Quantifying and Mitigating Cache Side Channel Leakage with Differential Set

CONG MA*, University of Waterloo, Canada
DINGHAO WU, Pennsylvania State University, USA
GANG TAN, Pennsylvania State University, USA
MAHMUT TAYLAN KANDEMIR, Pennsylvania State University, USA
DANFENG ZHANG, Duke University, Pennsylvania State University, USA

Cache side-channel attacks leverage secret-dependent footprints in CPU cache to steal confidential information, such as encryption keys. Due to the lack of a proper abstraction for reasoning about cache side channels, existing static program analysis tools that can quantify or mitigate cache side channels are built on very different kinds of abstractions. As a consequence, it is hard to bridge advances in quantification and mitigation research. Moreover, existing abstractions lead to imprecise results. In this paper, we present a novel abstraction, called differential set, for analyzing cache side channels at compile time. A distinguishing feature of differential sets is that it allows compositional and precise reasoning about cache side channels. Moreover, it is the first abstraction that carries sufficient information for both side channel quantification and mitigation. Based on this new abstraction, we develop a static analysis tool DSA that automatically quantifies and mitigates cache side channel leakage at the same time. Experimental evaluation on a set of commonly used benchmarks shows that DSA can produce more precise leakage bound as well as mitigated code with fewer memory footprints, when compared with state-of-the-art tools that only quantify or mitigate cache side channel leakage.

CCS Concepts: • **Security and privacy → Side-channel analysis and countermeasures**; • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: differential set, side channels, information flow

ACM Reference Format:
Cong Ma, Dinghao Wu, Gang Tan, Mahmut Taylan Kandemir, and Danfeng Zhang. 2023. Quantifying and Mitigating Cache Side Channel Leakage with Differential Set. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 274 (October 2023), 29 pages. https://doi.org/10.1145/3622850

1 INTRODUCTION

Program execution leaves footprints in CPU cache. Such footprints, when they are dependent on secret information, have long been leveraged in cache side-channel attacks to steal sensitive data from the physical implementations of cryptographic systems. A variety of powerful cache side-channel attacks have been demonstrated on ciphers such as RSA [Aciicmez and Seifert 2007; Percival 2005; Yarom and Falkner 2014], AES [Gullasch et al. 2011; Osvik et al. 2006; Tromer et al. 2010], and ElGamal [Liu et al. 2015; Zhang et al. 2012]. Cache side-channel attacks pose a serious

*The majority of this author's work was completed while as a student at Pennsylvania State University and was finished while at University of Waterloo.

Authors' addresses: Cong Ma, cong.ma@uwaterloo.ca, University of Waterloo, , Canada; Dinghao Wu, dinghao@psu.edu, Pennsylvania State University, , USA; Gang Tan, gtan@psu.edu, Pennsylvania State University, , USA; Mahmut Taylan Kandemir, mtk2@psu.edu, Pennsylvania State University, , USA; Danfeng Zhang, dz132@duke.edu, Duke University, Pennsylvania State University, , USA.

2475-1421/2023/10-ART274
https://doi.org/10.1145/3622850

threat to the security of many application domains, especially cloud computing. For example, researchers have confirmed that a remote attacker who only shares hardware resources with the confidential computation can quickly reveal private information [Brasser et al. 2017; Götzfried et al. 2017; Liu et al. 2015; Osvik et al. 2006; Percival 2005; Ristenpart et al. 2009; Schwarz et al. 2017; Van Bulck et al. 2017; Xiao et al. 2017; Zhang et al. 2012].

Since cache side-channels originate from secret-dependent memory footprints and cache effects, it is very challenging for programmers to identify such vulnerabilities in the source code. One well-known abstraction used in the development of cryptographic libraries, a common target of cache side-channel attacks, is called *constant-time programming paradigm* [Almeida et al. 2016]. Essentially, the constant-time programming paradigm prohibits both control-flow paths and memory-access patterns from depending on program secrets. This principle is used to implement almost all modern cryptography as (1) it protects software against *timing side-channel attacks* where an attacker can precisely measure the victim program's execution time, and (2) it is a simple abstraction whose violation can be detected via static program analysis, such as a static taint analysis. Various static analyses have been developed to detect violation of constant-time programming paradigm [Almeida et al. 2016; Cauligi et al. 2020; Daniel et al. 2020] and further, to mitigate detected violations via automatic code transformation [Borrello et al. 2021; Cauligi et al. 2019; Wu et al. 2012].

However, using sensitive control-flow paths and memory-access patterns as an abstraction to detecting and mitigating cache side channels has two limitations. First, while suitable for high assurance systems, ruling out all sensitive control-flow paths and memory-access patterns is either infeasible or has high overhead [Coppens et al. 2009], making it less appealing for other applications such as web application firewalls [Schmitt and Schinzel 2012] and web browsers [Felten and Schneider 2000; Jia et al. 2015; Kotcher et al. 2013; Stone 2013; Van Goethem et al. 2015]. Hence, it is appealing to *quantify* the information leakage of each program location and mitigate the most crucial ones. Unfortunately, the constant-time programming paradigm offers little information on how to quantify information leakage, a reason that quantification tools [Antonopoulos et al. 2017; Chen et al. 2017; Doychev et al. 2013; Doychev and Köpf 2017; Pasareanu et al. 2016; Phan et al. 2017] are all built on very different abstractions[1] of side channels. Second, the identified sensitive control-flow paths and memory-access patterns allow only imprecise mitigation. Consider an array access `A[k mod 2]` where `k` is confidential. Mitigation tools built on the constant-time programming paradigm lack information on *what* might be leaked, which is the parity of `k` in this example. Hence, they need to insert extra accesses to the whole array `A` to hide the sensitive access conservatively, while a more efficient mitigation only needs to access two elements `A[0]` and `A[1]`.

In this paper, we present a novel abstraction called *differential set* to reason about cache side channels at compile time. To the best of our knowledge, differential set is the *first* language-level abstraction that enables both automated quantification and mitigation of cache side channels. It is computed from *local* differential set for each memory access of a program. Intuitively, local differential set is a set of addresses that might be touched by either the memory access itself or its corresponding "sibling" memory accesses in other control flows (when there is a sensitive branch). As opposed to dynamic memory accesses, differential sets are defined as a compile-time concept that only depends on code structure. Hence, it allows static and compositional reasoning about cache side channels. Moreover, we show that the differential set of a program, a combination of its local differential sets, soundly approximates cache side channels in a program and furthermore, it enables precise quantification and mitigation of them. For example, an array access `A[k mod 2]`, where `k` is confidential, has a differential set of $D = \{\texttt{A[0]}, \texttt{A[1]}\}$ if the access is not under sensitive branches.

[1]However, those abstractions fail to localize side channel leakage and thus cannot offer sufficient information to aid mitigation.

Based on the result, it is straightforward to quantify its side channel leakage as $\log_2 |D| = 1$ bit, and further, mitigate the side channel precisely via always accessing both `A[0]` and `A[1]` in the transformed code where side channels are removed.

Further, we develop a static analysis tool *DSA (Differential Set Analyzer)* to automatically compute differential sets and utilize the information to automatically quantify and mitigate cache side channels. The core of DSA is a symbolic representation of differential sets as well as a novel alignment algorithm that leads to more efficient mitigation. In order to show the precision of leakage bound as well as the efficiency of mitigated programs produced by DSA, we evaluate DSA on a set of benchmarks used in prior work [Antonopoulos et al. 2017; Borrello et al. 2021; Doychev and Köpf 2017]. Compared with state-of-the-art tools that can *only* quantify [Doychev and Köpf 2017] or mitigate [Borrello et al. 2021] cache side-channel leakage, DSA provides the same or tighter leakage bounds in all cases and its mitigated code contain fewer memory accesses. Although DSA's static analysis is usually slower than existing tools, it analyzes all benchmark programs under 20 minutes, and the cost is only paid (for better precision) at compile time.

To summarize, the contributions of this work include:

- Differential set, a novel language-level abstraction for reasoning about cache side channels in a compositional and precise fashion (Section 4).
- DSA, the first static program analysis tool that can both automatically quantify and mitigate cache side channels in a program (Section 5).
- Evaluation on common benchmark programs for quantification and mitigation of side channels. In addition to being able to both quantify and mitigate side channels, DSA is able to produce the same or tighter leakage bounds in all cases and its mitigated code contains fewer memory accesses when compared with state-of-the-art tools (Section 6).

2 BACKGROUND AND THREAT MODEL

2.1 Cache Side Channel

Side channels are information channels that were not intended to convey information. In this paper, we consider cache side channels where a program reveals secret information via its CPU cache usage. We use the following code snippet to illustrate two kinds of cache side channels: sensitive control-flow paths and sensitive memory-access patterns. In this example, we assume that both `s1` and `s2` are secret inputs, `A` is an array storing public data, and cache line size is 64 bytes.[2]

```
1  char A[128];
2  if (s1==0)
3    x = A[0];
4  else
5    x = A[64];
6  x = A[s2];
```

Sensitive control-flow path. A sensitive control-flow path occurs when a branch condition's value depends on secrets, such as the branch at line 2 above. In this case, given two different secret values of `s1`, the program above stores data into two different cache lines corresponding to `A[0]` and `A[64]` respectively. When the branches contain longer code fragments, a sensitive branch might also store instructions into two different cache lines in the instruction cache. Although a cache-based attacker that shares CPU cache with the victim program is unable to observe the cache line being

[2]Although we use this contrived example for its simplicity, we emphasize that the secret-dependent branch at line 1 mimics real side channels in the modular exponentiation computation in RSA and ElGamal [Liu et al. 2015; Zhang et al. 2012]. Moreover, the secret-dependent array access at line 5 mimics real side channels in the block ciphers such as AES [Gullasch et al. 2011; Osvik et al. 2006; Tromer et al. 2010].

accessed directly, the attacker can learn the value of `s1` through techniques such as *prime and probe* [Bonneau and Mironov 2006; Liu et al. 2015; Osvik et al. 2006; Tromer et al. 2010; Zhang et al. 2012]: the attacker first primes cache sets being monitored (e.g., the two cache lines above) by filling the shared cache sets with his data and then probes the data again after the victim program runs. A cache line that belongs to the same cache set that is accessed by the victim introduces a longer latency in the probing step as the victim program evicts attacker's data in the cache.

Sensitive data-access pattern. A sensitive data access occurs when a program accesses a memory address that depends on secrets, such as line 6 in the code snippet above. As the value of `s2` decides which cache line is being accessed at line 6, a cache-based attacker (e.g., one who uses the prime and probe technique discussed above) can also reveal some information about the value of `s2` via revealing which cache line is being accessed by the program.

2.2 Threat Model

We assume a cache-based adversary who has no direct access to the victim program but is co-located on the same physical machine, and hence, share CPU cache (e.g., the second-level cache in some architectures and third-level cache in almost all architectures) with the victim. To make the threat model architecture-agnostic, we further assume a *trace-based model* [Doychev et al. 2013] where an attacker learns the full trace of memory addresses (which includes data addresses, and instruction addresses if instruction cache is a concern) being issued by the victim program. The threat model reflects powerful synchronous cache attacks [Gullasch et al. 2011; Liu et al. 2015; Yarom and Falkner 2014] where an attacker learns the shared cache state after each command in the victim program.

Since we focus on cache side channels, our threat model is a bit weaker than the threat model behind the constant-time programming paradigm, which assumes the timing side channel. The difference is that the latter also assumes that the adversary can directly control the victim program's execution and precisely measure its execution time. But, the cache-based model more precisely reflects more realistic cache attacks launched in cloud environments [Liu et al. 2015; Ristenpart et al. 2009; Schwarz et al. 2017; Wu et al. 2012; Xiao et al. 2017; Xu et al. 2011; Zhang et al. 2012], an emerging threat to cloud computing. Further, our threat model is also consistent with prior work on cache side-channel analysis [Brotzman et al. 2019, 2021; Doychev et al. 2013; Doychev and Köpf 2017; Wang et al. 2019, 2017].

Finally, we note that our threat model is stronger than most program analysis that quantifies timing channel leakage (i.e., leakage via program execution time), as they [Antonopoulos et al. 2017; Chen et al. 2017; Pasareanu et al. 2016; Phan et al. 2017] do *not* model CPU cache. Instead, they simply assume program execution time is described by the number of executed instructions.

3 OVERVIEW

We use the program in Figure 1 as a running example throughout this paper. This example demonstrates common side-channel leakage that appear in real-world cases (as discussed in Section 2). The program takes in a secret array `in`, a secret input `threshold`, and it copies elements of the array `in` to an output array `out` in a loop. If element `in[i]` is smaller than the secret threshold, the element is placed at the beginning of `out`; otherwise, it is placed at the end of `out`. Local variables `j0` and `j1` represent, respectively, the current insertion point at the beginning and end of `out`.

Note that the program contains one sensitive control-flow path at line 8 and two sensitive array accesses (`out[j0]` and `out[j1]`) at lines 9 and 11. They both can leak information as demonstrated by two sets of inputs and their corresponding access traces to array `out`:

$$\texttt{threshold} = 2,\ \texttt{in} = \{0,1,2,3\} \hookrightarrow \texttt{out[0]..out[1]..out[3]..out[2]}$$

$$\texttt{threshold} = 3,\ \texttt{in} = \{4,3,2,1\} \hookrightarrow \texttt{out[3]..out[2]..out[0]..out[1]}$$

```
1  #define SIZE 4 // SIZE is a constant
2  int main(int in[SIZE], int out[SIZE],
3           int threshold) {
4      j0 = 0;
5      j1 = SIZE-1;
6      #pragma unroll // loop is unrolled
7      for (i = 0; i < SIZE; i++) {
8          if (in[i] < threshold) {
9              out[j0++] = in[i];
10         } else {
11             out[j1--] = in[i];
12         }
13     }
14 }
```

Fig. 1. Running example. The actual program under static analysis has the loop unrolled.

```
1  i = 0
2  t = in[i] < threshold
3
4  ptr_0 = ct_select(t, &out[j0], ⊥)
5  ct_store(ptr_0, in[i], [out])
6
7  ptr_1 = ct_select(!t, &out[j1], ⊥)
8  ct_store(ptr_1, in[i], [out])
9
10 j0 = ct_select(t, j0+1, j0)
11 j1 = ct_select(t, j1, j1-1)
```

Fig. 2. Mitigated code produced by Constantine

Note that we omit insensitive array accesses in the example traces since they are identical, regardless of the inputs. Moreover, we omit accesses to the instruction cache as instructions at $L9$ and $L11$ are likely being allocated to the same cache line. By probing whether the offset to array `out` is at the beginning or at the end, an attacker learns whether each element is smaller or greater than the threshold. For example, the first trace reveals that the first two values of array `in` are *below* the threshold, while the second trace reveals that the first two values are *above* the threshold.

Beyond telling whether a program leaks information via cache side channel or not, we also tackle the following two challenging questions in this paper:

(1) How much information is leaked via the cache side channel?
(2) How can we automatically eliminate cache side channel leakage (if any)?

3.1 Quantification of Cache Side Channels

To quantify leakage, we adopt *Channel Capacity*, a widely-used measure of information leakage in the literature [Lowe 2002; Newsome et al. 2009; Smith 2009], which is defined as the logarithm (with base 2) of the total number of possible access traces that can be produced by a program. In this example, during each iteration there is always a write to the array `out` either at the beginning or at the end, depending on the input; all other memory operations are the same regardless of the input. Therefore, each iteration induces two different sub-traces. Moreover, during the last iteration we have $\texttt{j0} = \texttt{j1}$, so it induces only one sub-trace. Since the code branches on a different input in every iteration, we can multiply the possibilities of sub-traces to get the possibilities of the whole trace, and conclude that the leakage is $\log_2(2^{\texttt{SIZE}-1}) = \texttt{SIZE} - 1$ bits.

To compute channel capacity, a naive approach is to enumerate and count the number of *all sequences* of memory traces. However, such an approach is computationally infeasible (e.g., even the running example has $2^{\texttt{SIZE}}$ possible execution paths). Observing this, state-of-the-art methods use over-approximation to estimate the channel capacity. For example, CacheAudit [Doychev et al. 2013; Doychev and Köpf 2017] uses a set of possible cache states at each program point as the abstraction of analyzing cache side channel leakage: given N possible cache states at the end of program execution, the leakage is bounded by $\log_2(N)$. The abstraction has two limitations:

(1) CacheAudit analyzes programs under *abstract program states* and *abstract cache states* to trade precision for performance. After the first iteration of code in Figure 1, for example, there are two possible program states: either one with $\texttt{j0} = 1 \wedge \texttt{j1} = 3$ or one with $\texttt{j0} = 0 \wedge \texttt{j1} = 2$. Hence, the tool approximates the concrete states as $\texttt{j0} \in \{0, 1\} \wedge \texttt{j1} \in \{2, 3\}$, and it incorrectly

determines that the second iteration can induce 4 unique sub-traces, whereas in reality, there are only 2 possible sub-traces. Due to the imprecision, CacheAudit reports 384 unique traces (i.e., 8.6 bits of leakage) while the true leakage is only of 3 bits.

(2) The number of possible cache states at each program point provides little insights on how to mitigate the identified side channels. Most importantly, it does not pinpoint at the root cause of the identified leakage (i.e., the program locations that introduce cache side-channel leakage).

3.2 Mitigation of Cache Side Channels

Instead of using approximation of memory access traces, most state-of-the-art tools that automatically mitigate cache side channels are built on the constant-time programming paradigm: they identify sensitive control-flow paths and sensitive memory access patterns (Section 2.1) in code separately and then rewrite the code in a semantic-preserving manner to eliminate the violations.

For example, a recent work, Constantine [Borrello et al. 2021], employs a *dynamic* taint analysis to identify violations of constant-time programming paradigm and then rewrites the offending code in the following way. For sensitive control-flow paths, it uses *control flow linearization* technique to rewrite the code so that both branches are executed while ensuring that only the correct branch updates memory state. For sensitive memory accesses, it uses *data flow linearization* technique which directly hardens each vulnerable read/write operation by touching all possible memory locations that the pointers used by the operation might point to (via a points-to analysis). We show the pseudo-code of our running example after its first iteration is mitigated by Constantine in Figure 2, where each vulnerable store operation is replaced with a conditional assignment `ct_select` (e.g., line 4) and a secure memory subroutine `ct_store` (e.g., line 5). The subroutine `ct_store` strides through the memory objects specified in the third argument, and only performs memory store to the address specified by the first two arguments.

Using sensitive control-flow paths and sensitive memory access patterns as the abstraction to analyzing cache side channels has three limitations:

(1) While the abstraction can localize where information is leaked via cache side channels, it is too coarse-gained and localized for analyzing the overall leakage of a piece of code.

(2) Since sensitive control-flow paths cannot tell the *common* memory accesses in both branches, control flow linearization requires executing all code in both branches. The result is that array `out` is accessed *twice* in the mitigated code.

(3) Since sensitive memory access patterns cannot tell which array elements *might be accessed* at a program point, data flow linearization requires accessing the *whole* array `out`. However, as discussed above, only the first and last elements need to be touched *once* in order to hide the sensitive memory address.

3.3 Differential Set

In this paper, we introduce a novel concept called *Differential Set* and discuss how it can be used to automatically detect, quantify and mitigate side channels in a program. We tackle the following limitations of existing abstractions for analyzing cache side channels:

(1) It is computationally infeasible to compute memory access traces, since the number of traces is at least proportional to the execution paths the program can take, which can be exponential to the size of a program. As a consequence, existing tools sacrifice precision in other aspects of program analysis to reduce the complexity, as discussed in Section 3.1.

(2) Sensitive control-flow paths and sensitive memory access patterns are easier to compute, but they only provide coarse-grained cache side-channel information. As a consequence,

Table 1. Differential sets in the first loop iteration.

Observation Point	Memory Access	Local Differential Set
1	`in[i]`@line 8	{⟨`in[0]`, `true`⟩}
2	`in[i]`@line 9 `in[i]`@line 11	{⟨`in[0]`, `true`⟩}
3	`out[j0]`@line 9 `out[j1]`@line 11	{⟨`out[0]`, ⟦`in[0]` < `threshold`⟧⟩, ⟨`out[3]`, ⟦`in[0]` ≥ `threshold`⟧⟩}

mitigation tools using the abstraction need to insert extra memory accesses in a conservative way, as discussed in Section 3.2.

(3) Memory access traces provide little information for mitigation of cache side channels, while sensitive control-flow paths and sensitive memory access patterns are too coarse-grained and localized for quantification of cache side channels.

To tackle the challenges, we present *Cache Differential Set* (or Differential Set for short) as a more appropriate abstraction for analyzing cache side channels. We give an informal treatment in this section, and provide the formal definition in Section 4. The differential set of a program is a collection of *local* differential sets of all memory accesses of a program. Intuitively, each local differential set of a memory access is a set of addresses that might be touched by either the memory access itself or its "sibling" memory accesses in other control flows (when there is a sensitive branch). To enable composition, each set of addresses is also associated with its path condition (i.e., a set of initial memories that reaches the memory access).

Consider the first iteration of the loop in the running example, which has 5 memory accesses. We summarize in Table 1 the local differential set of each memory access (in the first loop iteration) along with their corresponding differential set in the running example. The access to array `in` at line 8 has a local differential set of size one as intuitively, (1) it is not inside a sensitive branch so it has no siblings, and (2) its address does not depend on any secret value. Hence, the only element in its local differential set is a pair ⟨`in[0]`, `true`⟩ where `in[0]` is the memory address that it accesses and `true` is its path condition (i.e., it is executed unconditionally).

On the other hand, the other 4 memory accesses all have siblings due to the sensitive branch at line 8. Among those, the two accesses to `in[i]` at lines 9 and 11 have local differential sets of size one as their siblings access the same address. They are grouped into the same *observation point* since they are both the first memory access after the branch at line 8. Moreover, the path condition of the local differential set is `true`, the logical or of the path conditions of `in[i]` at lines 9 and 11. However, the two accesses to array `out` at lines 9 and 11 have local differential sets of size two as their siblings access different addresses. They are grouped into the same observation point since they are both the second memory access after the branch at line 8. Since they access different addresses, their path conditions cannot be merged in the local differential set.

Compared with existing approaches, the major benefits of differential set are two-folded. First, differential set allows precise and compositional leakage quantification. For each memory access in the source code, its differential set D provides an upper bound ($\log_2 |D|$) on how much information is leaked by the access per se. In the running example, only accesses to array `out` leak one bit of information. Moreover, by concatenating local differential sets, we can quantify the leakage of any program segment. For instance, in the first iteration of the running example, there are two possible memory access traces, namely, {`in[0]`, `in[0]`, `out[0]`} and {`in[0]`, `in[0]`, `out[3]`}, which yields one bit of leakage. Note that although in this example, all combinations of local differential set are possible, in general, we can eliminate impossible combinations by the path conditions to compute a precise leakage bound. For instance, consider the first two iterations of the running example. The first iteration's access to `out` has the local differential set as

Expressions
$$E ::= n \mid X \mid A[E] \mid *X \mid E_1 \otimes E_2$$
Boolean Expressions
$$B ::= E_1 \odot E_2 \mid \neg B \mid B_1 \wedge B_2 \mid B_1 \vee B_2$$
Statements
$$S ::= \texttt{skip} \mid X := E \mid *X := E \mid A[X] := E \mid S_1; S_2 \mid \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2$$

where $\otimes$ represents binary arithmetic operations, and $\odot$ represents binary comparison operators

Fig. 3. Source Language Syntax.

$$D_1 = \{\langle \texttt{out[0]}, [\![\texttt{in[0]} < \texttt{threshold}]\!]\rangle, \langle \texttt{out[3]}, [\![\texttt{in[0]} \geq \texttt{threshold}]\!]\rangle\}$$

while the corresponding access in the second iteration has

$$\begin{aligned} D_2 = \{&\langle \texttt{out[0]}, [\![\texttt{in[0]} \geq \texttt{threshold}]\!] \wedge \texttt{in[1]} < \texttt{threshold}]\!]\rangle, \\ &\langle \texttt{out[1]}, [\![\texttt{in[0]} < \texttt{threshold}]\!] \wedge \texttt{in[1]} < \texttt{threshold}]\!]\rangle, \\ &\langle \texttt{out[2]}, [\![\texttt{in[0]} \geq \texttt{threshold}]\!] \wedge \texttt{in[1]} \geq \texttt{threshold}]\!]\rangle, \\ &\langle \texttt{out[3]}, [\![\texttt{in[0]} < \texttt{threshold}]\!] \wedge \texttt{in[1]} \geq \texttt{threshold}]\!]\rangle\} \end{aligned}$$

Among all $|D_1| \times |D_2| = 8$ combinations, 4 combinations (e.g., `out[0]` followed by `out[2]`) are impossible due to incompatible path conditions. Therefore, differential sets allow us to precisely quantify information leakage from both a single access, multiple accesses, and all accesses in a program in a compositional way.

Second, differential sets allow efficient mitigation of cache side channels. The reason is that by definition, a local differential set contains all possible memory addresses that might be accessed at the corresponding position in other executions where secret varies. Consider the local differential sets in Table 1. The first three accesses to array `in` always access `in[0]` regardless of the secret inputs. Hence, no mitigation is needed. The last two accesses to array `out` leak information. But, since they either access `out[0]` or `out[3]`, we can instrument a secure subroutine to sweep through all elements in its local differential set (i.e., `out[0]` and `out[3]`) in both branches to eliminate the cache side channel. Therefore, each branch has one extra memory access after mitigation, compared with accessing all four elements in array `out` twice by Constantine, as illustrated in Figure 2.

4 DIFFERENTIAL SET

In this section, we formally define differential set and prove that it provides a sound approximation of cache side channel leakage.

4.1 Execution Trace

We first formalize language and attack model to reason about cache side channels. We define a simple imperative language whose syntax is shown in Figure 3. For expressions, we use n as a numerical constant, X as a program variable, $A[E]$ as an access to an array with a base address of A at offset E and $*X$ as dereferencing memory. For commands, the simple language contains standard commands such as assignments, sequential composition and branches. `skip` represents a no-op. Note that the source language does not include loops; we assume that loops in the source code are unrolled before the analysis. Moreover, we assume that each memory access in the source code, e.g., `in`$[i]$ in the running example, has a unique identifier denoted as η.

We define a configuration as a pair $\langle m, S\rangle$ that consists of a memory m (i.e., a mapping from variables/memory locations, including array elements, to their values) as well as S, the remaining

program to be executed. Recall that a program leaks information through a cache side channel when it leaves different memory footprints in cache when running on different secret values. Hence, each small-step evaluation rule has the form of $\langle m, S\rangle \xrightarrow{e} \langle m', S'\rangle$ where event e tracks the memory addresses being accessed by the evaluation step, as well as their corresponding memory access IDs. When instruction cache is relevant, event e also includes branch labels (i.e., true/false branches). For example, the first iteration of the running example in Figure 1 emits true/false branch labels (i.e., tags $L9$/$L11$) when the branch outcomes are true/false respectively.

We consider two different observation models: data-only model where only data addresses are emitted in traces, and data+instruction model where both data addresses and branch tags are emitted in traces. The full semantics with the latter is provided in the Appendix B. We assume data-only model in most examples for its simplicity; we use the more secure data+instruction model in the evaluation.

Finally, we write $\langle m, S\rangle \hookrightarrow \tau$ if program S generates a memory trace τ (i.e., a sequence of events) when it is executed under m. We use $\tau \downarrow_{addr}$ to denote the subtrace of addresses.

4.2 Quantifying Cache Side Channel

Intuitively, a cache side channel leaks information when given the same values of *public inputs*, it produces different memory footprints (i.e., different traces) when secret inputs vary. The more unique traces are possible, the more information is leaked. To reflect the distinction between secret and public inputs, we write the public input variables as L and the secret input variables as H.

To quantify cache side channel leakage, we first introduce a *valuation* of public inputs L, written as $\mathsf{V} : L \to \mathfrak{R}$. Moreover, we say an initial memory m *agrees with* a valuation V, written as $m \sqsupset \mathsf{V}$, if m agrees with V on all public inputs:

$$m \sqsupset \mathsf{V} \iff \forall x \in L.\ m(x) = \mathsf{V}(x)$$

Channel capacity, a concept well-studied in information theory, quantifies the maximum amount of information leakage. Note that due to public inputs, we need to consider the *maximum* leakage among all valuations of public inputs [Doychev et al. 2013].

Definition 1 (Side Channel Leakage of a Program).

$$CC(S, L) = \log_2 \Big(\max_{\mathsf{V}:L\to\mathfrak{R}} |\{\tau \downarrow_{addr} | \ \exists m.\ m \sqsupset \mathsf{V} \wedge \langle m, S\rangle \hookrightarrow \tau\}| \Big)$$

Consider ($\texttt{if}\ x = 0\ \texttt{then}\ A[0] = 1\ \texttt{else}\ A[4] = 1$) as program S. We have $CC(S, \{x\}) = \log_2 1 = 0$ since for any valuation of public input x, the branch outcome must be identical, resulting in the same address trace. In other words, the program leaks no information if x is public. On the other hand, $CC(S, \{\}) = \log_2 2 = 1$ as the empty valuation puts no restriction on the value of x, resulting in two possible address traces with $A[0]$ and $A[4]$, respectively. In other words, the program leaks 1 bit of information when x is secret.

4.3 Differential Set

While Definition 1 measures the side-channel leakage of a program, enumerating all traces that can be produced by a program is, most of the time, infeasible. Instead, we employ a novel language abstraction called differential set. Like channel capacity, differential set is *parameterized* on a valuation V of public inputs L, which we use as implicit parameters throughout the section.

What makes differential set an appealing language abstraction for code analysis is its two main ingredients that are available locally to each memory access: *access set* of each memory access, and its "sibling" memory accesses. In short, access set defines the set of addresses that a single memory access *itself* can access when secrets differ; a memory access's "sibling" memory accesses

are the ones at the same distance (measured by the number of memory accesses) to a branch whose outcome depends on secret inputs. Based on those two ingredients, we define *local* differential set of a memory access as the set union of all access sets of itself *and its siblings* and moreover, (global) differential set as the combination of all local differential sets. Finally, we show that differential set leads to a sound approximation of side channel leakage (Theorem 1).

4.3.1 Access Set. For each memory access with ID η in the source code, we define its access set as the set of memory addresses that it can access with a given valuation V. Moreover, to allow composition that we discuss shortly, each address α in the access set is associated with a path condition PC. The path condition is represented by a set of initial memories that agrees with V and leads to an execution where the address α is accessed at η. Formally,

Definition 2 (Access Set (AS)). $\mathrm{AS}(\eta, \mathsf{V}) = \bigcup_\alpha \{\langle \alpha, \mathrm{PC}(\eta, \alpha, \mathsf{V})\rangle \mid \mathrm{PC}(\eta, \alpha, \mathsf{V}) \neq \emptyset\}$
where $\mathrm{PC}(\eta, \alpha, \mathsf{V}) = \{m \mid m \sqsupset \mathsf{V} \wedge \langle m, S\rangle \hookrightarrow \tau \wedge \langle \alpha, \eta\rangle \in \tau\}$.

Note that when η is never executed under any initial memory that agrees with V, $\mathrm{AS}(\eta, \mathsf{V}) = \emptyset$ (i.e., η is irrelevant to V). For example, consider ($\texttt{if}\ x = 0\ \texttt{then}\ A[0] = 1\ \texttt{else}\ A[4] = 1$), $L = \{x\}$ and $\mathsf{V} = \{x \mapsto 0\}$. We have $\mathrm{AS}(A[4] = 1, \mathsf{V}) = \emptyset$ since the false branch is never executed for any initial memory where public variable $x = 0$.

Consider the first iteration of the running example in Figure 1. The AS of `out[j0]` at line 9 is $\{\langle \texttt{out[0]}, [\![\texttt{in[0]} < \texttt{threshold}]\!]\rangle\}$ (we use the predicate on inputs to represent the path condition), and the AS of `out[j1]` at line 11 is $\{\langle \texttt{out[3]}, [\![\texttt{in[0]} \geq \texttt{threshold}]\!]\rangle\}$. For simplicity, we omit the path condition in the paper when it is irrelevant and simply write {`out[0]`} and {`out[3]`} as AS.

In data+instruction model, the branch instruction at Line 8 has instruction label $L9$ (resp. $L11$) in AS when the true (resp. false) branch is taken, as instruction is also fetched from memory.

4.3.2 Equi-Effect Form and Local Differential Set. Intuitively, access set models cache side channels due to data flows. To capture all cache side channels, we also need to model the ones due to control flows: the set of addresses being accessed by "sibling" memory accesses if the control flow differs. However, since different control flows might lead to different numbers of memory accesses, finding such "sibling" memory accesses is a challenge. To tackle the challenge, we introduce a code format called *equi-effect form*:

Definition 3 (Sensitive If-Statement). *An if-statement* $\texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2$ *in a program* S *is sensitive under valuation* V *if there are two initial memories* $m_1 \sqsupset \mathsf{V}$ *and* $m_2 \sqsupset \mathsf{V}$*, such that* S_1 *and* S_2 *are executed when* S *is executed under* m_1 *and* m_2 *respectively.*

Definition 4 (Equi-Effect Form). *An if-statement* ($\texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2$) *is in an equi-effect form if* S_1 *and* S_2 *contain the same number of memory accesses among all control-flow paths. A program is in an equi-effect form if every sensitive if-statement in the program is equi-effect*[3].

For example, a one-line program ($\texttt{if}\ x = 0\ \texttt{then}\ A[0] = 1\ \texttt{else}\ (A[4] = 1; A[8] = 1)$) is in an equi-effect form when $L = \{x\}$ since the only branch depends on public variable x. However, the same program is not in an equi-effect form when $L = \emptyset$ since the true and false branches have different number of memory accesses. While most source programs are not in equi-effect form, transforming an arbitrary program into equi-effect form is simple. To do so, we extend the source language with a special command `hole`, which represents a dummy memory access; it is used to pad shorter branches to make them equi-effect. Using the special command, we can pad a sensitive branch by first balancing inner-most nested branches and then pad outer ones recursively until the sensitive branch

[3]Note that with any over-approximation of sensitive if-statement (e.g., from a static taint analysis), the resulting equi-effect program is still equi-effect according to the most precise semantical definition in Definition 3.

is properly balanced. For example (`if` $x = 0$ `then` $(A[0] = 1;$`hole`$)$ `else` $(A[4] = 1; A[8] = 1)$) is in a equi-effect form when $L = \{\}$ since both branches are balanced.

In general, a program can be transformed into many equi-effect forms. We prove shortly that the particular form does not affect the soundness of differential set: it always provides an upper-bound of cache side-channel leakage (Theorem 1). However, the placement of holes in an equi-effect program does influence the performance of the mitigated program, which we explore in Section 5.2.

Sibling Accesses. We first note that an equi-effect program always produces the same number of memory accesses (including holes) with any valuation of public inputs:

Corollary 1. *For any equi-effect program and any valuation of public inputs, the program execution emits the same number of memory accesses regardless of secret inputs.*

Proof. By structural induction on program S. The interesting case is an if-statement. When the branch is sensitive, the result is implied by Definition 4 and induction hypothesis. Otherwise, the same branch is taken given a valuation of public inputs regardless of secret inputs. □

Now, we can define "sibling" memory accesses (in an equi-effect program) more precisely: two memory accesses are *siblings* if they appear at the same position of execution traces. Due to Corollary 1, it is equivalent to the following definition, which is more structural:

Definition 5 (Sibling memory accesses). *Within an equi-effect program, two memory accesses η_1 and η_2 are siblings, written as $\eta_1 \sim \eta_2$, if there exists a sensitive if-statement* (`if` B `then` S_1 `else` S_2) *such that $distance(B, \eta_1) = distance(B, \eta_2)$, where the distance of any two program points is the number of memory accesses between them.*

Consider (`if` $x = 0$ `then` $(A[0] = 1;$`hole`$)$ `else` $(A[4] = 1; A[8] = 1)$) which is in an equi-effect form when $L = \{\}$. Memory accesses `A[0]` and $A[4]$ are siblings, and moreover, `hole` and $A[8]$ are siblings. Consider (`if` $x = 0$ `then` (`if` $y = 0$ `then` $A[0] = 1$ `else` $A[4] = 1$) `else` $A[8] = 1$) where $L = \{\}$ (i.e., both x and y are secrets). In this example, memory accesses `A[0]`, `A[4]` and $A[8]$ are siblings as their distances to the branch with condition $x = 0$ are all 1. Note that by definition, a memory access that is outside any branch or only appears in public branches has no siblings. However, we make the sibling relation reflexive, i.e., $\forall \eta.\ \eta \sim \eta$ for technical convenience.

Local Differential Set. For each memory access η, its *local differential set* with respect to a valuation V, is the set of memory addresses that itself or its siblings might access:

Definition 6 (Local Differential Set). *Within an equi-effect program, the* local *differential set of a memory access η is: $LDS(\eta, \mathsf{V}) = \bigcup_{\eta \sim \eta'} AS(\eta', \mathsf{V})$*

Recall that sibling memory accesses are the ones that appear at the same position of execution traces of an equi-effect program. Hence, it captures all side channels produced *at* program point η due to both data flows and control flows. In other words, if we instrument a program such that it accesses all addresses in $LDS(\eta, \mathsf{V})$ when η is executed, all cache side channels are eliminated. Consider the running example in Figure 1. Since `out[j0]`@line 9 and `out[j1]`@line 11 are siblings, their local differential sets are both $\{\langle$`out[0]`, $[\![$`in[0] < threshold`$]\!]\rangle\} \bigcup \{\langle$`out[3]`, $[\![$`in[0] ≥ threshold`$]\!]\rangle\}$, as summarized in Table 1. If we rewrite both line 9 and line 11 to access both `out[j0]` and `out[j1]` in sequence, then in the instrumented code, the same sequence of memory addresses, namely `in[0]`, `in[0]`, `out[0]` and `out[3]`, will be accessed in both branches, thus removing the leakage.

4.3.3 Differential Set. Local differential set allows us to quantify and mitigate side channels at a single program point η. To quantify the side channel leakage of a whole program or a program

fragment, we define (global) differential set of a program S, which is intuitively a combination of local differential sets of memory accesses in S.

We first define an *observation point* o to be a maximal set of sibling memory accesses (i.e., $\forall \eta, \eta' \in o.\ \eta \sim \eta'$ and $\forall \eta, \eta'.\ \eta \in o \wedge \eta \sim \eta' \Rightarrow \eta' \in o$) and use $\mathcal{OP}$ to represent all unique observation points in an equi-effect program. Additionally, we use o_i where $1 \leq i \leq |\mathcal{OP}|$ to refer to each observation point. For example, Table 1 shows three observation points in the running example, where the second and third observation points contain two memory accesses each.

To formalize the set of possible memory accesses issued by an observation point, we lift the local differential set definition to an observation point o as follows:

$$LDS^o(o, \mathsf{V}) \triangleq \langle \diamond, \overline{\cup_{\langle a, pc\rangle \in G(o,\mathsf{V})} pc} \rangle \cup G(o, \mathsf{V}) \text{ where } G(o, \mathsf{V}) = \bigcup_{\eta \in o} \{AS(\eta, \mathsf{V})\}$$

Note that we use a special memory access $\diamond$ to represent the *absence* of the observation point o in the definition; an observation point is "absent" when the given valuation V disagrees with its path condition, so none of the memory accesses that belong to the observation point will be executed. For example, consider (`if` $x = 0$ `then` $A[0] = 1$ `else` $A[4] = 1$), $L = \{x\}$ and $\mathsf{V} = \{x \mapsto 0\}$. We have $LDS^o(\{\mathtt{A[4]}\}, \mathsf{V}) = \{\langle \diamond, \mathtt{true}\rangle\}$ since `A[4]` is never executed under V. The reason why we need to account for absence of the observation point is to allow all path conditions to cover the entire input space. Without $\diamond$, LDS^o might have partial or even zero coverage as in the example, which is problematic when we compose observation points and take the intersection of their path conditions, which is the case in the definition of *differential set of an equi-effect program S*:

Definition 7 (Differential Set of an Equi-Effect Program). *For any equi-effect program S and a valuation* V*,*

$$DS(S, \mathsf{V}) = \left\{ a_1 a_2 \cdots a_{|\mathcal{OP}|} \mid (\forall i. \langle a_i, pc_i\rangle \in LDS^o(o_i, \mathsf{V})) \wedge (\bigcap_j pc_j \neq \emptyset) \right\}$$

Recall that each element in AS also tracks its corresponding path condition pc. The differential set of a program is essentially a Cartesian set of the local differential sets of all observation points; the difference is that the last condition ($\bigcap_j pc_j \neq \emptyset$) rules out impossible combinations when composing differential sets for leakage quantification.

Finally, we note that the leakage of a program S can be measured by DS as follows:

Definition 8 (Differential Set Leakage). *Given a set of public inputs L, an equi-effect program S, the side channel leakage induced by the differential set of S is:*

$$\mathcal{L}(S, L) = \log_2(\max_{\mathsf{V}:L \to \mathfrak{R}} |DS(S, \mathsf{V})|)$$

Example. We use the running example in Figure 1 to walk-through the key definitions introduced above. For simplicity we will consider the first iteration of the loop only (note that the program is already in its equi-effect form). Consider any valuation of public inputs V. There are in total three observation points in the first iteration[4], where `in[i]`@line 8 belongs to point 1, `in[i]`@lines 9 and 11 belong to point 2 and `out[j0]`@line 9 and `out[j1]`@line 11 belong to point 3. Moreover, the access sets of the first three memory accesses are {`in[0]`} as they always access `in[0]` in the first iteration regardless of secret inputs. The access sets of `out[j0]`@line 9 and `out[j1]`@line 11 are {`out[0]`} and {`out[3]`} respectively in the first iteration. The differential set at each observation

[4]Note that starting from the branch at line 7, the first loop iteration only has three memory accesses to arrays.

point, according to Definition 7, are shown below, parameterized on valuation V.

$$\begin{aligned} LDS^o(1, \mathtt{V}) &= \{\langle \mathtt{in[0]}, [\![\mathtt{true}]\!]\rangle\} \\ LDS^o(2, \mathtt{V}) &= \{\langle \mathtt{in[0]}, [\![\mathtt{true}]\!]\rangle\} \\ LDS^o(3, \mathtt{V}) &= \{\langle \mathtt{out[0]}, [\![\mathtt{in[0]} < \mathtt{threshold}]\!]\rangle, \\ &\qquad \langle \mathtt{out[3]}, [\![\mathtt{in[0]} \geq \mathtt{threshold}]\!]\rangle\} \end{aligned}$$

Further, we can compute that the differential set of the first iteration of the loop is

$$\{\mathtt{in[0]in[0]out[0]}, \mathtt{in[0]in[0]out[3]}\}$$

Hence, by Definition 8 and the fact that DS remains the same for all public inputs, we have

$$\mathcal{L}(S, \emptyset) = \log_2 |\{\mathtt{in[0]in[0]out[0]}, \mathtt{in[0]in[0]out[3]}\}| = 1$$

which provides a precise side channel leakage of 1 bit. Differential sets can be automatically approximated via a static program analysis, which we explore further in Section 5.

4.3.4 Comparison with Execution Traces. The key difference between differential set and memory access trace is that differential set trades precision for composition, making it more suitable for static program analysis. To illustrate the difference more clearly, consider the following equi-effect program with two consecutive secret-dependent branches where s is a secret input:

$$(\mathtt{if}\ s\ \mathtt{then}\ A[0] = 1\ \mathtt{else}\ \mathtt{hole}); (\mathtt{if}\ \neg s\ \mathtt{then}\ \mathtt{hole}\ \mathtt{else}\ A[0] = 1)$$

This program has no data cache leakage before being transformed into an equi-effect form, as it only produces one trace with A[0]. However, according to Definition 7, its differential set is {A[0] hole, hole A[0]} with any valuation of V. Accordingly, the differential set leakage of the program is 1 bit according to Definition 8. However, we emphasize that the loss of precision in such corner cases enables us to reason about differential set in a compositional way: it is sufficient to compute the *local* differential set of each memory access in isolation, and then combine them to compute the (global) differential set of a whole program. The soundness result in the next subsection assures that differential set serves as a sound approximation of memory access traces. Moreover, the empirical study in Section 6.3 suggests that the over-approximation is precise enough in practice: it reports the *exact* leakage bound on all benchmark programs used by prior work.

4.4 Soundness of Differential Sets

We prove that differential sets are sound, in the sense that the cache side channel leakage (Definition 1) of an *arbitrary* program (which may or may not be in its equi-effect form) is always bounded by the differential set leakage (Definition 8) of *any* of its equi-effect variant. As a consequence, it is safe to use differential set to reason about cache side channels. Moreover, if the differential set leakage of a program's any equi-effect variant is zero, then the program has no cache side channels.

To show soundness, the key observation is that a program in its equi-effect form has the same memory access traces as the original program, except possibly with some holes inserted. So it is possible to build an injection from traces produced by the original program to ones produced by its equi-effect form. Consider an example with two sensitive branches:

$$\mathtt{if}\ s1\ \mathtt{then}\ (\mathtt{A[0]};\mathtt{B[0]})\ \mathtt{else}\ (\mathtt{A[0]};\mathtt{hole});$$
$$\mathtt{if}\ s2\ \mathtt{then}\ (\mathtt{A[0]};\mathtt{hole})\ \mathtt{else}\ (\mathtt{B[0]};\mathtt{A[0]})$$

which is in equi-effect form. The original program (with holes removed) produces three traces T_1 = {A B A, A B B A, A A}, while its equi-effect form produces 4 traces T_2={A B A hole, A B B A, A hole A hole, A hole B A}. Note that with extra holes, a program can only create new traces when it runs (e.g., from A B A to A B A hole and A hole B A); different traces remain different.

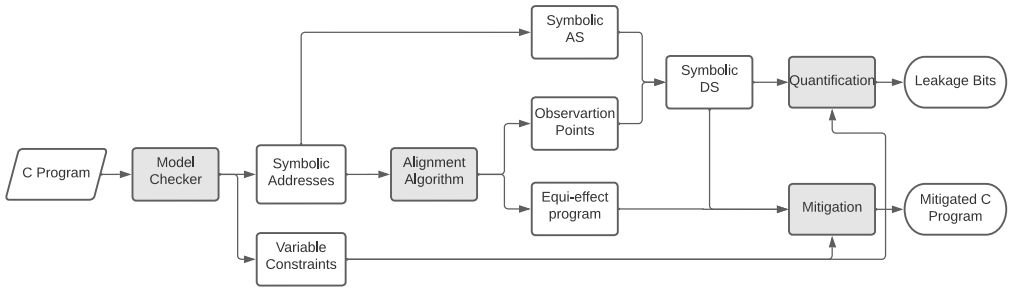

Fig. 4. Overview of DSA

Theorem 1. *Given a program S and any of its equi-effect variant S', public inputs L, we have*

$$CC(S, L) \leq \mathcal{L}(S', L)$$

by expanding definitions, this is equivalent to

$$\log_2 \left(\max_{V:L \to \Re} |\{\tau \downarrow_{addr} | \exists m.\ m \sqsupset \vee \wedge \langle m, S \rangle \hookrightarrow \tau\}| \right) \leq \log_2 \left(\max_{V:L \to \Re} \left| \left\{ a_1 a_2 \cdots a_{|OP|} \mid \langle a_j, pc_j \rangle \in LDS(o_j, \vee) \wedge (\bigcap_j pc_j \neq \emptyset) \right\} \right| \right)$$

The proof is included in the Appendix A.

5 DSA

In this section, we develop a static analysis tool, DSA, to automatically compute differential sets. To showcase the value of differential set, DSA also automatically quantifies and mitigates cache side channels based on the computed differential sets. The workflow of DSA is shown in Figure 4 where the grey boxes represent the major components that we will discuss further in this section, and the white boxes represent data in between of the major components.

5.1 Symbolic Inputs, Addresses and AS

DSA is built on a model checker, CBMC [Clarke et al. 2004], to analyze the semantics (in the form of propositional formulas) of an input C program. During the analysis, all program inputs are treated in a symbolic way. The model checker computes the following information for *each memory access* in the input C program: (1) a symbolic address that consists of a memory object ID variable $\alpha^{\mathtt{obj}}$ and a memory offset variable $\alpha^{\mathtt{off}}$, both in the domain of natural numbers, and (2) constraints on symbolic address variables as well as variables induced by the program, including a symbolic path condition that guards the memory access.

According to the definition of AS (Definition 2), it is natural to generate *symbolic* AS in the following format for each memory access η in the source code:

$$\text{SymAS}(\eta) = ((\alpha_\eta^{\mathtt{obj}}, \alpha_\eta^{\mathtt{off}}), \mathtt{pc}_\eta, cons_\eta)$$

where $\mathtt{pc}_\eta$ is the path condition, and $cons_\eta$ is the rest of constraints on variables. SymAS is then fed into an alignment algorithm to help transforming the input program into an equi-effect form, and subsequently produce a symbolic form of differential sets.

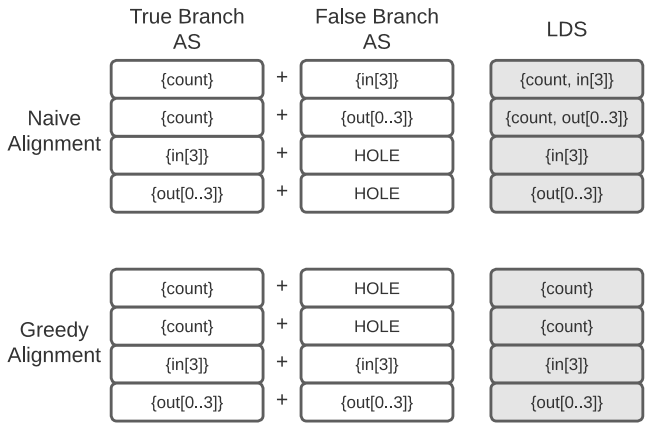

Fig. 5. Naive Alignment vs Greedy Alignment: top half shows the result of native alignment and the lower half shows the result of greedy alignment. Each white box shows AS for one memory access. The grey boxes show LDS after merging ASes according to the alignment. The notation `A[0..1]` is a shortcut for {`A[0]`, `A[1]`}

5.2 Alignment Algorithm

Recall that differential set is only defined on equi-effect programs. While Theorem 1 assures that differential set always provides an upper bound of side channel leakage, the particular equi-effect form chosen affects the tightness of the bound, as well as the efficiency of mitigated code.

For example, consider a naive algorithm that matches memory accesses in the true and false branches one by one and a variant of our running example where the following line is added to the beginning of the true branch (`count` is an added pointer input):

$$(*\texttt{count})++$$

The top of Figure 5 shows the equi-effect form of the *last* iteration of the running example. The ASes are shown in white boxes and local differential sets are shown in grey boxes. Notice that the total size of two local differential sets is larger compared with that of another equi-effect form shown at the bottom of Figure 5; a larger local differential set typically implies less efficient mitigation.[5]

To construct an equi-effect form that leads to efficient mitigation, we first note that only sensitive branches requires alignment. Then for each sensitive branch, DSA uses a *greedy alignment algorithm* to match memory accesses on one branch to "similar" memory accesses on the other branch (e.g., the last two memory accesses in both branches). More precisely, the greedy alignment algorithm is shown in Algorithm 1. This algorithm takes in two sequences of symbolic ASes – one from the memory accesses within the true branch (*left*) and one from the memory accesses within the false branch (*right*). It outputs the input program in one of its equi-effect forms (represented as a list of pairs which represents how memory accesses are aligned across the two branches).

The algorithm first picks one memory access with the largest AS size on the left side (line 8), and then finds the memory access on the other side that shares the most elements with it (line 9). Both of functions `GetIndexOfLargestAS` and `GetIndexOfMostSimilarAS` internally use an SMT solver to enumerate the concrete models of each symbolic AS to identify a pair (l_{pivot}, r_{pivot}) that has the most similar ASes. We will discuss how enumeration is done and how we handle complicated constraints in Section 5.4. The pair then splits both branches in half and the algorithm recursively

[5]Recall that to eliminate cache side channels, we need to access all elements in a local differential set to hide any sensitive memory access.

```
1  Function GreedyAlign(left, right):
       Input: Lists of symbolic AS of two branches, each in the form [l_0, l_1, ..., l_n] where l_i is a symbolic
              AS
       Output: A equi-effect program represented as a list of pairs: [(l_0, r_0), (l_1, r_1), ..., (l_m, r_m)] where
               l_i or r_i is either an added hole or a memory access in the original program
2      if left is empty then
3          return [(hole, o), ∀o ∈ right]
4      end
5      if right is empty then
6          return [(o, hole), ∀o ∈ left]
7      end
8      l_pivot ← GetIndexOfLargestAS(left)
9      r_pivot ← GetIndexOfMostSimilarAS(right, left[l_pivot])
10     before_pivot ← GreedyAlign(left[: l_pivot], right[: r_pivot])
11     after_pivot ← GreedyAlign(left[l_pivot + 1 :], right[r_pivot + 1 :])
12     return before_pivot + [(left[l_pivot], right[r_pivot])] + after_pivot
```

Algorithm 1: Greedy Alignment Algorithm

aligns the remaining chunks (Lines 10 and 11). In case of nested branches, the algorithm aligns the innermost branches first and then progressively align the outer ones. As each aligned branch is in equi-effect form, the algorithm eventually produces a program in its equi-effect form.

Return to the example in Figure 5. The Greedy algorithm first identifies `out[0..3]` on both branches as pivots, then recursively identifies `in[3]` on both branches as pivots and inserts two `holes` at the beginning of the false branch. The produced program in its equi-effect form is illustrated at the bottom of Figure 5. Note that the resulting differential set is smaller compared to the naive algorithm.

Observation points and symbolic local differential sets. From the output of the greedy alignment algorithm, it is straightforward to compute observation points: each pair of memory accesses in the output belongs to the same observation point. By Definition 6, the symbolic local differential set of a memory access η and its corresponding observation point are computed as follows:

$$\text{SymLDS}(\eta) = \{\text{SymAS}(\eta') \mid \exists i.\ \eta, \eta' \in \mathcal{OP}(i)\}$$

5.3 Quantification

We now explain how to quantify cache channel leakage via the computed symbolic local differential set of each memory access. Recall that the differential set leakage (Definition 8) is defined based on differential set (Definition 7), the set of unique and feasible combinations of local differential sets *given any valuation of public inputs*.

We reduce the problem of computing differential set leakage to a (projected) model counting problem. We first create observation point variables $\beta_{\text{i}}^{\texttt{obj}}$ and $\beta_{\text{i}}^{\texttt{off}}$ for each observation point; they stand for the memory object ID and object offset that the observation point might access. Since for any program input, only one memory access $\eta \in \mathcal{OP}(i)$ is executed, we use path condition of each memory access to "select" which one will occur for each observation point, giving us:

$$\mathcal{F}(i) = \bigwedge_{\eta \in \mathcal{OP}(i)} (\texttt{pc}_\eta \Rightarrow \beta_i^{\texttt{obj}} = \alpha_\eta^{\texttt{obj}}) \wedge (\texttt{pc}_\eta \Rightarrow \beta_i^{\texttt{off}} = \alpha_\eta^{\texttt{off}}) \wedge cons_\eta$$

where $\text{SymAS}(\eta) = ((\alpha_\eta^{\texttt{obj}}, \alpha_\eta^{\texttt{off}}), \texttt{pc}_\eta, cons_\eta)$.

We can now compute the information leakage of one observation point by counting how many different addresses the observation point can reach given the constraints.

$$\log_2 \left(\#\{(\beta_i^{\texttt{obj}}, \beta_i^{\texttt{off}}) \mid \mathcal{F}(i)\}\right)$$

To compute the information leakage of the entire program, we concatenate observation point variables into a tuple that represents the trace, and compute the size of the set of traces predicated on the conjunction of constraints of each observation point:

$$\log_2 \left(\#\{(\beta_1^{\texttt{obj}}, \beta_1^{\texttt{off}}, \ldots, \beta_N^{\texttt{obj}}, \beta_N^{\texttt{off}}) \mid \bigwedge_{i=1}^{N} \mathcal{F}(i)\}\right)$$

In order to avoid over-counting the size of each local differential set due to public inputs, DSA runs a *noninterference test* on $\mathcal{F}(i)$: given any valuation of public inputs but different valuations of secret inputs, could there be different $(\beta_i^{\texttt{obj}}, \beta_i^{\texttt{off}})$ under the constraint $\mathcal{F}(i)$? If the noninterference test passes, we know that the size of differential set is 1 given any valuation of public inputs. Hence, the observation point is removed from further analysis. For example, the observation point in `if(p){arr[p]+ = 1}` is noninterferent on secret input; hence, it is removed before counting.

Compositional quantification. We emphasize that differential set by definition enables compositional reasoning, which is important in practice when the constraints on the whole program is complicated. We can partition all observation points into sub-ranges $(r_0 = 0, r_1], (r_1, r_2], ..., (r_{M-1}, r_M = N]$ and compute a sound but less precise leakage bound as follows:

$$\sum_{0 \le i \le M-1} \log_2 \left(\#\{(\beta_{1+r_i}^{\texttt{obj}}, \beta_{1+r_i}^{\texttt{off}}, \ldots, \beta_{r_{i+1}}^{\texttt{obj}}, \beta_{r_{i+1}}^{\texttt{off}}) \mid \bigwedge_{j=1+r_i}^{r_{i+1}} \mathcal{F}(j)\}\right)$$

5.4 Mitigation

To mitigate cache side channels in a program, DSA transforms each memory access (including holes) in the equi-effect program to access all addresses in its local differential sets in sequence. As a consequence, each observation point in the mitigated program will make the same sequence of memory accesses, and hence, the mitigated program becomes cache side-channel free.

Enumeration. Similar to quantification, DSA first runs a noninterference test on each $\mathcal{F}(i)$ to reduce imprecision due to public inputs. That is, whenever $(\beta_i^{\texttt{obj}}, \beta_i^{\texttt{off}})$ is noninterferent on secret inputs under constraint $\mathcal{F}(i)$, the corresponding memory accesses are left in source code as is, since they leak no information.

For the remaining observation points, DSA uses an SMT solver to enumerate concrete models (i.e., values of object ID and offset variables) of the symbolic LDS of each memory access. More precisely, for a memory access η, DSA first enumerates the concrete memory object IDs that the $\alpha_\eta^{\texttt{obj}}$ can refer to, and it then enumerates possible offsets for each concrete memory object. As constraints on both object ID variable and offset variable can be large in practice, the naive enumeration approach can be inefficient. In such cases, we use the following constraint simplification technique to trade precision for performance, while still soundly over-approximating each local differential set.

We observe that not all clauses in the constraint system are equally important for enumerating a local differential set: clauses C_1 that directly constrain $\alpha_\eta^{\texttt{obj}}$ and $\alpha_\eta^{\texttt{off}}$ are the most important. Further, clauses C_2 that directly constrain variables defined in C_1 are also relevant, and so on. Based on the observation, DSA builds a clause graph where nodes represent SMT clauses and edges represent def-use relations on constraint variables. To enumerate concrete models of a local differential set of a memory access η within reasonable amount of time, DSA prunes the constraint clauses that are K-distance away from the $\alpha_\eta^{\texttt{obj}}$ and $\alpha_\eta^{\texttt{off}}$. The variable K is initially set to 10, and if enumeration

times out after 3 seconds, we gradually decrement K and retry enumeration until it succeeds within 3 seconds. Note that the pruning procedure is sound, since it can only over-approximate, but not under-approximate a local differential set.

Linearization. DSA adopts data-flow linearization technique from Constantine [Borrello et al. 2021] to efficiently touch each concrete memory address identified in the enumeration step. The details are included in the Appendix C due to space constraint.

6 IMPLEMENTATION AND EVALUATION

6.1 Implementation

DSA uses the frontend of CBMC [Clarke et al. 2004] to inline functions and unroll loops. It then translates a C program into propositional formulas for further analysis. The propositional formulas encode (1) the symbolic address at each memory accesses and their corresponding path conditions, and (2) transition relation of the entire program. CBMC also runs an alias analysis and encode the information in the formula. DSA implements the alignment algorithm in Section 5.2 and transforms code into its equi-effect form in Python using pyscparser [Bendersky 2022]. For quantification, DSA uses a state-of-art approximate model counter approxMC [Soos et al. 2020] to compute the differential set leakage as discussed in Section 5.3. For mitigation, DSA uses Z3 [De Moura and Bjørner 2008] to generate concrete models of each local differential set as discussed in Section 5.4. We use the AVX512 ISA extension to implement secure memory access, which is the same as Constantine.

6.2 Evaluation Setup

Since DSA targets two distinct challenges, namely quantification and mitigation of cache side-channel, we answer the following questions in the evaluation:

(1) How precise are the leakage bounds produced by DSA, and how do they compare to state-of-art quantification tools?
(2) How efficient are the mitigated programs produced by DSA, and how do they compare to the ones produced by state-of-art mitigation tools?

We use three sets of programs evaluated in prior works. The first two sets of programs are from Blazer [Antonopoulos et al. 2017] and CacheAudit [Doychev and Köpf 2017]; these programs are commonly used to evaluate quantification tools. Blazer's benchmark consists of vulnerable programs from previous literature and snippets from the DARPA STAC challenge [STAC 2017]. Since the benchmark is written in Java, we manually ported them to C. CacheAudit's benchmark consists of variants of the square and multiple routine in RSA implementation. The third set of programs is from Constantine [Borrello et al. 2021], the state-of-the-art side-channel mitigation tool. Those programs were collected from multiple previous works and consist of crypto code and textbook algorithms. We exclude Botan C++ since DSA only supports C code.

We fully unroll the loops that have constant bounds. For programs that contain loops whose conditions depend on inputs, we unroll the loop 100 times, which is also the maximum loop unrolling factor used by CacheAudit. Note that for fairness reason, all tools are run on the *same* unrolled version of source code in the evaluation. Moreover, we found that the approximate model counter used by DSA is slow for a few programs where control-flow and data-flow dependency is complicated. For those cases, we manually apply the compositional quantification approach as discussed in Section 5.3, with each loop iteration as the unit of leakage quantification. Moreover, all public and secret program inputs are symbolized in the analysis.

Table 2. Quantification Results: the case marked with ⋆ uses compositional reasoning.

Program	Leakage (bits)	Leakage Reported by DSA	Leakage Reported by CacheAudit	Time (s)
Blazer/Array Safe	1	1	1	52
Blazer/Array Unsafe	1	1	1	6
Blazer/LoopAndBranch Safe	$\log_2(3)$	$\log_2(3)$	$\log_2(3)$	29
Blazer/LoopAndBranch Unsafe	$\log_2(3)$	$\log_2(3)$	$\log_2(3)$	28
Blazer/Sanity Safe	1	1	1	31
Blazer/Sanity Unsafe	1	1	1	13
Blazer/Straightline Safe	1	1	$\log_2(3)$	7
Blazer/Straightline Unsafe	1	1	$\log_2(3)$	34
Blazer/Unixlogin Safe	$\log_2(3)$	$\log_2(3)$	N/A	12
Blazer/Unixlogin Unsafe	$\log_2(3)$	$\log_2(3)$	N/A	8
Blazer/modPow1 Safe	32	32	N/A	432
Blazer/modPow1 Unsafe	32	32	N/A	364
Blazer/modPow2 Safe	32	32	N/A	463
Blazer/modPow2 Unsafe	32	32	N/A	(⋆) 54
Blazer/passwordEq Safe	16	16	16	4
Blazer/passwordEq Unsafe	$\log_2(17)$	$\log_2(17)$	$\log_2(17)$	4
Blazer/k96 Safe	32	32	N/A	(⋆) 38
Blazer/k96 Unsafe	32	32	N/A	(⋆) 38
Blazer/gpt14 Safe	32	32	N/A	332
Blazer/gpt14 Unsafe	32	32	N/A	(⋆) 9
Blazer/login Safe	16	16	16	4
Blazer/login Unsafe	$\log_2(17)$	$\log_2(17)$	$\log_2(17)$	15
CacheAudit RSA/sqr_and_mult	1	1	1	11
CacheAudit RSA/sqr_and_always_mul	1	1	1	10
CacheAudit RSA/win_mod_exp_libgcrypt1.6.1	$\log_2(9)$	$\log_2(9)$	$\log_2(51)$	3
CacheAudit RSA/win_mod_exp_libgcrypt1.6.3	0	0	0	28
CacheAudit RSA/scatter_gather_openssl_1.0.2f	$\log_2(9)$	$\log_2(9)$	1152	892
CacheAudit RSA/defensive_gather	1	1	1	435

All experiments are performed on a c5.24x machine on AWS with Ubuntu 20.04 featuring Intel Xeon Platinum 8000 series processor with 96 vCPU and 192 GB of memory.

6.3 Quantification Results

We first present the results on quantification benchmarks from Blazer [Antonopoulos et al. 2017] and CacheAudit [Doychev and Köpf 2017]. Since the Blazer benchmark assumes a different adversary model (i.e., leakage via the number of instructions being executed), whereas we assume a cache-based adversary, we adjusted the true leakage of each program due to the difference in attack model. For example, a sensitive branch where two branches have the same number of instructions is deemed free of side channels in their model, but it is insecure in terms of the absence of cache side channels, as even with the same number of instructions, each branch might leave its unique secret-dependent footprints in the cache. The CacheAudit benchmark also assumes a cache-based adversary, which allows us to reuse the benchmark programs and perform a direct comparison with CacheAudit [Doychev and Köpf 2017]. We obtained the public version of CacheAudit and run

it on the same machine as our tool. Note that since the public version supports only a subset of x86 instructions, it is unable to analyze some benchmark programs, marked with N/A.

We summarize the evaluation results in Table 2 where the second column is the actual side channel leakage of each program, obtained from prior works. The exception is those from Blazer which were designed for timing side-channel research, so we manually adjust them. Note that DSA reports accurate cache side channel leakage for all programs within a reasonable amount of time. The program `scatter_gather_openssl_1.0.2f` takes the longest time (15 minutes), which is due to a high loop unrolling factor. In comparison, for the majority of programs that it can analyze, CacheAudit also generates an accurate bound; notably, other than a few cases that it fails to analyze due to unsupported instructions, CacheAudit is accurate on all Blazer benchmark programs. However, CacheAudit reports imprecise results for `Blazer/Straightline_Safe`, `Blazer/Straightline_Unsafe`, `win_mod_exp_libgcrypt1.6.1` and `scatter_gather_openssl_1.0.2f`.

For the former two programs in Blazer benchmarks, DSA is more precise because DSA only counts variations due to the secret inputs, whereas CacheAudit does not distinguish public and secret inputs. Hence, for a code pattern like (`if` *public* `then` (`if` *secret* `then` $A[0]$ `else` $A[4]$) `else` $A[8]$) which are found in those benchmarks, DSA correctly reports a leakage of 1 bit, meaning that varying secret inputs can at most induce two different memory traces, while CacheAudit reports a leakage of $\log_2(3)$ bits, which over-counts the influence of public value. To understand why DSA is more precise on the latter two cases, we present the code segment of `scatter_gather_openssl_1.0.2f`:

```
1  if (0 <= k && k <= 7) {
2       for (i = 0; i < 384; i++){
3           p[i] = buf[k+i*8];
4       }}
```

Here, k is a secret input, and it is used to index into an array in the loop. CacheAudit fails to reason about correlated leakage across loop iterations and produces an imprecise approximation ($3 \times 384 = 1152$ bits). Nevertheless, the predicates associated with each differential set allows DSA to reason about such correlation precisely, and correctly conclude that only 3 bits are leaked by the for loop. In addition, the else branch adds another possible memory trace, making the total leakage $\log_2(9)$ bits. For the same reason, DSA is able to report a precise bound for `win_mod_exp_libgcrypt1.6.1`.

6.4 Mitigation Results

We note that Constantine assumes a slightly different attack model than DSA: it enforces the constant-time principle [Almeida et al. 2016]. To make an apple-to-apple comparison, we use data+instruction mode of DSA and use the same technique as Constantine to remove sensitive branches. Hence, the mitigated code of DSA and Constantine has the same level of security. Since the mitigated code of sensitive branches is the same, we elaborate on a subset of programs that contain no sensitive branches. Both implementations of Constantine and DSA use the AVX512 vector extension and striding width $\lambda = 4$ to capture a strong attacker model, where an attacker can observe accesses to the individual banks within a cache line [Yarom et al. 2017]. For each benchmark program, we compare the runtime overhead of the mitigated programs produced by DSA and Constantine. We run both programs using the same random input file for 2000 times, and collect the number of user-level CPU cycles using the tool *perf*.

We first note that the mitigated code generated by DSA removes cache side channels with *fewer memory accesses* compared to taint-based approaches, such as Constantine. In particular, we found that among 15762 memory accesses that use sensitive array index, for 1056 (6.7%) cases, the memory region that each differential set covers is smaller than the whole array. Note that existing mitigation tools that use taint analysis to identify vulnerably accesses, including Constantine, need to stride

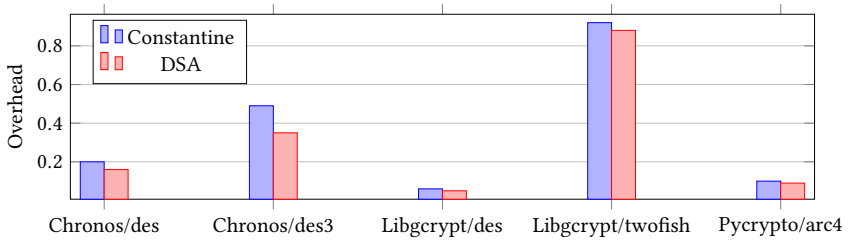

Fig. 6. Runtime overhead for mitigation benchmarks where DSA incur less memory footprint due to the preciseness of differential set

over the entire array for those 1056 accesses. Because of the extra precision offered by differential set, we observe a consistent reduction of mitigation overhead in all five benchmark programs that contain at least one of those 1056 cases (shown in Figure 6).

We use two examples to demonstrate the strength of DSA. The first comes from Chronos/DES3:

```
1  const uint8_t pc1[256] = {...};
2  const uint32_t pc2[1024] = {...};
3  unsigned long a=sec0;
4  a = pc1[a];
5  p = pc2[2 * a + 0]
```

Notice that line 5 contains a sensitive memory access because `a` is secret-dependent. The mitigated code produced by Constantine accesses all elements in array `pc2` due to the lack of differential set, whereas the mitigated code produced by DSA only accesses elements in the differential set, which consists only 1/8 elements of the entire array. The reason is that the constant lookup table `pc1` only contains 128 unique even values from 0 to 254. So the differential set contains exactly 128 elements. In this case, DSA reports 35% overhead, compared to 49% overhead of Constantine.

The second example comes from ARC4 implementation in pycrypto:

```
1  unsigned char state[256];
2  ...
3  int i2 = ( key[i1] + state[i] + i2) % 32;
4  state[i] = state[i2];
```

`i2` contains secret key, which is leaked through the array access at line 4. Because `i2` is computed using a modular, its value can only range from 0 to 31. Therefore the DS guided mitigation only strides over a partial array. In this case, DSA reports 9% overhead, compared to 10% of Constantine.

In other benchmarks, both DSA and Constantine need to stride over the entire array; they exhibit similar overhead for both DSA and Constantine (data included in Appendix D). While overheads of DSA and Constantine differ, we are unable to attribute the difference to the use of differential set. For example, there are a few cases where DSA produces less efficient mitigated code. One reason is that Constantine works on IR code while DSA works on C code. Consequently, DSA sometimes mitigates more sensitive memory accesses compared with Constantine, as some of those accesses are either removed or reduced by a compiler. For example, we find that the LLVM optimizer eliminates consecutive array accesses whose value is unchanged. For example, the following code snippet is taken from ARC4 program in PYCRYPTO benchmark:

```
1      register int t = self->state[x];
2      self->state[x] = self->state[y];
3      self->state[y] = t;
4      register int xorIndex=
5          (self->state[x]+self->state[y]) % 256;
```

Here, `self->state[y]` is loaded at line 5, but the array element is not changed since its last stored at line 3. The compiled IR code preserves the value of `self->state[y]` in a virtual register and skips the load at Line 5, reducing the memory accesses. This pattern is made especially prevalent due to loop unrolling. We leave an IR- or binary-level version of DSA as future work.

7 RELATED WORK

Various automated vulnerability analyses have been developed for cache side channels.

Detection. There are various tools that detect cache side-channels. CacheD [Wang et al. 2017] performs symbolic execution on one concrete execution path to check whether each data access on the path is secret-dependent or not. However, it cannot detect sensitive control-flow paths. CaSym [Brotzman et al. 2019] and SpecSafe [Brotzman et al. 2021] use cache-aware symbolic execution that keeps track of abstract cache state to cover multiple execution paths in the program, and hence soundly detects cache side channels. CacheS [Wang et al. 2019] uses abstract interpretation with a novel abstract domain to soundly detect both sensitive control-flow paths and sensitive memory accesses separately. DATA [Weiser et al. 2018] and Stacco [Xiao et al. 2017] follow a dynamic approach that runs a program multiple times and uses statistical methods to detect leakage based on the produced memory traces. However, the dynamic approach might miss cache side channels. Compared to these works, DSA can soundly detect, quantify and mitigate cache side channels, thanks to the novel abstraction of differential set.

Quantification. CacheAudit [Doychev et al. 2013] develops a novel abstract interpretation to reason about cache side channels. Notably, the abstract domain supports three models: timing-based, access-based, and trace-based adversaries, while DSA assumes trace-based adversaries. Its successor [Doychev and Köpf 2017] further relaxes the cache model to be more abstract and flexible. At a high-level, both tools use abstract interpretation to over-approximate the number of abstract memory traces that a program can produce. Due to the imprecise nature of abstract interpretation as well as the fact that it cannot model correlation of leakage at multiple program points, these tools sometimes report imprecise leakage bounds, as confirmed on both our running example and the benchmark programs evaluated in Section 6.3. Abacus [Bao et al. 2021] uses symbolic execution on one concrete execution trace to obtain SMT formulas for each branch condition and data access address along the trace. Then, it uses Monte Carlo sampling to estimate the leakage in bits for each branch condition and data access address. However, Abacus might underestimate side channel leakage from unexplored control-flow paths. In contrast, DSA uses differential set to soundly compute what addresses might be accessed in all control-flow paths.

Also related to our work are prior analysis techniques that target timing channels, which assume a related but different threat model (Section 2.2). Various tools [Antonopoulos et al. 2017; Chen et al. 2017; Noller and Tizpaz-Niari 2021; Pasareanu et al. 2016; Phan et al. 2017] address the problem of quantifying timing side-channel leakage, i.e., to measure how much information is leaked via a program's execution time. However, most of them do *not* model microarchitectural features, such as CPU cache. Instead, they simply assume program execution time is accurately described by the number of executed instructions (or, steps taken in an operational semantics). For example, Pasareanu et al. [2016] use symbolic execution and Max-SMT solver to automatically derive public inputs that leads to most leakage, and quantify the leakage for attackers that use these inputs. In comparison, dynamic tool QFuzz [Noller and Tizpaz-Niari 2021] uses program running time and hence, does model microarchitectural features. It builds a fuzzing driver that partitions the input space with their corresponding execution time, and then uses the number of partitions to compute timing side-channel leakage. However, the dynamic tool is unsound. Moreover, these quantification techniques are not directly applicable to cache side-channel, since execution time

is often modeled as one number, whereas DSA needs to examine traces of memory access traces, which is less tractable without the use of differential set.

Mitigation. Side channel detection and mitigation tools are typically built on the constant-time programming paradigm: they detect sensitive control-flow paths and memory accesses and rewrite them separately to remove side channels. For sensitive control-flow paths, Molnar et al. [2005] introduce the program counter security model and propose a program transformation strategy using conditional assignment, and Coppens et al. [2009] implement a compiler backend that removes timing channels via predicated execution on X86 processors. Recent mitigation tools [Borrello et al. 2021; Wu et al. 2012] implement a secure select function that executes both branches while preserving the original program semantics. For sensitive memory accesses, SC-Eliminator [Wu et al. 2012] pre-loads all elements in lookup tables with sensitive memory accesses. However, pre-loading offers weak security assurance: an attacker might interrupt between the pre-loading and sensitive memory access. Raccoon [Rane et al. 2015] uses transactional memory to carry out operations in decoy paths, and uses Path ORAM [Stefanov et al. 2018] to obliviously access array elements. However, ORAM can introduce substantial run-time overhead [Borrello et al. 2021]. FaCT [Cauligi et al. 2019] is a domain-specific language that allows programmers to write code in a subset of C, which is then automatically compiled to constant-time binary. However, FaCT also puts restrictions on its source code, such as programs that index memory based on secret. Constantine [Borrello et al. 2021] implements a linearization technique that transforms each sensitive memory access with a subroutine that efficiently sweeps through memory regions that can be reached by the memory access. Linearization (the same approach adopted by DSA) offers strong security assurance, but due to the lack of differential set information, Constantine needs to sweep through a whole region reachable from problematic pointers. In contrast, DSA can reason at granularity of elements in differential set, allowing the mitigated programs to introduce fewer extra memory accesses, as shown in Section 6.4.

8 CONCLUSION AND FUTURE WORK

We introduce the concept of *differential set* for modeling and mitigating cache-based side channels. Differential set provides a sound abstraction for describing the memory access patterns of an entire program. Based on this new concept, we have developed an automated analysis tool called DSA, *Differential Set Analyzer*. DSA can both quantify and mitigate cache side channels. The evaluation shows that it reports similar or tighter leakage bounds and more efficient mitigation code, compared to the state of the art tools.

In this paper, we use a technique similar to bounded model checking to reason about differential sets. It might be useful to explore other techniques, such as abstract interpretation, to improve DSA's scalability. Moreover, we adopt the linearization approach of prior work to hide a real address among all possible addresses in a local differential set. However, such software-level mitigation can be quite expensive. We find that the abstraction of differential set is simple enough that the mitigation task can be offloaded to hardware. For example, before executing a vulnerable program, we can provide local differential sets to memory system as a blueprint for mitigation. We can modify the memory system such that it efficiently obfuscates the memory access according to differential set. We plan to investigate such software-hardware cooperative solutions in the future.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their constructive feedback that was very helpful toward improving this work. This research was supported in part by the NSF grants CNS-1956032, CNS-1942851, CNS-2207197, CNS-1652790, and CNS-1801534.

DATA-AVAILABILITY STATEMENT

The DSA implementation and instruction to reproduce the results are publicly available[Ma et al. 2023].

A PROOF OF THEOREM 1

Proof. We prove that $CC(S,L) \leq CC(S',L) = \mathcal{L}(S',L)$. We first prove $CC(S',L) = \mathcal{L}(S',L)$ by showing that for any valuation V, the following condition holds:

$$|\{\tau \downarrow_{addr} | \ \exists m.\ m \sqsupset \mathsf{V} \wedge \langle m, S'\rangle \hookrightarrow \tau\}| = \\ |\{a_1 a_2 \cdots a_{|OP|} \mid \langle a_j, pc_j\rangle \in LDS(o_j, \mathsf{V}) \wedge (\bigcap_j pc_j \neq \emptyset)\}|$$

Consider any two execution traces $\langle m, S'\rangle \hookrightarrow \tau$ and $\langle m', S'\rangle \hookrightarrow \tau'$ where $m \sqsupset \mathsf{V}$ and $m' \sqsupset \mathsf{V}$. We have $|\tau| = |\tau'|$ by Corollary 1. Moreover, for each pair of events $\tau[i] = \langle addr, \eta\rangle$ and $\tau'[i] = \langle addr', \eta'\rangle$, η and η' must belong to some observation point $o \in \mathcal{OP}$, as both sensitive and non-sensitive branches have a constant number of memory accesses in equi-effect form. In other words, the set of observation points that participate in all executions with valuation V are the same regardless of the secret input.

To prove $CC(S',L) = \mathcal{L}(S',L)$, we show that there is a bijection between the set of actual memory traces and the set derived from composing observation points. We build the bijection in the following way. For any $\langle m, S'\rangle \hookrightarrow \tau$ where $\tau = \langle a_1, \eta_1\rangle\langle a_2, \eta_2\rangle \cdots \langle a_N, \eta_N\rangle$. If $\eta_i \in \mathcal{OP}(j)$ for some j, we construct $a'_j = a_i$. For the remaining observation points, we set the corresponding address to be $\diamond$. It is easy to check that the constructed trace of a'_j exists on the RHS since $(\bigcap_j mset_j \neq \emptyset)$ as at least m is an element of the intersection, and moreover, it is unique. On the other hand, consider any two traces $\langle m', S'\rangle \hookrightarrow \tau$ and $\langle m', S'\rangle \hookrightarrow \tau'$ such that $\tau \neq \tau'$. Since $\tau \neq \tau'$, there exist at least one observation point, which we call o, at which point the two traces differ. Hence, the mapping constructed above maps τ and τ' to different elements on the RHS. Similarly, for every composition on the RHS, we can find an initial memory state that induces a unique η on the LHS. Thus we build a bijection. Hence, $CC(S',L) = \mathcal{L}(S',L)$.

Next, we prove $CC(S,L) \leq CC(S',L)$ by showing that for any valuation V, the following condition holds:

$$|\{\tau \downarrow_{addr} | \ \exists m.\ m \sqsupset \mathsf{V} \wedge \langle m, S\rangle \hookrightarrow \tau\}| \leq |\{\tau \downarrow_{addr} | \ \exists m.\ m \sqsupset \mathsf{V} \wedge \langle m, S'\rangle \hookrightarrow \tau\}|$$

By fixing V, we prove that there is an injection from the set of memory traces of S (named T) to the set of possible memory trace of S' (named T'). We first choose arbitrary $\tau_0, \tau_1 \in T$ produced by initial memories m_0, m_1 such that $\tau_0 \neq \tau_1$. The same memories m_0, m_1, when ran on S', produce traces in T' which we call τ'_0 and τ'_1 respectively. We argue that $\tau'_0 \neq \tau'_1$ by contradiction. Notice that we get τ_0 (resp. τ_1) if we remove all holes in τ'_0 (resp. τ'_1). Hence, if $\tau'_0 = \tau'_1$ that means $\tau_0 = \tau_1$, contradiction. Since the construction holds for all traces in T, the constructed relation is an injection. Therefore, we have $|T| \leq |T'|$ and hence, $CC(S,L) \leq CC(S',L)$.

Putting all pieces together, we showed that $CC(S,L) \leq \mathcal{L}(DS,L)$.

□

B FORMALIZING TRACE SEMANTICS

Language Semantics. Recall that to model program execution with side channels, we define a configuration as a pair $\langle m, S\rangle$ that consists of a memory m (i.e., a mapping from variables/memory addresses, including array elements, to their values) as well as S, the remaining program to be executed. Recall that a program leaks information through a cache side channel when it leaves

$$\frac{eval(E, m) = v}{\langle m, X := E\rangle \xrightarrow{loc(E)} \langle [X/v]m, \texttt{skip}\rangle} \text{(Asgn)} \qquad \frac{m(X) = x \wedge eval(E, m) = v}{\langle m, *X := E\rangle \xrightarrow{loc(E)\cdot x} \langle [x/v]m, \texttt{skip}\rangle} \text{(Asgn-Ptr)}$$

$$\frac{m(X) = x \wedge eval(E, m) = v}{\langle m, A[X] := E\rangle \xrightarrow{loc(E)\cdot(A+\texttt{off}_A\times x)} \langle [A + \texttt{off}_A \times x/v]m, \texttt{skip}\rangle} \text{(Asgn-Arr)}$$

$$\frac{\langle m, S_1\rangle \xrightarrow{e} \langle m', S_1'\rangle}{\langle m, S_1; S_2\rangle \xrightarrow{e} \langle m', S_1'; S_2\rangle} \text{(Seq1)} \qquad \frac{}{\langle m, \texttt{skip}; S_2\rangle \longrightarrow \langle m, S_2\rangle} \text{(Seq2)}$$

$$\frac{\eta \text{ is } c_1\text{'s id} \qquad eval(B, m) = \texttt{true}}{\langle m, \texttt{if } B \texttt{ then } c_1 \texttt{ else } c_2\rangle \xrightarrow{loc(B)\cdot\eta} \langle m, c_1\rangle} \text{(If-True)}$$

$$\frac{\eta \text{ is } c_2\text{'s id} \qquad eval(B, m) = \texttt{false}}{\langle m, \texttt{if } B \texttt{ then } c_1 \texttt{ else } c_2\rangle \xrightarrow{loc(B)\cdot\eta} \langle m, c_2\rangle} \text{(If-False)}$$

Fig. 7. Language semantics with emitted events e.

different memory footprints in cache when running on different secret values. Hence, each small-step evaluation rule has the form of $\langle m, S\rangle \xrightarrow{e} \langle m', S'\rangle$ where event e tracks the memory addresses being accessed by the evaluation step.

We assume that each memory access in the source code has a unique id, denoted by η. The semantics for the source language syntax of Figure 3 is shown in Figure 7, where most small-step semantics rules are standard. For each expression E, the auxiliary function $loc(E)$ collects all memory addresses used in E. Depending on the LHS of an assignment, the evaluation step either issues no memory address (when the LHS is a local variable stored in a register), issues the value stored in X (when the LHS is a pointer deference), or issues the address associated with an array element (when LHS is an array element). For an if-statement, the tag of either the true or false branch is also included in events, depending on the truth value of the branch condition B. Note that when instruction cache is irrelevant, we can also omit branch tags in the semantics.

C LINEARIZATION

DSA adopts data-flow linearization technique from Constantine [Borrello et al. 2021] to efficiently touch each concrete memory address identified in the enumeration step. In particular, DSA inherits three types of subroutines for data-flow linearization: *simple*, *gather*, and *bulk*. Under each type, one subroutine performs oblivious load and the other subroutine performs oblivious store [Borrello et al. 2021].

- A *simple* subroutine uses a for-loop to iterate through a set of addresses. The pseudocode for memory load is shown in Figure 8, which takes in the pointer (`ptr`) that the original memory access operates on, and an array of addresses (`DS`) representing differential set. It loads the content of each address in the differential set at line 4 and uses a conditional expression at line 5 to keep the true value loaded from the real address `ptr`. For memory store, each address is loaded into a local variable. The variable is conditionally updated depending on whether

```
1  int secure_load(ptr, DS){
2      int res = 0;
3      for (addr in DS) {
4          int val = load(addr);
5          res = (addr == ptr) ? val : res;
6      }
7  }
```

Fig. 8. *Simple* Secure Load Subroutine

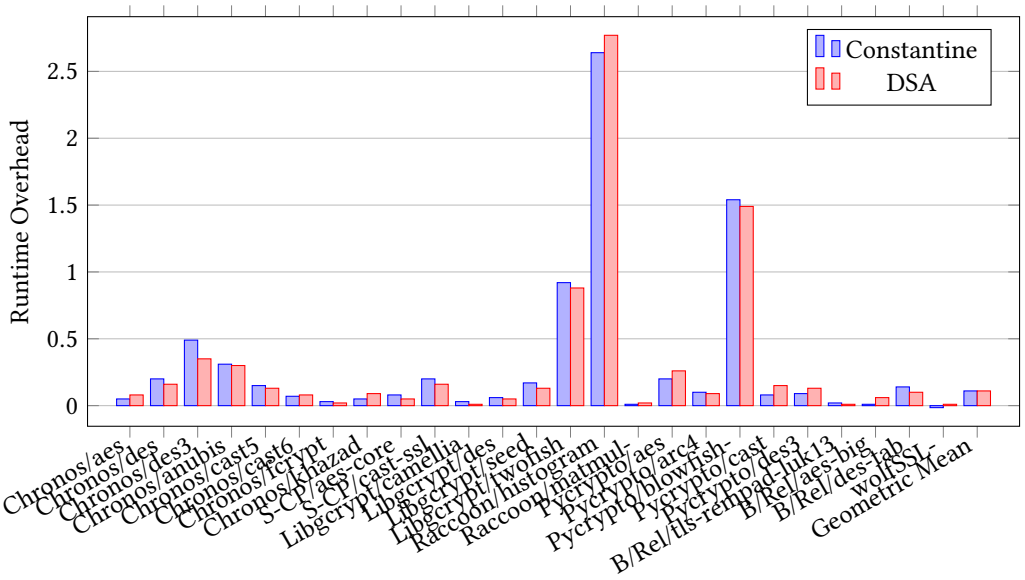

Fig. 9. Mitigation Results: Benchmarks marked with "-" are modified to have smaller loop bounds. Due to the reduction in loop bound, wolfSSL used in evaluation do not contain sensitive branches or sensitive memory accesses.

the memory address under the iteration matches the real address, and the value (which may or may not have changed) is then stored back.

- A *gather* subroutine is similar to a simple subroutine, except that it utilizes gather/scatter instruction from SIMD extension to touch multiple memory locations in parallel and uses a mask operation to wipe out bits in the vector except the bits that are from the real address
- A *bulk* subroutine is suitable for loading/storing sequential data to and from the memory; it similarly utilizes SIMD instructions to load/store a trunk of continuous memory in a oblivious way.

Following the strategy of Constantine, DSA picks a load/store subroutine from the three types of subroutines based on the sparsity and the size of the differential set to be mitigated. If the average distance between elements in the differential set is smaller or equal to 16 bytes, DSA uses bulk subroutines since they can efficiently operate on a continuous region of memory. If there are fewer than 8 elements in the differential set, DSA uses simple subroutines to avoid the overhead of vector instructions. Otherwise, DSA uses gather subroutines since it can simultaneously load memory from a set of addresses, so it can deal with sparse differential set efficiently.

D COMPLETE MITIGATION RESULTS

Figure 9 presents mitigation results on benchmarks that do not contain sensitive branches. In most cases, Constantine and DSA performs similarly, the noticeable differences are discussed in Section 6.4. For some benchmarks, we reduce the unrolling factor due to scalability issue. These are marked at the end of their names with -. We note that wolfSSL does not contain any sensitive branches or sensntive memory accesses due to reduced loop bound.

REFERENCES

Onur Aciicmez and Jean-Pierre Seifert. 2007. Cheap Hardware Parallelism Implies Cheap Security. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*. 80–91. https://doi.org/10.1109/FDTC.2007.16

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*. 53–70.

Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition Instead of Self-Composition for Proving the Absence of Timing Channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 362–375. https://doi.org/10.1145/3062341.3062378

Qinkun Bao, Zihao Wang, Xiaoting Li, James R. Larus, and Dinghao Wu. 2021. Abacus: A Tool for Precise Side-Channel Analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 238–239. https://doi.org/10.1109/ICSE-Companion52605.2021.00110

Eli Bendersky. 2022. pysparser. https://github.com/eliben/pycparser. https://github.com/eliben/pycparser

Joseph Bonneau and Ilya Mironov. 2006. Cache-Collision Timing Attacks Against AES. In *Cryptographic Hardware and Embedded Systems - CHES 2006*, Louis Goubin and Mitsuru Matsui (Eds.). Lecture Notes in Computer Science, Vol. 4249. Springer Berlin Heidelberg, 201–215. https://doi.org/10.1007/11894063_16

Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. 2021. Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 715–733. https://doi.org/10.1145/3460120.3484583

Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings of the 11th USENIX Conference on Offensive Technologies (WOOT'17)*. 11–11.

Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. 2019. CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation. In *2019 IEEE Symposium on Security and Privacy (SP)*. 505–521. https://doi.org/10.1109/SP.2019.00022

Robert Brotzman, Danfeng Zhang, Mahmut Taylan Kandemir, and Gang Tan. 2021. SpecSafe: Detecting Cache Side Channels in a Speculative World. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 129 (oct 2021), 28 pages. https://doi.org/10.1145/3485506

Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 913–926. https://doi.org/10.1145/3385412.3385970

Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: A DSL for Timing-Sensitive Computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 174–189. https://doi.org/10.1145/3314221.3314605

Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities Using Quantitative Cartesian Hoare Logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 875–890. https://doi.org/10.1145/3133956.3134058

Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004) (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *2009 30th IEEE Symposium on Security and Privacy*. 45–60. https://doi.org/10.1109/SP.2009.19

Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2020. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1021–1038. https://doi.org/10.

1109/SP40000.2020.00074

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*. https://doi.org/10.1007/978-3-540-78800-3_24

Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 431–446. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev

Goran Doychev and Boris Köpf. 2017. Rigorous Analysis of Software Countermeasures against Cache Attacks. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 406–421. https://doi.org/10.1145/3062341.3062388

Edward W. Felten and Michael A. Schneider. 2000. Timing Attacks on Web Privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security* (Athens, Greece) *(CCS '00)*. Association for Computing Machinery, New York, NY, USA, 25–32. https://doi.org/10.1145/352600.352606

Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security* (Belgrade, Serbia) *(EuroSec'17)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3065913.3065915

David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *2011 IEEE Symposium on Security and Privacy*. 490–505. https://doi.org/10.1109/SP.2011.22

Yaoqi Jia, Xinshu Dong, Zhenkai Liang, and Prateek Saxena. 2015. I Know Where You've Been: Geo-Inference Attacks via the Browser Cache. *IEEE Internet Computing* 19, 1 (2015), 44–53. https://doi.org/10.1109/MIC.2014.103

Robert Kotcher, Yutong Pei, Pranjal Jumde, and Collin Jackson. 2013. Cross-Origin Pixel Stealing: Timing Attacks Using CSS Filters. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 1055–1062. https://doi.org/10.1145/2508859.2516712

Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*. 605–622. https://doi.org/10.1109/SP.2015.43

Gavin Lowe. 2002. Quantifying Information Flow. In *15th IEEE Computer Security Foundations Workshop (CSFW-15 2002), 24-26 June 2002, Cape Breton, Nova Scotia, Canada*. IEEE Computer Society, 18–31. https://doi.org/10.1109/CSFW.2002.1021804

Cong Ma, Dinghao Wu, Gang Tan, Mahmut Taylan Kandemir, and Danfeng Zhang. 2023. *Quantifying and Mitigating Cache Side Channel Leakage with Differential Set*. https://doi.org/10.5281/zenodo.8418984

David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3935)*, Dongho Won and Seungjoo Kim (Eds.). Springer, 156–168. https://doi.org/10.1007/11734727_14

James Newsome, Stephen McCamant, and Dawn Song. 2009. Measuring Channel Capacity to Distinguish Undue Influence. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security* (Dublin, Ireland) *(PLAS '09)*. Association for Computing Machinery, New York, NY, USA, 73–85. https://doi.org/10.1145/1554339.1554349

Yannic Noller and Saeid Tizpaz-Niari. 2021. QFuzz: Quantitative Fuzzing for Side Channels. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 257–269. https://doi.org/10.1145/3460319.3464817

Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. 3860 (2006), 1–20. https://doi.org/10.1007/11605805_1

Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 387–400. https://doi.org/10.1109/CSF.2016.34

Colin Percival. 2005. Cache missing for fun and profit. In *BSDCan*.

Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of Adaptive Side-Channel Attacks. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. 328–342. https://doi.org/10.1109/CSF.2017.8

Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing digital side-channels through obfuscated execution. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 431–446.

Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '09)*. Association for Computing Machinery, New York, NY, USA, 199–212. https://doi.org/10.1145/1653662.1653687

Isabell Schmitt and Sebastian Schinzel. 2012. WAFFle: Fingerprinting Filter Rules of Web Application Firewalls. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*. 34–40.

Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 3–24. https://doi.org/10.1007/978-3-319-60876-1_1

Geoffrey Smith. 2009. On the foundations of quantitative information flow. *Foundations of Software Science and Computational Structures* 5504 (2009), 288–302. https://doi.org/10.1007/978-3-642-00596-1_21

Mate Soos, Stephan Gocht, and Kuldeep S. Meel. 2020. Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I* (Los Angeles, CA, USA). Springer-Verlag, Berlin, Heidelberg, 463–484. https://doi.org/10.1007/978-3-030-53288-8_22

STAC 2017. DARPA space/time analysis for cybersecurity (STAC) program. http://www.darpa.mil/program/ space-time-analysis-for-cybersecurity. http://www.darpa.mil/program/space-time-analysis-for-cybersecurity

Emil Stefanov, Marten Van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: An Extremely Simple Oblivious RAM Protocol. *J. ACM* 65, 4, Article 18 (apr 2018), 26 pages. https://doi.org/10.1145/3177872

Paul Stone. 2013. Pixel Perfect Timing Attacks with HTML5. https://www.contextis.com/media/downloads/Pixel_Perfect_Timing_Attacks_with_HTML5_Whitepaper.pdf.

Eran Tromer, DagArne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology* 23, 1 (2010), 37–71. https://doi.org/10.1007/s00145-009-9049-y

Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution* (Shanghai, China) *(SysTEX'17)*. Association for Computing Machinery, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/3152701.3152706

Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. 2015. The Clock is Still Ticking: Timing Attacks in the Modern Web. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1382–1393. https://doi.org/10.1145/2810103.2813632

Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. 2019. Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation. In *28th USENIX Security Symposium (USENIX Security 19)*. 657–674.

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

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

Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. 159–173.

Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. 2017. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 859–874. https://doi.org/10.1145/3133956.3134016

Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. 2011. An Exploration of L2 Cache Covert Channels in Virtualized Environments. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop* (Chicago, Illinois, USA) *(CCSW '11)*. Association for Computing Machinery, New York, NY, USA, 29–40. https://doi.org/10.1145/2046660.2046670

Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (San Diego, CA) *(SEC'14)*. USENIX Association, USA, 719–732.

Y. Yarom, D. Genkin, and N. Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *J Cryptogr Eng* 7 (2017), 99–112. https://doi.org/10.1007/s13389-017-0152-y

Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. Association for Computing Machinery, New York, NY, USA, 305–316. https://doi.org/10.1145/2382196.2382230

Received 2023-04-14; accepted 2023-08-27