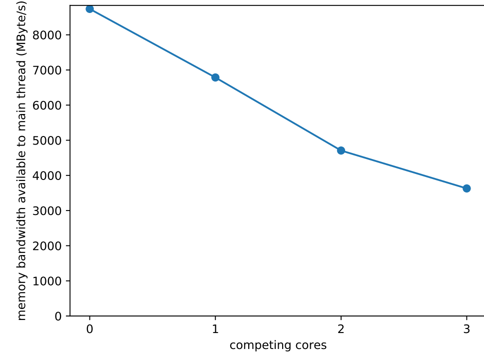


Fig. 16: DRAM bandwidth available to primary thread attempting to do maximum rate memory reads, as a function of number of competing physical cores trying to do the same. Clearly, the DRAM bandwidth is already halved when two cores are competing, and all bandwidth has to be shared between cores.

VIII. Discussion

A. Generalization

We designed ABSynthe to automate side-channel analysis, useful both for software security analysts and CPU designers in assessing side-channel leakage. The black box approach and high degree of automation benefit greatly from our focus on (1) contention-based side channels, and (2) resources that are shared between logical processors on a physical CPU core. We see two avenues towards the generalization of ABSynthe.

Eviction-based attacks As explained earlier in this paper, eviction-based attacks require reverse engineering of the internal state of the components involved, but this can be amenable to automation. For example, for resources such as the L1 cache that are set associative, we could incorporate a high-level model of their behavior in ABSynthe towards supporting "greybox" synthesis of eviction-based attacks.

Cross-core components ABSynthe can generalize beyond resources of a single core if we could extend our measurements beyond the core. We note that such cross-core and even cross-CPU components actually exist. As an example, Figure 16 shows that it is trivial to observe cross-CPU interference in DRAM bandwidth. The key challenge is to target a victim contending on DRAM accesses to find exploitable signals. Victim software with a sufficiently large working set (and normally accessing DRAM) is an obvious (but restrictive) candidate already at reach of ABSynthe. Using automatic synthesis of eviction-based attacks to create (otherwise-absent) contention on DRAM and other resources (at the cost of less stealthy attacks) is a promising direction for future research.

B. Limitations

Some parts of ABSynthe can be extended in the future to better support new software targets, architectures and key recovery.

Software targets We assume that the target software spend a significant amount of its time with secret computation. While

TABLE VI: Classification performance of ABSynthe when observing 16 different cache sets.

Target	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
GnuPG 2.2.17/gcrypt 1.6.3 25519	0.87	0.98	0.90	0.92	0.98	0.96	0.90	0.97	0.96	**0.99**	0.97	0.92	0.66	0.96	0.91	0.97
gcrypt 1.6.3 ECDSA P-256	0.94	0.96	0.97	**0.97**	0.93	0.91	0.73	0.96	0.51	0.92	0.96	0.97	0.81	0.97	0.89	0.91
gcrypt 1.6.3 EDDSA 25519	0.95	0.85	0.93	0.96	0.94	0.95	0.91	0.96	0.95	0.94	0.97	**0.99**	0.93	0.76	0.96	0.97
gcrypt 1.6.3 EDDSA 25519-hardened	0.65	0.69	0.73	0.80	0.75	0.73	0.66	**0.84**	0.71	0.69	0.82	0.72	0.71	0.73	0.78	0.74
gcrypt 1.6.3 RSA	0.59	0.54	0.57	0.58	0.58	0.52	0.58	0.58	0.56	0.54	0.51	0.57	0.56	0.57	0.57	**0.59**
gcrypt 1.8.5 ECDSA P-256	0.98	0.99	0.99	**0.99**	0.97	0.97	0.92	0.99	0.98	0.96	0.99	0.99	0.78	0.98	0.99	0.99
gcrypt 1.8.5 EDDSA 25519	0.95	0.83	0.96	0.98	0.95	0.96	0.95	0.98	0.97	0.96	0.96	**0.99**	0.79	0.98	0.98	0.99
gcrypt 1.8.5 EDDSA 25519-hardened	0.70	0.68	0.73	**0.83**	0.68	0.78	0.69	0.71	0.67	0.67	0.79	0.72	0.63	0.73	0.69	0.72
gcrypt 1.8.5 EDDSA 25519-secure	0.51	0.51	0.51	0.51	0.49	0.51	0.52	0.48	0.49	0.48	0.50	0.49	0.47	0.49	0.50	**0.53**
gcrypt 1.8.5 RSA	0.60	0.56	0.60	0.60	0.60	0.52	0.60	0.60	0.59	0.53	0.53	0.59	0.56	**0.71**	0.60	0.61

this is usually the case with cryptographic software as we showed in this paper, it may not necessarily be the case for other software targets. In those cases, an analyst may need to manually annotate the target software. We further assume that the secret key is loaded from the file system for our automated taint analysis. This can easily be extended to other sources of secret information that should be tainted (e.g., network sockets).

Architectures ABSynthe requires the ISA definition in a convenient format for building leakage maps for different microarchitectures. While this was readily available for the x86_64 [21], supporting new architectures in ABSynthe will require convenient ISA definitions. Future work can extend our x86_64 *leakage maps* to ARM using the ARM Machine Readable Architecture (MRA) [28].

Key recovery As we showed in our evaluation, our LSTM model achieves high F1 scores with non-cooperative victims on different microarchitectures. We presented an example to show that an analyst armed with ABSynthe's results, by performing tailored post-processing, can implement full key recovery from the ABSynthe-recovered key bit stream even with a single capture. A promising direction for future work is to also automate the post-processing, investigating brute-force heuristics that may apply to a wide variety of (cryptographic) programs.

IX. Related Work

Microarchitectural and physical side-channels Side channels have a rich history and have been applied to many different microarchitectural components. Cache attacks are the oldest and most widespread microarchitectural side-channel attacks. Such attacks originally targeted the L1 cache [3, 31, 32, 33], and more recently expanded to higher-level caches all the way to DRAM [34]. There are different eviction-based techniques to exploit cache side channels. The classic variant is PRIME+PROBE, which requires exact eviction sets but is the most general. Other variants are EVICT+TIME [3, 35, 36], which allows over-estimating an eviction set, FLUSH+RELOAD [1], which relies on shared physical memory but is high-resolution, and easy to use.

Non-software side channels are even older than microarchitectural attacks, and can use physical properties of the device that is doing the computation in order to leak secret information [37, 38].

The Branch Target Buffer (BTB) has also been heavily studied in prior work [8, 39, 40, 41, 42]. For instance, it allows ASLR information to leak from the kernel as well as from other processes. Recent work has also exploited the Pattern History Table (PHT) [9]. The PHT is shared across threads and, a spy thread can leak data by evicting PHT entries and deducing the direction of a particular branch. The Translation Lookaside Buffer (TLB) is also a shared resource. Prior work has shown that the L1 dTLB can be exploited for a reliable side-channel attack through the TLB [7] using a PRIME+PROBE-style attack. Memory Order Buffer (MOB) is yet another shared resource that can leak information by creating a false dependency across threads [43] and stalling the victim thread while the CPU decides whether store forwarding should proceed (in case of a true dependency).

The focus on contention-based side channels is recent and to our knowledge they have only been applied to execution ports. PortSmash [10] can leak cryptographic keys on Intel processors by creating contention on execution ports. Port contention has also been used to simplify gadgets [11] used in speculative execution attacks [44]. Previous work show the possibility of information leakage with port contention [42, 45]. Compared to these attacks that require highly specialized analysis that is often not portable to other (micro)architectures, ABSynthe can automatically synthesize contention-based side channels for a given software target and a microarchitecture. Instead of focusing on a single component (e.g., execution ports), ABSynthe automatically discovers the best set resources that leak information with a blackbox analysis.

Side-channel attack automation Other prior efforts have proposed systems to automate side-channel attacks, although none can support the blackbox synthesis strategy proposed in this paper. For instance, [46] focuses an automating side-channel attacks with a traditional side-channel analysis tailored to a specific microarchitectural component (last-level cache). Covert Shotgun [20] is more related in that it runs many combinations of instructions to automatically find covert channels. ABSynthe's contention-based side channel strategy draws inspiration from such approach, but covers the entire x86_64 ISA and synthesizes a side-channel attack rather than a much simpler covert channel.

While not aiming at automatic blackbox synthesis, other efforts have used machine learning techniques to ease side-channel exploitation, for instance to differentiate the key-dependent side channel signals from one another [7, 47, 48, 49]. Recent work has also applied deep learning techniques to in-browser cache fingerprinting attacks [50].

X. Conclusion

As a result of ever more advanced attacks, side-channel vulnerabilities have become important attack vectors in recent years. Most attacks such as PRIME+PROBE rely on targeted eviction operations on specific components (e.g., caches). Such whitebox attack strategies require a deep understanding of the target component, often involving labor-intensive reverse engineering that must be repeated for each microarchitecture.

In this paper, we created comprehensive leakage maps for on-core resources on three x86_64 microarchitectures. These leakage maps show the possibility of creating a variety of side-channel attacks by creating contention on a variety of microarchitectural components that are constantly and unavoidably used by victim code. We built ABSynthe based on this key observation for constructing powerful contention-based attacks in a black box, automated fashion without any need for labor-intensive reverse engineering. ABSynthe shows that simply treating the CPU as a black box and evaluating the information leakage across sequences of instructions is enough for crafting reliable side-channel attacks. Through extensive evaluation, we showed that ABSynthe can automatically craft practical side-channel attacks to recover key bit streams on different microarchitectures (Intel, AMD, ARM) and execution environments (native, virtualized) against a variety of software targets. We also presented a case study where an analyst armed with ABSynthe's results can recover the full secret key. ABSynthe can also be used by hardware designers for microarchitecture regression testing purposes (e.g., to automatically test whether new ISA extensions introduce new side channels) or by software designers to test for side-channel leakage.

Acknowledgements

`packet.net` and WorksOnArm gracefully sponsored the use of several of their bare metal machines, including the AMD EPYC machine and the Cavium Thunder X2 machine, for which the authors are very grateful. We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct) and No. 825377 (UNICORE), by Intel Corporation through the Side Channel Vulnerability ISRA, and by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing," NWO 639.021.753 VENI "PantaRhei," and NWO 016.Veni.192.262. As to the opinions and positions in this document that the authors express or to which the authors contributed, they are those of the authors and do not represent the views of any current or previous employer, including Intel Corporation or its affiliates. The same applies to the funding agencies, which are also not responsible for any use that may be made of the information the paper contains.

References

[1] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack." in *USENIX Security Symposium*, 2014, pp. 719–732.

[2] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+ abort: A timer-free high-precision l3 cache attack using intel tsx," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 51–67.

[3] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' Track at the RSA Conference*. Springer, 2006, pp. 1–20.

[4] N. Lawson, "Side-Channel Attacks on Cryptographic Software," in *IEEE Symposium on Security and Privacy*, 2009.

[5] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level Cache Side-channel Attacks are Practical," in *IEEE Symposium on Security and Privacy*, 2015.

[6] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack Directories, Not Caches: Side-Channel Attacks in a Non-Inclusive World," in *IEEE Symposium on Security and Privacy*, 2019.

[7] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security Symposium*, 2018.

[8] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–13.

[9] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, D. Ponomarev *et al.*, "Branchscope: A new side-channel attack on directional branch predictor." ACM, 2018, pp. 693–707.

[10] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *Security and Privacy (SP), 2019 IEEE Symposium on*. IEEE, 2019.

[11] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," *arXiv preprint arXiv:1903.01843*, 2019.

[12] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, "Data–differential address trace analysis: Finding address-based side-channels in binaries," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 603–620.

[13] R. Singhal, "Inside intel next generation nehalem microarchitecture," in *Hot Chips*, vol. 20, 2008, p. 15.

[14] Y. Tian, C. Lin, and K. Hu, "The performance model of hyper-threading technology in intel nehalem microarchitecture," in *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on*, vol. 3. IEEE, 2010, pp. V3–379.

[15] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty *et al.*, "Hyper-threading technology in the netburst® microarchitecture," *14th Hot Chips*, 2002.

[16] Intel, "Intel 64 and ia-32 architectures optimization reference manual," 2016.

[17] H. Wong, "Measuring reorder buffer capacity," http://blog.stuffedcow.net/2013/05/measuring-rob-capacity/, Accessed on 10.11.2018., may 2013.

[18] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 217–233.

[19] D. Genkin, L. Valenta, and Y. Yarom, "May the fourth be with you: A microarchitectural side channel attack

on several real-world applications of curve25519," in *Proceedings of the 2017 ACM SIGSAC*. ACM, 2017, pp. 845–858.

[20] A. Fogh, "Covert shotgun," september 2016. [Online]. Available: https://cyber.wtf/2016/09/27/covert-shotgun/

[21] A. Abel and J. Reineke, "uops. info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures," in *Proceedings of the Twenty-Fourth ASPLOS*. ACM, 2019, pp. 673–686.

[22] L. Team, "Dataflowsanitizer design document," https://clang.llvm.org/docs/DataFlowSanitizerDesign.html, Accessed on 03.01.2019., march 2019.

[23] J. Zhang, X. Lu, J. Jose, R. Shi, and D. K. D. Panda, "Can inter-vm shmem benefit mpi applications on sr-iov based virtualized infiniband clusters?" in *European Conference on Parallel Processing*. Springer, 2014, pp. 342–353.

[24] K. Price, R. M. Storn, and J. A. Lampinen, *Differential evolution: a practical approach to global optimization*. Springer Science & Business Media, 2006.

[25] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[26] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[27] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[28] A. Reid, "Trustworthy specifications of arm® v8-a and v8-m system level architecture," in *2016 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2016, pp. 161–168.

[29] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer, "Ecdh key-extraction via low-bandwidth electromagnetic attacks on pcs," in *Cryptographers' Track at the RSA Conference*. Springer, 2016, pp. 219–235.

[30] K. Okeya and T. Takagi, "The width-w naf method provides small memory and fast elliptic scalar multiplications secure against side channel attacks," in *Cryptographers' Track at the RSA Conference*. Springer, 2003, pp. 328–343.

[31] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel." *IACR Cryptology ePrint Archive*, vol. 2002, no. 169, 2002.

[32] C. Percival, "Cache missing for fun and profit," 2005.

[33] O. Aciiçmez, "Yet another microarchitectural attack: exploiting i-cache," in *Proceedings of the 2007 ACM workshop on Computer security architecture*. ACM, 2007, pp. 11–18.

[34] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "Drama: Exploiting dram addressing for cross-cpu attacks," in *USENIX Security Symposium*, 2016.

[35] R. Spreitzer and T. Plos, "Cache-access pattern attack on disaligned aes t-tables," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2013, pp. 200–214.

[36] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "Aslr on the line: Practical cache attacks on the mmu," *NDSS (Feb. 2017)*, 2017.

[37] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "Ecdsa key extraction from mobile devices via nonintrusive physical side channels," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1626–1638.

[38] G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo, "Aes power attack based on induced cache miss and countermeasure," in *International Conference on Information Technology: Coding and Computing (ITCC'05)-Volume II*, vol. 1. IEEE, 2005, pp. 586–591.

[39] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. ACM, 2007, pp. 312–320.

[40] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers' Track at the RSA Conference*. Springer, 2007, pp. 225–242.

[41] O. Acıiçmez, S. Gueron, and J.-P. Seifert, "New branch prediction vulnerabilities in openssl and necessary software countermeasures," in *IMA International Conference on Cryptography and Coding*. Springer, 2007, pp. 185–203.

[42] O. Acıiçmez and J.-P. Seifert, "Cheap hardware parallelism implies cheap security," in *Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop on*. IEEE, 2007, pp. 80–91.

[43] A. Moghimi, T. Eisenbarth, and B. Sunar, "Memjam: A false dependency attack against constant-time crypto implementations in sgx," in *Cryptographers' Track at the RSA Conference*. Springer, 2018, pp. 21–44.

[44] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[45] S. M. D'Antoine, "Exploiting processor side channels to enable cross vm malicious code execution," Ph.D. dissertation, Rensselaer Polytechnic Institute, 2015.

[46] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches." in *USENIX Security Symposium*, 2015, pp. 897–912.

[47] I. P. Samiotis, "Side-channel attacks using convolutional neural networks: A study on the performance of convolutional neural networks on side-channel data," *Delft University of Technology Master Thesis*, 2018.

[48] G. Hospodar, B. Gierlichs, E. De Mulder, I. Verbauwhede, and J. Vandewalle, "Machine learning in side-channel analysis: a first study," *Journal of Cryptographic Engineering*, vol. 1, no. 4, p. 293, 2011.

[49] T. Zhang, Y. Zhang, and R. B. Lee, "Analyzing cache side channels using deep neural networks," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 174–186.

[50] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," *arXiv preprint arXiv:1811.07153*, 2018.