Hardware Support for Constant-Time Programming

Yuanqing Miao
The Pennsylvania State University
University Park, PA, USA
ypm5112@psu.edu

Mahmut Taylan Kandemir
The Pennsylvania State University
University Park, PA, USA
mtk2@psu.edu

Danfeng Zhang
The Pennsylvania State University
University Park, PA, USA
Duke University
Durham, NC, USA
dz132@duke.edu

Yingtian Zhang
The Pennsylvania State University
University Park, PA, USA
yjz5396@psu.edu

Gang Tan
The Pennsylvania State University
University Park, PA, USA
gtan@psu.edu

Dinghao Wu
The Pennsylvania State University
University Park, PA, USA
dinghao@psu.edu

ABSTRACT

Side-channel attacks are one of the rising security concerns in modern computing platforms. Observing this, researchers have proposed both hardware-based and software-based strategies to mitigate side-channel attacks, targeting not only on-chip caches but also other hardware components like memory controllers and on-chip networks. While hardware-based solutions to side-channel attacks are usually costly to implement as they require modifications to the underlying hardware, software-based solutions are more practical as they can work on unmodified hardware. One of the recent software-based solutions is *constant-time programming*, which tries to transform an input program to be protected against side-channel attacks such that an operation working on a data element/block to be protected would execute in an amount of time that is *independent* of the input. Unfortunately, while quite effective from a security angle, constant-time programming can lead to severe performance penalties.

Motivated by this observation, in this paper, we explore novel hardware support to make constant-time programming much more efficient than its current implementations. Specifically, we present a new hardware component that can greatly improve the performance of constant-time programs with large memory footprints. The key idea in our approach is to add a *small structure* into the architecture and two accompanying *instructions*, which collectively *expose* the existence/dirtiness information of multiple cache lines to the application program, so that the latter can perform more efficient side-channel mitigation. Our experimental evaluation using three benchmark programs with *secret data* clearly show the effectiveness of the proposed approach over a state-of-the-art implementation of constant-time programming. Specifically, in the three benchmark programs tested, our approach leads to about 7x reduction in performance overheads over the state-of-the-art approach.

CCS CONCEPTS

• **Security and privacy → Hardware security implementation**.

KEYWORDS

Side channel leakage, Constant time programming, Cache

ACM Reference Format:
Yuanqing Miao, Mahmut Taylan Kandemir, Danfeng Zhang, Yingtian Zhang, Gang Tan, and Dinghao Wu. 2023. Hardware Support for Constant-Time Programming. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3613424.3623796

1 INTRODUCTION

For a long time, computer architectures have been designed with no provisions for preventing side-channel attacks. As a result, many computer systems today have "vulnerabilities" that can be taken advantage of by adversaries. Exploiting simple properties, such as timing, heat or electromagnetism, attackers can find ways to steal secrets from victim programs. Among those properties, "timing" is probably the most popular one for the reason that attackers do not need any physical access to a victim computer. Additionally, an application's running time normally varies across different execution environments with different cache hierarchies/capacities and shared memory controllers, and attackers might be able to infer the current state of a program by making use of the timing information. As a result, there has been a plethora of recent research [2–5, 12, 15, 16, 24, 27, 28, 30, 31, 41, 51] focusing on both timing attacks and their potential consequences.

Along with the discovery of various side channel attacks, researchers have also investigated various ideas of mitigation, either from hardware side [11, 13, 29, 32, 36, 44, 48, 53] or software side [10, 21, 25, 34, 37, 46, 47]. One of the ideas for hardware-based mitigation is "resource partitioning" [18, 22, 29, 35, 36, 42, 43, 45]. It is to be noted that the root cause of the timing side channels is "resource sharing". With shared hardware components, attackers are able to observe the "footprints" left by the execution of victim programs to gain information if those footprints are "secret-dependent". One particular mitigation strategy that prevents attackers from learning application footprints is to partition hardware components/resources spatially or temporarily, as discussed by [18, 22, 35, 36] and [42, 43, 45], respectively. However, partitioning contradicts the goal of one of the mechanisms through which hardware utilization can be improved, namely, resource sharing. Simultaneous multithreading (SMT), shared last-level caches (LLCs), and shared memory controllers are all built upon the general idea of resource sharing and are essential for improving latency and/or throughput of modern hardware. While in some scenarios it might be acceptable to significantly sacrifice performance to guarantee security, in most cases severe performance degradation is *not* acceptable.

Another hardware-centric idea for improving security is to employ "reshaping" [53] [11]. The idea behind this approach is to reshape a program's memory accesses into "predefined patterns" that are *independent* of the secret. Note however that this approach requires extensive profiling and may not, at the end, fully guarantee security. Also, a reshaping strategy typically cannot deal with early termination attacks [14, 17] and is *not* suitable for components with very heavy data traffic like on-chip caches. Another hardware-based mitigation strategy is "randomization" [23, 32]. However, randomization typically does not provide full security guarantees [33, 38].

In comparison, a software-based mitigation approach transforms application code, either manually or through a compiler, so that the transformed code does *not* leave secret-dependent footprints in hardware. Compared to hardware-based mitigation, software-based mitigation is more flexible, and in particular, it is applicable to computer systems that have been developed with no built-in hardware mitigation strategy. Also, different programs usually have different levels of security requirements. Hardware-based mitigation might be able to meet those requirements precisely [13, 43]; however, it does not scale well to a system with hundreds of security domains. Software mitigation, thanks to its flexibility, allows programs with and without security requirements to run on the same machine. The cons are that a not-so-clear (or undefined) "security contract" between software and hardware can make software-based mitigation less effective than its potential.

Another problem with software-based mitigation is the potential "performance loss". It is well acknowledged that efforts for security almost always hurt performance. Past research [9, 21, 34] has designed and implemented various techniques and libraries for software-based mitigation, with little loss in performance. However, we show in this paper that, their performance appeal gets diminished when the underlying **Dataflow Linearization Set** is large. Dataflow linearization set is the set of "all" possible addresses for a memory access. It is a set meant for dataflow linearization. Mitigated programs need to access each and every element in this set to make their memory footprints entirely "secret-independent". In general, a dataflow linearization set with size greater than 1 leads to a side channel. Generally speaking, while cryptography libraries have very small dataflow linearization sets for their secret-relevant memory accesses, this is not always the case with other types of programs. Some existing proposals might help with this problem. For example, "scratchpad memories" [21] are programmable "cache-like" structures. By loading protected programs into a scratchpad at the beginning, one could prevent attackers from learning the secrets through the observation of the footprints in the caches or memory controllers. It is to be emphasized however that, it usually takes a large memory space to put a whole dataflow linearization set in. Researchers have also explored the idea of preloading the programs to be protected into a cache [47]; however, under this strategy, an attacker can still evict lines from the cache.

Motivated by these observations, in this work, we focus on the problem of "large dataflow linearization sets" in mitigating "cache side-channels". The key aspect of our solution is to provide a given application program with information of whether the required cache lines are "valid" or "dirty" in the cache. With a new structure added to the hardware, such cache information can be *exposed* to the application program and, in this way, the performance of the mitigated application code can be significantly improved. Furthermore, compared to a scratchpad-based solution, we require far less memory area, and compared to a preloading-based strategy, we provide strict security guarantees.

The specific **contributions** this paper makes can be summarized as follows:

- We expose the "large dataflow linearization set" problem in software mitigation.
- We design a small hardware structure that *records* the "existence" and "dirtiness" information for the cache lines and add two new instructions into the mico-operation set of X86-64 ISA to use the proposed structure. With this new hardware structure in place, the cache state gets exposed so as to help the application program to reduce the performance degradation originating from side channel mitigation.
- We modify a state-of-art software mitigation tool with new load and store algorithms which make use of our new instructions. As a result, application programs can *automatically* get transformed into their secure versions with minimal impact on performance.
- We test the effectiveness of our proposed solution using benchmark programs provided in a state-of-art software mitigation paper [9]. Our experimental results clearly show that the performance of the programs with large dataflow linearization sets can be improved by 7x by our work, compared to a state-of-the-art side-channel mitigation scheme.

2 BACKGROUND

In this section, we provide background on side channels in caches and other hardware components, resource partitioning, constant-time programming, and the threat model adopted in this work.

2.1 Cache Side Channels

Limited cache capacities and access time variances make side channel attacks on caches possible. Three popular cache side channel attack models include FLUSH+RELOAD, EVICT+TIME and PRIME+PROBE, which are briefly explained below.

Figure 1 is an example of how to use Prime+Probe to launch an attack by Algorithm 1. In the victim program, there needs to be a "knob", which is either a branch on secret data or a secret-dependent memory access. The attacker primes first, then waits for some time, and finally probes.

2.2 Non-Cache Side Channels

Non-cache hardware components also have vulnerabilities that can pave the way for various side channel attacks. Contentions may happen in such components due to inter-process/inter-application

Algorithm 1: Attacker: Prime and Probe

```
while (1) do
    for (each cache set) do                    // Prime Phase
        start = time();
        access all cache ways
        end = time();
        access_time = end - start;
    wait for victim access
    for (each cache set) do                    // Probe Phase
        start = time();
        access all cache ways
        end = time();
        access_time = end - start;
```

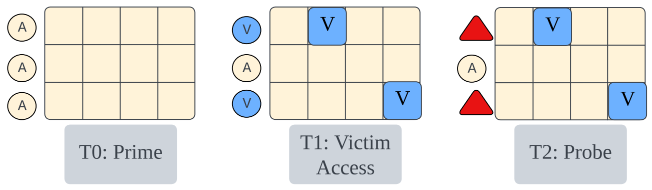


Figure 1: Illustration of the Prime+Probe attack. A: attacker, V:victim, square: cache block, triangle: cache miss of attacker

sharing. For example, in a multicore system, memory controllers are typically shared among multiple cores. An attacker can keep sending requests to a memory controller and observe the delays of those requests [42]. An increased delay indicates that there are other parties sending requests to the same memory controller. Cache ports, unpipelined algorithmic units, network-on-chip, and other shared components all have vulnerabilities to potential contention-based attacks [3, 4, 12, 28].

2.3 Constant-Time Programming

Constant-time programming is a programming principle to direct programmers writing programs that are *safe* from side-channel attacks. Constant-time programming have two main rules to mitigate cache-based side channels: i) *no branch on secrets (control-flow security)* and ii) *no secret-dependent memory accesses (data-flow security).* Prior works [34, 37, 46, 47] have proposed various solutions to *automatically* transform programs into their so called "constant-time" versions.

There are multiple ways to implement the first rule mentioned above, including *control flow linearization.* The idea is to execute *both sides* of a secret-dependent branch regardless of the true branch outcome. A constant-time programming compiler transforms a given original (insecure) program into its branchless version by keeping a "taken" predicate for each branch region. Consequently, both the "if" path and "else" path are executed, and the "taken" predicate determines how to *merge* the results of the "if" and "else" blocks when exiting the current branch region [9]:

```
if (secret) {                       taken = secret;
    A;                              A;
} else {               ⟹           B;
    B;                              Merge(secret, A, B);
}
```

For the second rule, which is the main target of this paper, dataflow linearization (explained below) is a common technique. To clarify, we say a memory access is "secret-dependent" if an attacker can infer secrets from the memory address that is accessed. In the following example, we show how a secret is *leaked* from a secret-dependent access:

```
void histogram(int in[], int out[]) {
    ... ...
    for (i=0;i<SIZE;i++) {
        int v=in[i];
        if (v>0) t=v%SIZE;
        else t=(0-v)%SIZE;
        out[t]=out[t]+1;
    }
}
```

In this example, array *in* contains secret inputs. A sufficiently powerful attacker (e.g., using prime-and-probe attack) would be able to learn the value of each element $in[i]$ by observing which address is accessed by $out\ [t]$. Consequently, we say that the memory access $out\ [t]$ is "secret-dependent".

In constant-time programming, *all* possible addresses of a secret-dependent memory access (with any input) form what is called a **Dataflow Linearization Set** in this paper. For example, the dataflow linearization set of the memory access $out\ [t]$ in the previous example is *all* elements in array *out*. Given a dataflow linearization set *DS*, **Dataflow Linearization** eliminates cache side channel by accessing each and every element of *DS* in every execution instance so that the attacker *cannot* figure out which of these accesses was really the intended one. Note that, addresses in dataflow linearization set are often continuous. A software-mitigated version of the example code fragment above is as follows:

```
void histogram(int in[], int out[]) {
    ... ...
    for(i=0;i<SIZE;i++) {
        int v=in[i];
        if(v>0) t=v%SIZE;
        else t=(0-v)%SIZE;
        for (j=0;j<SIZE;j++){
            int p = out[j];
            out[j] = (j==t)?p+1:p;
        }
    }
}
```

It is to be noted that, in this case, the dataflow linearization set of memory access $out\ [t]$ is array *out*. In the transformed (secure/mitigated) program, each load requires accessing *all* elements in the dataflow linearization set while each write requires first reading the data out and then writing it back. Clearly, by accessing each and every element in a dataflow linearization set, the mitigated program leaves "secret-independent" footprints. Note also that, the elements in the dataflow linearization set can be of different *strides* based on the threat model assumed (more on this below). For example, if the attacker cannot tell memory accesses to the same cache line from one another, the *stride* in the dataflow linearization set is the size of a cache line (e.g., 64 bytes) [37]. As a result, the size of dataflow linearization set is reduced. However, even an optimized dataflow linearization set can still be *very large.* As a result, accessing an entire dataflow linearization set may cause significant performance penalties in many practical scenarios.

2.4 Threat Model

While the published works in the literature employ different threat models [11, 21, 28], in this work, we assume that the attacker *cannot* directly access victim's data, though the attacker and the victim are assumed to be executing on the same machine and share the same (target) cache. The attacker is assumed to have access to the source/binary of the program and seeks to leak secret-dependent computations via the potential cache-based side channels. We further assume that the victim program does *not* share any writable data lines. Note that similar attack models have been used in prior research [9, 21, 34] as well. By the very definition of "sharing", the cache is assumed to be *not* partitioned between the attacker and the victim. We do not have a constraint on what level of the cache hierarchy the attacker and the victim are sharing. More specifically, the attacker and the victim can be running on the same core, in which case they share private caches (say, L1 and L2) as well as the last-level cache (LLC). Alternatively, the attacker and the victim could be running on different cores, in which case they only share the LLC. Also, we do not stipulate anything on the "inclusivity" of the caches in the system: caches can be inclusive, non-inclusive, or exclusive (and inclusivity does not influence the effectiveness of our work).

In this work, our focus is on an "access-driven" attacker that can observe the *changes* in the target cache, via *Prime+Probe* (explained above), with the goal of inferring secrets from the victim's program. We do not focus on leakages by operations other than memory accesses such as computations in unpipelined hardware structures [40]. The leakage from such structures can be easily eliminated by constant-time programming. We do not discuss the leakage from branch prediction units either, which could be handled by a technique known as branch linearization [9] in constant-time programming.

Note that, with control flow linearization and dataflow linearization, instruction and data memory access traces are identical regardless of the secret value. As a result, no leakage can originate from memory/storage units such as TLBs, last-level caches (LLCs), and memory controllers.

Also, we assume that the attacker *cannot* distinguish accesses to the same cache line from one another. This assumption is due the fact that the cache side channel attacks are generally at the granularity of "cache lines". Some prior works [26, 51] have demonstrated that cache side channels could even be formed at smaller granularities (e.g., in the context of bank conflict-based attacks). But, such bank conflicts are not reported in newer architectures [26]. As a result, we do *not* consider bank conflict attacks in our work. Consequently, the stride of the dataflow linearization set is of the size of a cache line (64 bytes) in this paper.

Prior research [52] has discussed that there is *no* clear "security abstraction" between software and hardware; and, this can make constant-time programming insecure. In particular, the main concern about secret-dependent memory access is "silent stores" [40]. However, if or how silent stores are implemented in modern commercial architectures is *not* publicly available at the time of this writing. Hence, in this paper, we do *not* consider the potential effects of silent stores. Since our work is based on existing constant-time programming, we leave the silent store issue to a future study.

3 MOTIVATION

In this section, we motivate for our proposed hardware structure for constant-time programming.

3.1 Performance Problem of Constant-Time Programming

Previous works [9, 21, 34, 37, 47] provide various constant-time programming tools with reasonable performance overheads. However, the focus of such prior works has been on "small" dataflow linearization sets. This is because most of the benchmark programs tested in these prior studies come from the *cryptography libraries*, which typically have look-up tables of small-to-moderate sizes. It is noticeable that not only cryptography libraries are secret sensitive, but also common processing tasks, especially in the era of cloud computing. The processing tasks, such as statistics-related programs[7] and graph processing[8], can be vulnerable to data leakage from cache side channel. We target the programs of which dataflow linearization sets are not as small as those in cryptography libraries.

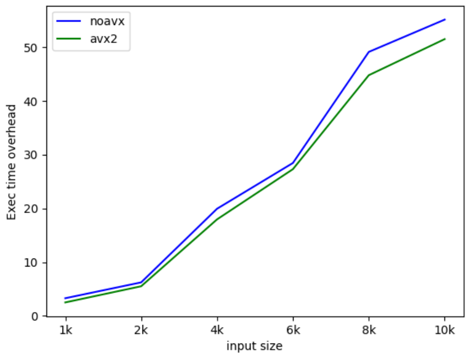


Figure 2: Overhead in Histogram with various size of dataflow linearization set.

To better understand how prior works scale with dataflow linearization set size, we have tested two application programs from [9], but with *varying* input sizes. The results we have collected with *Histogram* are plotted in Figure 2, and show how much the overhead is, with respect their insecure (original) versions. Note that the original input sizes used in these benchmark programs is 1,000 for *Histogram*, which makes the sizes of dataflow linearization sets to be $\frac{1000}{64}$(note also that the size of a dataflow linearization set depends on the data flow as well as the size of the secret input; also cache line size, 64 bytes, is the stride for dataflow linearization set in this case). As can be seen from the results presented in Figure 2, the overhead observed with the default inputs is twice the number of the cycles of the "original" ("insecure") programs. However, as the input size grows, the overheads grow very rapidly, and, with

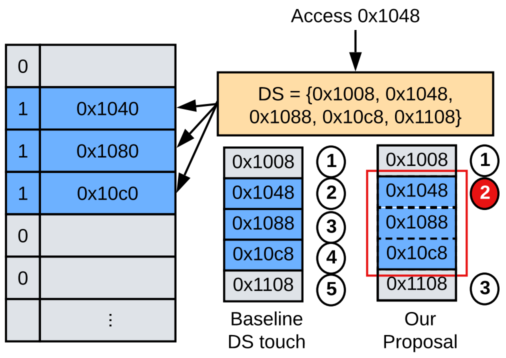


Figure 3: An example of accessing a dataflow linearization set. The targeted load address is 0x1048, data linearization set of that address is DS={0x1008, 0x1048, 0x1088, 0x10c8, 0x1108}. In cache, we have lines of 0x1040, 0x1080 and 0x10c0 valid. If we issue every address in DS to cache, there are 5 requests. However, If we know validness, only 3 requests are required. See Section 3.2 for more detail.

an input size of 10,000 in *Histogram*, we see that, even with the support of avx2 optimization, the performance overheads can be as high as 50 times of the cycles of the original programs!

As shown in [9], there is no sensitive branch in *Histogram*. As a result, we can contribute the overhead mainly to the dataflow linearization, which is to access the elements in the dataflow linearization set with stride. We test *Histogram* with cachegrind [1], a profiling tool that prints out statistics of a cache, including the number of cache accesses and the number of cache misses. The statistics reported below give the number of accesses to the L1 data/instruction cache (L1d ref/L1i ref) and the number of misses in the LLC cache (LL misses) for three versions (original, secure, and secure with the avx2 support) of *Histogram* with an input size of 10,000.

Input size	L1d ref	L1i ref	LL misses
origin	142,154	510,720	3793
secure	18,912,170	138,380,746	3796
secure with avx	19,022,174	83,230,746	3807

From the table above, we can eliminate the extra accesses to main memory (DRAM) as the cause for very high overhead. This is because the number of LLC misses represents the main memory accesses. The significant increase in L1i/L1d ref indicates that the extra accesses to the data cache and the execution of the extra instructions (for address calculation mainly) have significant influence on performance.

3.2 Can the Number of Accesses Be Reduced?

Our idea is based the insight illustrated by the following example where an attacker *cannot* distinguish accesses to the lines already in the cache from one another.

Normally, a mitigated program needs to access each and every element in a dataflow linearization set. Otherwise, it might *leak* information via side channel. However, we observe that, accessing each and every element in a dataflow linearization set is only necessary in a "worst-case" scenario. For instance, consider the example shown in Figure 3. In this example, the secret-dependent memory access is a load from 0x1048, and the dataflow linearization set of this access is *DS*={0x1008, 0x1048, 0x1088, 0x10c8, 0x1108}. A straightforward mitigation strategy such as [9] needs to load *all* the elements in *DS*, which takes a total of 5 accesses. However, if the program could somehow know the **existence** of each element in *DS* when it is about to issue the memory access to 0x1048, the number of accesses could be reduced. In this example, since cache lines for 0x1048, 0x1088 and 0x10c8 all exist in the cache, the mitigated program only needs to issue one (real) access to the address 0x1048 and then issues accesses to 0x1008 and 0x1108 to ensure that all elements are in the cache, regardless of which address is accessed. Note that this results in a total of 3 accesses. We emphasize that selectively accessing a *subset* of *DS* in this way does *not* leak any information because i) the cache status before accessing 0x1048 is secret-independent, as previous cache-side channels are all mitigated, and ii) the selective access does not bring/evict any lines to/from the cache, and as a result, the attacker *cannot* learn which of the three lines has been accessed. One potential concern is that the cache replacement side-channels can still tell the accesses apart. However, this problem can easily be addressed by *not* updating replacement bit (LRU bit) if the access is secret-relevant.

The above example demonstrates the case of accesses to the dataflow linearization set of a read operation. For accesses to the dataflow linearization set of a write operation on the other hand, the **dirtiness** information is required, which indicates whether the line has its dirty bit set to 1 in the cache. This is because, a write operation modifies the dirty bit of the corresponding line in the cache and this modification cannot be handled in the same way as the replacement bit (i.e., by skipping the update), due to cache coherency. In this case, only one access is issued to DS_{dirty} (a subset of *DS*). Note that the lines in DS_{dirty} are all in the cache and have their dirty bits set to 1. This insight can help the mitigated program to reduce its number of accesses to the cache, as long as DS_{exist} / DS_{dirty} is *not* empty. Since a dataflow linearization set is usually to be accessed more than once in cryptography libraries (e.g., a look-up table is checked many times) as well as other programs with security concerns, $DS_{exist/dirty} \neq \emptyset$ holds in many cases.

While guaranteeing security and providing performance improvement for programs of various data linearization set sizes, our technique is most efficient when the data linearization set of a program can wholly fit into cache. When the size of the linearization set is larger than the cache size, with some naive cache replacement policies (e.g., LRU), frequent capacity misses can happen. A straightforward way to deal with this problem is to change the replacement policy. We will discuss an insight that utilizes coarser attack granularity in the memory controller to handle this problem (Section 6.5).

4 DESIGN DETAILS

It should be clear from the discussion above that we need to, somehow, maintain the "information of existence" and the "information of dirtiness" in the system, and make the information available to the application program to be protected against side-channel

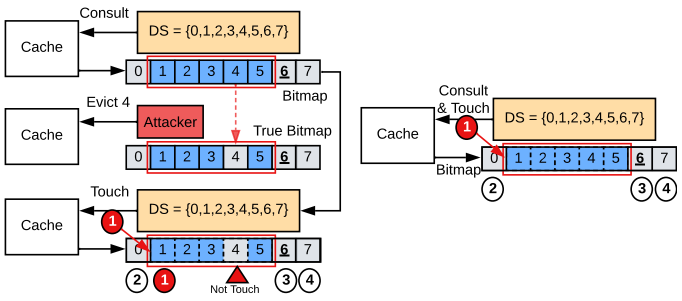


Figure 4: Illustration of the functionalities of the proposed instructions.

attacks. Unfortunately, current architectures lack both i) hardware structures that can help us maintain such information and ii) interface (instructions) to expose them to the application program. These missing components are what our research explores and evaluates.

4.1 Proposed Instructions

In this section, we give the details of our proposed interface, i.e., the new instructions needed for our proposed scheme to work. We start by observing that maintaining the existence and dirtiness information can require a substantial amount of space. For example, with a 4GB main memory, there are $4GB \div 64B = 2^{26}$ cache lines in total. New instructions should not provide information of all those lines as a whole since it is too large.

We propose two new instructions to be added to the micro-operation set of X86-64 ISA, to capture the existence/dirtiness information for 64 lines – size of a single page:

$$page\ size = 4096\ bytes = 64\ lines \times 64\ bytes\ per\ line.$$

If we use one bit to indicate the existence/dirtiness of one line, our new instructions would return 64 bits for existence information or 64 bits for dirtiness information.

Intuitively, the granularity of managing dataflow linearization set (DS) affects the number of lookups into bitmap table. Specifically, with a coarser granularity, the number of lookups gets decreased. In this work, we set the granularity of managing DS (denoted by M) to be the *page size* ($M = 12$) for two reasons: first, virtual address and physical address have same 12 least significant bits (same offset within a page). As shown in Algorithms 2 and 3 (and discussed later in the paper), Bitmask needs to be generated for preprocessing (further details can be found in Section 5). Bitmask generation takes the least M significant bits of the physical address as input. By setting the granularity to page size (12 bits), we can use the virtual address to generate this information. Second, page size is often the "granularity of attack" at the memory controller level. This offers us with more choices for accesses in our algorithms, which will be discussed in detail in Section 6.5.

Our new instructions enable users/programs to access the existence/dirtiness information. However, these instructions cannot just implement a simple information load from new hardware structure; rather, our instructions should i) launch a cache access, i.e., load/store from/into cache and ii) load the existence/dirtiness information at same time. To see why this is necessary, let us focus on the left side of Figure 4. In this example, element 6 is the the data element to be loaded by the program, which is "secret-dependent". The dataflow linearization set for this access is $DS = \{0, 1, 2, 3, 4, 5, 6, 7\}$. For simplicity, we assume that the lines in DS are within the same page. Suppose that lines of 1, 2, 4 and 5 are in the cache at the beginning. If our new instruction simply loads the existence information, the program would get $Exist_p = \{1, 2, 3, 4, 5\}$ at first (we use, for simplicity, set $Exist_p$ rather than 64 bits to denote the information returned to the program). Let us assume now that an attacker evicts the line that holds element 4 from the cache *right after* loading the existence information. In this case, the existence of the elements should be changed to $Exist_t = \{1, 2, 3, 5\}$. However, the victim program is unaware of the change, meaning that it issues accesses to elements 0, 6 and 7, and only one access to $Exist_p$ – to element **1**. As a result, element 4 does not exist in the cache after the accesses issued by the victim. Based on this observation, the attacker learns that element 4 is *not* the victim's required access and subsequently excludes some candidates of the victim's secret. Note that the above information leakage is due to the fact that the existence information obtained by the victim *cannot* accurately represent the existence of the lines by the time the victim is accessing the cache.

Motivated by this discussion, we propose two new instructions, one "load" instruction and one "store" instruction. Each of these instructions, at the same time, i) performs a *cache access* and ii) loads existence/dirtiness information. As shown on the right side of Figure 4, our new load instruction loads the information of existence and accesses the cache in one step. It is to be noted that the *cache access* of our new instructions is not the same as the *cache access* of normal *load* and *store* instructions. For example, as shown on the right side of Figure 4, element 6 is the requested data item but the new instruction does *not* load it. Furthermore, the new instruction does *not* forward misses to the next level in the cache hierarchy or to the main memory, for security reasons. So, this new load instruction may *not* load the data requested by the original program. It is the same with the new store instruction, i.e., it may *not* write the data into the requested address.

Instead, to ensure security and functionality of the application program, *we design algorithms for loading and storing elements in the dataflow linearization set*, which are discussed in detail in the next section. However, for now, let us define our new instructions more formally:

1) Our new load instruction, **CTLoad**, has one input, *address*, and two outputs, *data* and *existence*. *data* may or may not be the actual data requested by program originally; that depends on whether the line of the address is in the cache or not. The *existence* information on the other hand is of 64 bits (8 bytes), and helps the program to get the actual data using the algorithm explained in the next section. The semantics of CTLoad is more precisely shown with the pseudo-code below:

```
Input: address
Output: data, existence
IF address hit in cache
    data = load(address)
ELSE
    data = 0
```

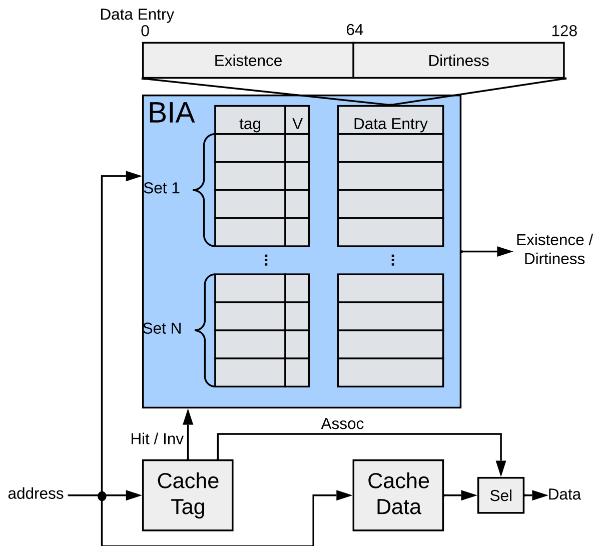


Figure 5: Proposed hardware design. When a request arrives at cache, the address also goes to BIA which outputs Existence/Dirtiness information. All cache hit/invalid signal from cache tag will also goes to BIA for Data Entry updating.

```
existence = getExistenceByPageIdx(address)
return data, existence
```

2) Our new store instruction, **CTStore**, has two inputs, *address* and *data*, and one output, *dirtiness*. *data* may or may not get stored into *address* provided by CTStore. With *dirtiness* (64 bits = 8 bytes) and some additional processing (explained shortly) after CTStore, *data* is stored into *address*. The semantics of CTStore can be more precisely defined as follows:

```
Input: address, data
Output: dirtiness
IF address has dirty bit set to 1 in cache
    store(address, data)
ELSE
    DO NOTHING
dirtiness = getDirtinessByPageIdx(address)
return dirtiness
```

4.2 Proposed Hardware Structure

Our proposed hardware structure **BIA** (BItmAp) is essentially a *table* of "bitmap" entries. As shown in Figure 5, each entry of BIA records the existence and dirtiness information of 64 lines in a page. The existence information is of 64 bits, with each bit indicating the existence of one line in the cache. The dirtiness information is also 64 bits, and specifies if the corresponding line has its dirty bit set to 1 in the cache. The tag of the entry records the index of the page that it refers to.

BIA is similar to a cache memory; hence, the placement and replacement policies employed by conventional caches can be applied here. In our design, we opt to use a set-associative policy for placement and an LRU policy for replacement. An entry in BIA is installed when the corresponding address required by CTLoad or CTStore is *not* found in BIA, triggered by a miss after tag comparisons in BIA. An entry is allocated and initialized with the existence and dirtiness bits set to 0, and it fills the tag with the page index.

BIA monitors the cache for any update. When there is a hit in the cache and there is an entry in BIA for the corresponding page of that memory address, the corresponding existence bit in the entry is set to 1. In addition, the dirty bit of that line in the cache is checked; this bit is used for updating the dirtiness information in the BIA entry. When there is an invalidation, BIA checks if the page index of that line hits in the BIA. If it hits, the existence and dirtiness bits for that line in the entry for that page are set to 0. When there is a change in the dirty bit of the cache lines, the corresponding entry is updated with the dirtiness information. One caveat though is that the BIA may not be *consistent* with the cache. Specifically, at the initialization time, some lines of the page may already be in the cache, but BIA initiates the existence and dirtiness with all 0s. However, this inconsistency does *not* hurt security in any way. We elaborate on this further in Section 5.3.

Note that the BIA can be placed into the L1 cache or the L2 cache. For an L2-resident BIA though, the behavior of CTLoad and CTStore is slightly different as, in this case, those two instructions will *bypass* the L1 cache; they will check the L2 cache and the L2-resident BIA. Whether to place the BIA into the L1 cache or the L2 cache depends largely on the tradeoff between cache capacity and cache latency. To make it clear, L1 cache has a lower accessing latency than L2 cache but has smaller capacity. As a result, lines are more likely find themselves hit in L2. However, in a L2-resident BIA design, CTLoad and CTStore instructions need to bypass the L1 cache for security, resulting in extra overhead.

5 PROPOSED ALGORITHMS

As discussed in the previous section, the proposed CTLoad and CTStore instructions themselves *cannot* complete the whole functionality of loading/storing data from/to the targeted address in a secure fashion. In this section, we first provide our algorithms for "secure" loading and storing with the newly-proposed instructions. We also *prove* the functionality and security of the proposed algorithms. Finally, we discuss our algorithms under different constraints.

5.1 Details of Our Algorithms

We now present the algorithms we have implemented in software to finish secure loading and storing via the CTLoad and CTStore instructions. It is worth mentioning that the previous constant-time programming methods/tools [9] provide programs with dataflow linearization sets of their secret-dependent memory accesses at compile-time. We utilize those methods/tools for generating the dataflow linearization sets.

In Algorithms 2 and 3, first, a dataflow linearization set is *divided* into pages; this is achieved by grouping lines by their page index bits (i.e., the 13th-64th bits in a 64-bit machine). Within the loop, the address input for CTLoad/CTStore needs to be regenerated. Note that the targeted address of the program may not be within current $page_i$. In such a case, an address that is within the current page is picked.

```
Algorithm 2: Use CTLoad to load
    Input: address: $ld\_addr$
    Output: data: $ret\_data$
1  $DS$ = getDataflowLinearizationSetOfAddr($ld\_addr$)
2  Get the set of pages $PAGES = \{page_0, page_1, \cdots, page_n\}$ that
     covers $DS$
3  for $page_i$ in $PAGES$ do
4      $addr\_to\_read = page_i \mid ld\_addr\,[11:0]$
5      Get $Bitmask$ of $page_i$
6      $data, existence$ = CTLoad($addr\_to\_read$)
7      $tofetch = Bitmask\ \&\ \sim existence$
8      $fetchset$ = generateAddrs($page_i$, $ld\_addr$, $tofetch$)
9      for $address$ in $fetchset$ do
10         $tmpdata$ = LOAD($address$)
11         $data = (address{=}{=}addr\_to\_read)?tmpdata{:}data$
12     $ret\_data = (ld\_addr$ in $page_i)?data{:}ret\_data$
13 return $ret\_data$
```

```
Algorithm 3: Use CTStore to store
    Input: address: $st\_addr$
1          data: $st\_data$
2  $DS$ = getDataflowLinearizationSetOfAddr($st\_addr$)
3  Get the set of pages $PAGES = \{page_0, page_1, \cdots, page_n\}$ that
     covers $DS$
4  for $page_i$ in $PAGES$ do
5      $addr\_to\_write = page_i \mid st\_addr\,[11:0]$
6      Get $Bitmask$ of $page_i$
7      $ld\_data, existence$ = CTLoad($addr\_to\_write$)
8      $st\_data\_tmp = (st\_addr$ in $page_i)?st\_data{:}ld\_data$
9      $dirtiness$ = CTStore($addr\_to\_write$, $st\_data\_tmp$)
10     $tofetch = Bitmask\ \&\ \sim dirtiness$
11     $fetchset$ = generateAddrs($page_i$, $st\_addr$, $tofetch$)
12     for $address$ in $fetchset$ do
13         $tmpdata$ = LOAD($address$)
14         $tmpdata = (address{=}{=}st\_addr)?st\_data{:}tmpdata$
15         STORE($adddress$, $tmpdata$)
```

Bitmask is generated based on the dataflow linearization set. *Bitmask* has 64 bits with each bit identifying if the corresponding line of that page is in the dataflow linearization set or not. *Bitmask* is required by our algorithm because, in a page, some lines might not be in the dataflow linearization set. For example, if DS={0x1080, 0x10c0, 0x1100, ..., 0x1f80, 0x1fc0}, $Bitmask = 111\ldots111_{62}00$ because the first two lines, 0x1000 and 0x1040, are not in *DS*. While *Bitmask* shows if a line is in *DS*, *existence* shows if a line is in cache.

In the load algorithm, *addr_to_read* is passed to *CTLoad*. As discussed before, *data* may or may not be the data from *addr_to_read*, because CTLoad does *not* forward request to the next level of cache or main memory. The job of generateAddrs is to generate a set of addresses by checking the bits in $tofetch$. Specifically, if bit i is 1, a new address is generated using this formula: $address = page[63:12] + i << 6 + st\_addr\,[5:0]$. The first part of this formula is to obtain the page index; the second part is to obtain the line index within the page; and the third part is to obtain the offset within the line. After the generation, generateAddrs puts the new address into *fetchset* and checks the next bit of *tofetch*.

The store algorithm is more involved. CTStore *cannot* be directly used because *address_to_write* in the current iteration may *not* be the same as the targeted address. If $address\_to\_write \neq st\_addr$ and *addr_to_write* is already dirty in the cache, then CTStore corrupts its content. Our solution is to use first the CTLoad instruction. Note that, doing so would prevent the data in *address_to_write* from being corrupted. First, let us assume that there exist no other processes that access the cache in the meantime. As shown in Figure 6(a), if the line that includes *address_to_write* is dirty in the cache (i.e., D=1) when CTLoad accesses the cache, the returned data is correct. In this case, CTStore is issued with *address_to_write* and the correct data (shown as the yellow circle). Figure 6(b) shows that, if the line that includes *address_to_write* does not exist (i.e., V=0) or is not dirty in the cache (i.e., D=0) at the time CTLoad accesses the cache, the returned value is not correct (i.e., it is fake, shown as the red triangle). Since there are no other processes bringing in the line, CTStore will find that line absent in cache, so that fake data will *not* be written back into the cache.

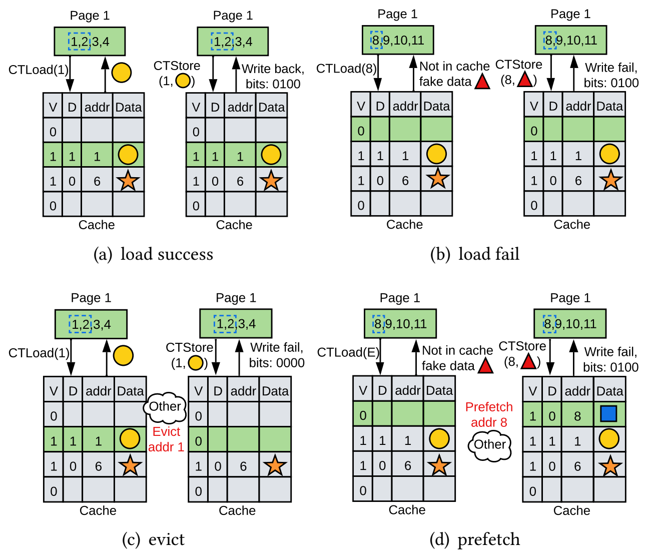


Figure 6: Using CTLoad in the store algorithm. Circle: dirty data in cache; Star: clean data in cache; Triangle: fake data returned by cache; Square: preloaded data(also clean) in cache; V: valid bit, D: dirty bit; For every subgraph, the left shows line 7 in Algorithm 3 and the right shows line 9. In between, there might be other process changes the state of cache, which is shown by red text.

Let us now consider a more general case where there can be other applications/processes using the "same cache" at the "same time". In cases where the line that contains *addr_to_write* exists in the cache when CTLoad is executed but later gets evicted by other processes, as depicted in Figure 6(c), CTStore will find that line absent in the cache and DO NOTHING(see Section 4.1(2)). Figure 6(d) illustrates what will happen if CTLoad encounters a miss and returns the fake data but that line then gets brought into the cache by the prefetcher. Note however that, in such a case, the line should not be dirty (i.e., D=0) in the cache; as a result, CTStore cannot write the fake data back into the cache.

5.2 Functionality Proof

In this part, we prove the functionality of our algorithms that work along with CTLoad and CTStore. The main functionality of our algorithms is to *load/store data from/to the requested address while not altering the contents in other addresses.*

In the algorithm for load, there is no concern for data corruption. Hence, all we need to do is to identify if the requested data gets loaded and if it is returned by the algorithm. When the loop iterates to the page of the requested line, if that line exists in cache, CTLoad returns the data from the requested line. If, on the other hand, it does not exist, the address will be put into *fetchset* and will get loaded later. The correctness for algorithm for store is similar to that for load. There is also no wrong modification to any other addresses, which we have discussed earlier when discussing the algorithms. As pointed out before, the BIA is not fully consistent with the cache. This is because, when a new entry is allocated, some lines in the related page of this entry might have already been in the cache. Note however that, this will *not* influence the correctness of the functionality of our algorithms. Since the existence information recorded in the BIA is a **subset** of the authentic (ground-truth) existence information, the missed requested address is still put into *fetchset*.

5.3 Security Proof

In this part, we prove that our proposed algorithms *guarantee* the constant-time programming principle. For each secret-dependent memory access, a dataflow linearization set **DS** is generated. Elements in DS stride with the size of a cache line. Let us assume that $DS = \{L_1, L_2, L_3, L_4, \ldots\}$, where L_i is the physical address of line i; an existence set $Exist = \{C_1, C_2, C_3, C_4, \ldots\}$, where C_j is the physical address of line j in the cache that is *recorded* to exist by the BIA; and a dirtiness set $Dirty = \{D_1, D_2, D_3, D_4, \ldots\}$, where D_k denotes the physical address of line k in the cache that is *recorded* dirty in the BIA. We have $Exist_t$ and $Dirty_t$ for every time point t. Clearly, $Dirty_t \subseteq Exist_t$ and both $Exist_t$ and $Dirty_t$ are *subsets* of the "actual" existence and dirtiness information in the cache. Note that, for security purposes, $Exist_t$ and $Dirty_t$ do *not* need to be exactly the same as in the cache status.

We use "induction" to prove that $Exist_{t(i)}$ is "secret-independent", where $t(i)$ is a timestamp. For example, $t(1)$ is the time when the mitigated program is about to access DS for the first time. Obviously, $Exist_{t(1)}$ is secret-independent. Suppose $Exist_{t(r)}$ is secret-independent. In our algorithm, DS is split into groups by page index. So, DS is split into $DS_{p_1}, DS_{p_2}, DS_{p_3}, \ldots$. Here, DS_{p_i} stores the lines in DS that are in page p_i. In our algorithm for data load, a CTLoad is issued for every DS_{p_i}. Let us assume that, at time $t(r)$, *CTLoad* for page p_i arrives at the cache and the BIA. Note that *CTLoad* does *not* change the contents in the cache no matter what the secret is. *tofetch* is generated by the existence information $DS_{p_i} - DS_{p_i} \cap Exist_{t(r)}$ which has nothing to do with the secret. Therefore, accessing all the addresses in *tofetch* is secret-independent, which leaves "insensitive" footprints in the cache. Additionally, there is no secret-related accesses between accesses to DS. Consequently, the next time DS is accessed, $Exist_{t(r+1)}$ is secret-independent. A similar proof strategy can be used for the algorithm for the store instruction as well. The additional part would be the trick, which we have discussed earlier, of invoking CTLoad first. Note however that executing CTLoad-/CTStore does not change the meta-contents of the cache (i.e., they do not change anything except data).

6 DISCUSSION

In this section, we go over a few important aspects of our proposed approach.

6.1 Comparison with Cache Pinning

PLcache[44] is a cache design that *locks* lines in cache (the locked lines *cannot* be evicted). PLcache+preload[19], which is based on PLcache, preloads and locks all sensitive lines *in advance*, to remove cache side-channels. While PLcache mitigates the problem of large data linearization set because only one access to cache will be issued once data is pinned (although pinning data into cache still requires accessing the whole data linearization set), it has limitations in terms of security and fairness. First, PLcache does *not* achieve the same level of security as our work since it does *not* mitigate information leakage from dirty bits and LRU bits: those bits can still be "secret-dependent". Once those pinned cache lines get "unpinned", the memory access patterns to those lines will be leaked from the replacement and writeback behaviors. Second, PLcache does *not* provide the same level of "fairness of service", compared with our work. Although cache pinning is beneficial to the protected program, it hurts the performance of other processes by exclusively holding a portion of the cache, i.e., it reduces the effective cache capacity. Furthermore, this problem can be more severe when the dataflow linearization set is large as the protected program would occupy a significant portion of the cache space, and leave few lines for other processes. Even when the process gets switched out by the OS, its large pinned area cannot get released. Our proposed design, in contrast, does *not* lock lines in the cache, and so, it can be considered "fair".

6.2 Bitmap Exposure Discussion

One might also have a concern regarding whether the BIA has any negative side effect on security, given that the attacker can potentially be able to read the existence and dirtiness information from the BIA, saving attackers from more costly Prime+Probe-like attacks to reveal the same information. However, we first note that the existence and dirtiness information from the BIA *does not* leak any information for the protected program. For unprotected program, we propose to prevent direct exposure of the bitmap information to users by packing the **CTLoad** / **CTStore** with the follow-up LOAD / STORE instructions (i.e., packing the whole algorithms 2 and 3) into macro-operations in X86-64 ISA. By only exposing the macro-operations to users, the sensitive bitmap reading instructions **CTLoad** / **CTStore** *cannot* be called directly, and the loaded existence/dirtiness information remains invisible to users. We leave the integration of macro-operations to a future study.

6.3 Cryptography Libraries

As mentioned in Section 3.1, our technique does not target workload whose dataflow linearization sets are as small as those of cryptography libraries. The dataflow linearization set in cryptography library is usually small. For example, in AES encryption, accesses to T-table are secret dependent [5]. The size of dataflow linearization set is |T-table| = 1024 bytes, which equals to 1024/64=16 cache lines. BIA implementation, on the other hand, groups 64 cache lines together, which means the dataflow linearization set of AES even does no exceed the boundary of a single BIA entry. However, BIA can still benefit cryptography libraries from the security angle and does not cause much performance penalty. We will show that in Section 7.3.3.

6.4 Putting BIA into LLC

In Section 4.2, we briefly discussed the options of placing BIA into L1 or L2 caches. Now, we consider the feasibility of placing BIA into LLC (last-level cache). In many modern architectures, LLC is sliced and distributed, and the cache lines of LLC are hashed into slices. Apart from cache side-channels introduced in Section 2.1, LLC also suffers from the interconnect side-channels. More specifically, an attacker could learn which slice/core is sending/receiving information from another slice/core by observing the traffic latency [28]. Hence, the LLC interconnect leakage granularity also relates to LLC slice ID hashing function. Prior works on reverse engineering the slice hashing function [49, 50] indicate that hashing function takes some bits of physical address as input.

Let us denote n-bit physical address as $A = (A_{n-1}, A_{n-2}, \cdots, A_0)$, denote the index of the least significant bit used in slice hashing function as LS_Hash (e.g., $LS\_Hash = 12$ if A_{12} is the least significant bit used by the hash function), and denote the granularity of DS management as M (e.g., $M = 12$ if the granularity is of page size). Note that LS_Hash is no less than 6, because cache line is of size 2^6.

If $LS\_Hash \geq 12$ (e.g., as in the case of Intel Skylake-X [49]), it is feasible to put BIA into the LLC with $M = 12$. Since the traffic among slices leaks information at a granularity larger than page size, the traffic trace among slices/cores is identical with all different offsets within the same page. Hence, an attacker cannot gain information by simply observing the traffic. The implementation of this case is basically similar to the L2 cache BIA design – we bypass the L1 and L2 caches and fetch the BIA information from the LLC BIA. The following access in DS (lines 9-11 in Algorithm 2 and lines 12-15 in Algorithm 3) should also bypass the L1 and L2 caches.

If $6 < LS\_Hash < 12$, it is still feasible to put BIA into LLC, but doing so requires M be set to LS_Hash. Note that the leakage from the interconnect gets eliminated because addresses in the same DS management set (2^M addresses in a set) are resident in the same LLC slice and the attacker cannot distinguish, by just observing the traffic among LLC slices/cores, which address in this DS set is accessed. The implementation of this case is quite similar to that of the previous case. M is reset to LS_Hash. The BIA management granularity is no longer the page size; instead, it is $2^M = 2^{LS\_Hash}$. As a result, there are more CT_Load and CT_Store traffic among slices and cores, and the traces across the slices and cores do not leak any information.

Table 1: Gem5 Configuration

Configuration	Parameter
CPU	DerivO3CPU
L1d cache	64 KB, 2 cycles latency
L2 cache	1MB,15 cycles latency
Last Level cache	16 MB, 41 cycles latency
BIA	in L1d/L2 cache, 1 KB, 1 cycle latency

If $LS\_Hash = 6$ (e.g., as in the case of Intel Xeon E5-2430[50]), it is not feasible to implement BIA in the LLC, as, in this case, the continuous cache lines are distributed across different slices.

6.5 Granularity-Based Optimization

Information leakage granularity varies across different memory units. The finest granularity is of a "cache line" size, given that bank conflicts are not reported in recent architectures any more [26]). In addition to cache side-channels, LLC interconnects, memory controllers [42] and DRAM row buffers [31] can also leak information. The granularity of the LLC leakage is 2^{LS_Hash}. With a closed-row policy, the granularity of memory controller leakage is no less than the page size. In this work, we have opted to set the DS management granularity to the page size because it opens up an optimization space. It is to be noted that, if the entire DS cannot fit into cache, then the access to DS (lines 9-11 in Algorithm 2 and lines 12-15 in Algorithm 3) might cause frequent cache line replacements and eventually hurt the performance. However, if $LS\_Hash \geq 12$, we can wrap those parts with the following code:

```
if (sizeof(fetchset) < threashold) {
    line 9-11 in Algorithm 2
    (line 12-15 in Alorithm 3)
} else {
    directly load from DRAM
    (directly store to DRAM)
}
```

We set a threshold for the size of $fetchset$. If the size of $fetchset$ exceeds the threshold, we can bypass all the caches below and access the DRAM directly. In this case, we can avoid the frequent cache replacements.

7 EXPERIMENTAL EVALUATION

In this section, we first provide the details of our implementation and experimental setup, and then present and discuss the collected experimental results.

7.1 Implementation and Experimental Setup

We implemented our proposed BIA on top of the Gem5 simulation toolset [6]. The default values of the major experimental parameters of our implementation are listed in Table 1.

The BIA is similar, structure-wise, to a conventional cache memory, with tags and lines (blocks). We add our new instructions into the x86-64 ISA. Note that there is no direct store/load style instructions in x86-64; instead, it has instructions for **mov**ing data between the memory and the register file. During execution, such mov instructions are broken into "micro-operations" (micro-ops), which are technically equivalent to the load and store instructions. Hence, we add our new instructions in style of micro-ops.

We integrate our instructions into a state-of-art constant-time programming tool – Constantine [9], which is implemented on top of LLVM [20]. Constantine identifies *all* secret-dependent memory accesses in the program and replaces each of them with accesses to *all* of the addresses in the dataflow linearization set. We implement our algorithms in the Constantine's dataflow linearization library.

7.2 Benchmark

In this section, we present an experimental evaluation of our proposed approach to side-channel mitigation.

As mentioned in Section 3.1, common processing tasks are also vulnerable to side channel leakage; their dataflow linearization sets are of various sizes. For example, the dataflow linearization set of the histogram program shown in Section 3.1 is array **out**. The size of array **out** depends on the configuration, which gives rise to different dataflow linearization sizes. In this work, we primarily target programs whose dataflow linearization sets are much greater than those of the crypto libraries. This is because the crypto libraries do not have much performance decrease even with software-based constant time programming [9].

We provide evaluation results of programs from the Ghostrider benchmark [21]. We select five programs whose memory access patterns are either partially predictable or data-dependent (*Dijkstra*, *Histogram*, *Permutation*, *Binary Search* and *Heappop*), shown in Table 2.

Table 2: Programs with partially predictable or data-dependent memory access patterns in Ghostrider benchmarks [21] and their leakage description.

Program	Leakage Description
dijkstra	Access to not-yet-selected vertex with minimum distance to source vertex in each iteration leaks graph structure; Size of DS: $O\left(number\_of\_Vertices^2\right)$
histogram	Calculating bin number based on data value; Accesses to bins expose data; Size of DS: $O\left(number\_of\_Bin\right)$
permutation	Permutation *a[b[i]] = i* exposes b[i]; Size of DS: $O\left(length\_of\_array\right)$
binary search	Accesses to elements in array leak comparison trace; Size of DS: $O\left(length\_of\_array\right)$
heappop	Heap adjusting procedure brings different access patterns with different internal data values; Size of DS: $O\left(length\_of\_array\right)$

7.3 Performance Evaluation

Figures 7(a), 7(b), 7(c), 7(d) and 7(e) plot the "execution time overheads" of our approach(implement BIA in **L1** or **L2**) and software-based constant time programming (**CT**) compared to that of the *insecure baseline*. Note that the sizes of dataflow linearization sets and the number of visits to the dataflow linearization set are both very large (when using large inputs) in these three application programs. It explains why execution time overhead of software constant time programming increases very fast. With our design on the other hand, the observed performance overhead reduces significantly compared to the software constant time programming.

7.3.1 *Source of Performance Gain.*

We gather, from Gem5, the statistics on the number of executed instructions, number of accesses to the L1 instruction caches, number of accesses to the L1 data caches, and number of accesses to the DRAM. We calculate the ratio of all those statistics of **CT** to all those of **L1d BIA**. We plot part of the results in Figure 8. We also plot the execution time ratio in the same figure. As shown in Figure 8, our design greatly reduces the number of instructions, the number of accesses to the instruction caches and the number of accesses to the data caches, which all contribute to the overall performance. In comparison, the number of accesses to DRAM remains almost the same ($y \simeq 1$), which means the performance gain is *not* related to the DRAM accesses.The other programs in our benchmarks show similar results: the performance gain comes from reduced instruction numbers and reduced cache accesses, *not* from DRAM accesses.

7.3.2 *L1d BIA vs. L2 BIA.*

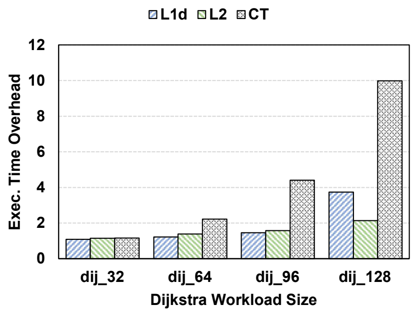
We evaluate performance with BIA in the L1d cache and the L2 cache. As shown in Figure 7, the performance of the L1d BIA design is better than that of the L2 BIA design in most of the tests, mainly due to the higher latency of the L2 cache. The L2 BIA design performs better than the L1 BIA design in dij_128 program (Figure 7(a)). Notice that the size of dataflow linearization set (DS) is 128*128*sizeof(int) = 64KB, which cannot fit into the L1d cache (L1d cache is 64 KB but there is some other data, e.g. output array). With the L1d BIA design, the self-eviction effect in the L1d cache causes the overall performance to drop, and with the L2 BIA design, 64KB sensitive data bypasses the L1d cache and can all fit into the L2 cache.

7.3.3 *Cryptography Libraries.*

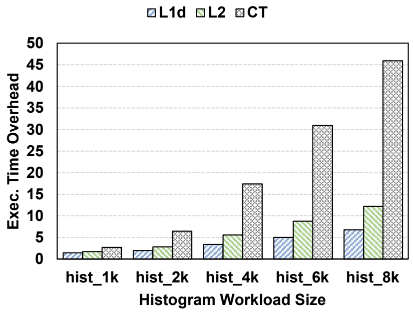
Although we mainly target workload whose dataflow linearization sets are greater than those of the cryptography libraries, we also perform evaluation using the cryptography libraries. Since the DS size is small for the crypto libraries, we provide only the results of L1d BIA and software constant-time programming. The baseline in these experiments is the insecure version. The y-axis shows execution time ratio of L1d or CT to the baseline.

Most of the evaluated programs in Figure 9 show slightly better results with constant time programming than L1d BIA. Because the DS size is even smaller than the size of cache lines grouped by a single BIA entry (Section 6.3), BIA provides little performance benefits. Also, there are more pre-processing and post-processing steps in our proposed algorithms (Algorithm 2 and Algorithm 3), such as grouping DS into page granularity and accessing addresses based on bitmap entries.

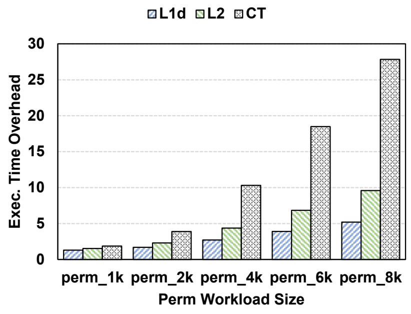
It is also noticeable that L1D BIA shows much better performance than the software constant-time programming in program Blowfish. The DS size in Blowfish is 1024 bytes, which equals to that of the AES program. The reason why our technique gives better results on Bliwfish is that, Blowfish has an expensive setup phase,



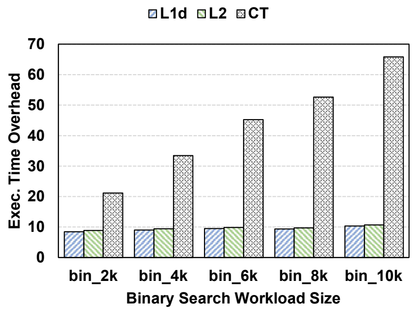
(a) dijkstra



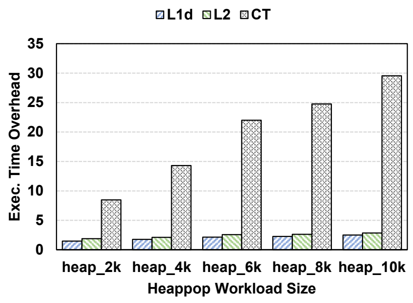
(b) histogram



(c) permutation



(d) binary search



(e) heap pop

Figure 7: Execution time overhead compared with insecure baseline. (a) dijkstra program with $number\_of\_Vertices = \{32, 64, 96, 128\}$**; (b) histogram program with** $number\_of\_Bin = \{1000, 2000, 4000, 6000, 8000\}$**; (c) permutation program with** $length\_of\_array = \{1000, 2000, 4000, 6000, 8000\}$**; (d) binary search program with** $length\_of\_array = \{2000, 4000, 6000, 8000, 10000\}$**; (e) heap pop program with** $length\_of\_array = \{2000, 4000, 6000, 8000, 10000\}$**.**

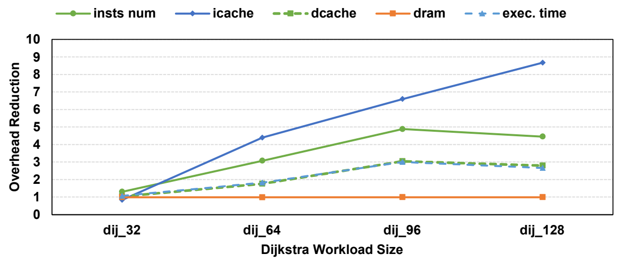


Figure 8: Overhead Reduction Ratio (in multiples) of software constant time programming to L1d BIA design. insts num: number of instructions; icache: number of accesses to instruction cache; dcache: number of accesses to data cache; dram: number of accesses to DRAM; exec.time: execution time.

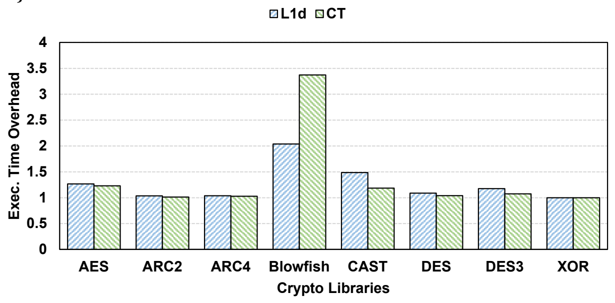
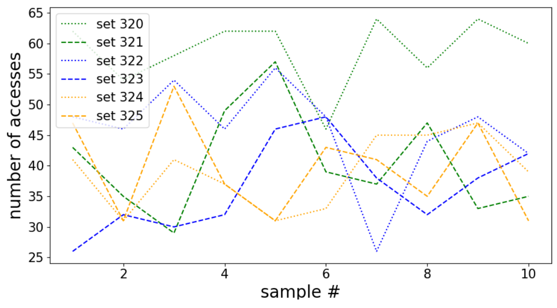


Figure 9: Execution time overhead of crypto libraries, with our L1d BIA implementation and software constant time programming respectively.

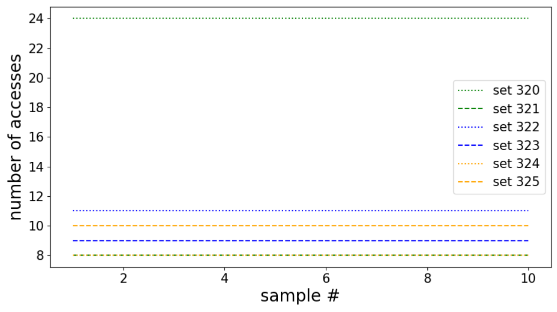
in which there are lots of data-dependent memory accesses. Hence, there are many more accesses to DS in Blowfish than AES, thereby amortizing pre-processing and post-processing overheads.

7.4 Security Test

As explained earlier, the cache side-channels originate from the secret-dependent cache access patterns. Consequently, if the cache access patterns are identical with all possible values of the secret, the secret will not get leaked from the cache. We modified Gem5 to output the number of accesses to each cache set. We provided programs with different secret values, gathered the number of accesses to each cache set, and found the statistics to be the same. Figure 10 serves as a representation of our result: we run Histogram_1k program with randomly-generated secret inputs and plot 10 samples of the original insecure baseline (Figure 10(a)) and our approach (Figure 10(b)). There are 2048 cache sets in our experiment setting. Due to the space concerns, we only present the number of accesses to cache sets 320-325. It can be observed from these results that, with different secret inputs, the cache access pattern varies in the insecure baseline. With our proposed design on the other hand, the number of accesses is identical across all 10 samples tested.



(a) Insecure baseline



(b) Our work

Figure 10: The number of accesses to cache set 320-325 in hist_1k program.

8 RELATED WORKS

In this section, we focus on the prior works that are most relevant to our work, in addition to those mentioned in the introduction and background sections.

SC-Eliminator [47] transforms a given program by preloading lines of dataflow linearization sets into the cache; thus, a sensitive memory access does not miss in the cache. Unfortunately, this approach cannot guarantee security because an attacker can evict the preloaded lines from the cache. Raccoon [34], on the other hand, employs Path ORAM (Oblivious RAM [39]). Oblivious RAM shuffles data in it so that programs can completely hide its data access patterns from other security domains. However, ORAM introduces significant runtime overheads that can have a devastating impact on application performance. In comparison, GhostRider [21] turns off caches and uses scratchpads for both instructions and data. It uses compiler to check whether the relevant data blocks are in cache, and if not, they are loaded from the memory. Note that GhostRider requires substantial changes to the underlying architecture.

Our work differs from these prior works in that it provides hardware support, with a new hardware structure and two accompanying instructions, for constant-time programming. Our work guarantees security with very small memory area overhead, and substantially improves over state-of-the-art.

9 CONCLUDING REMARKS

In this paper, we first identify the "large dataflow linearization set" problem in software-based side-channel mitigation. We then solve this problem by adding a new hardware structure into the architecture, to record the existence and dirtiness information of the cache lines and providing two new load/store instructions for accessing this information and exposing it to the application program. We also design load and store algorithms to safely access the secret-dependent addresses. Our experimental evaluation indicates that, with very small memory area overhead, the proposed approach is able to increase the performance of the side channel-mitigated programs by about 7x.

ACKNOWLEDGMENTS

We are thankful to the anonymous reviewers and our shepherd for their constructive feedback. This work was supported in part by the National Science Foundation under Grant 1956032.

REFERENCES

[1] [n. d.]. Cachegrind: a cache and branch-prediction profiler. https://valgrind.org/docs/manual/cg-manual.html.
[2] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting Secret Keys Via Branch Prediction. In *Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4377)*, Masayuki Abe (Ed.). Springer, 225–242. https://doi.org/10.1007/11967668_15
[3] Jaeguk Ahn, Jiho Kim, Hans Kasan, Leila Delshadtehrani, Wonjun Song, Ajay Joshi, and John Kim. 2021. Network-on-chip microarchitecture-based covert channel in gpus. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 565–577.
[4] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 870–887. https://doi.org/10.1109/SP.2019.00066
[5] Daniel J Bernstein. 2005. Cache-timing attacks on AES. (2005).
[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.
[7] Marina Blanton, Mikhail J Atallah, Keith B Frikken, and Qutaibah Malluhi. 2012. Secure and efficient outsourcing of sequence comparisons. In *European Symposium on Research in Computer Security*. Springer, 505–522.
[8] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. 2013. Data-oblivious graph algorithms for secure computation and outsourcing. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. 207–218.
[9] 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 *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 715–733. https://doi.org/10.1145/3460120.3484583
[10] 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*. 174–189.
[11] Peter W. Deutsch, Yuheng Yang, Thomas Bourgeat, Jules Drean, Joel S. Emer, and Mengjia Yan. 2022. DAGguise: mitigating memory timing side channels. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 329–343. https://doi.org/10.1145/3503222.3507747
[12] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. 2021. Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 972–984.
[13] Andrew Ferraiuolo, Yao Wang, Danfeng Zhang, Andrew C Myers, and G Edward Suh. 2016. Lattice priority scheduling: Low-overhead timing-channel protection

for a shared memory controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 382–393.

[14] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2009. Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications. In *Information, Security and Cryptology - ICISC 2009, 12th International Conference, Seoul, Korea, December 2-4, 2009, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5984)*, Dong Hoon Lee and Seokhie Hong (Eds.). Springer, 176–192. https://doi.org/10.1007/978-3-642-14423-3_13

[15] Daniel Gruss, Clémentine Maurice, and Klaus Wagner. 2015. Flush+Flush: A Stealthier Last-Level Cache Attack. *CoRR* abs/1511.04594 (2015). arXiv:1511.04594 http://arxiv.org/abs/1511.04594

[16] Zhen Hang Jiang, Yunsi Fei, and David R. Kaeli. 2016. A complete key recovery timing attack on a GPU. In *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*. IEEE Computer Society, 394–405. https://doi.org/10.1109/HPCA.2016.7446081

[17] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. 2011. Timing-and termination-sensitive secure information flow: Exploring a new approach. In *2011 IEEE Symposium on Security and Privacy*. IEEE, 413–428.

[18] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, Tadayoshi Kohno (Ed.). USENIX Association, 189–204. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/kim

[19] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. 2009. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *15th International Conference on High-Performance Computer Architecture (HPCA-15 2009), 14-18 February 2009, Raleigh, North Carolina, USA*. IEEE Computer Society, 393–404. https://doi.org/10.1109/HPCA.2009.4798277

[20] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 75–86.

[21] Chang Liu, Austin Harris, Martin Maas, Michael W. Hicks, Mohit Tiwari, and Elaine Shi. 2015. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, Özcan Öztürk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, 87–101. https://doi.org/10.1145/2694344.2694385

[22] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos V. Rozas, Gernot Heiser, and Ruby B. Lee. 2016. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*. IEEE Computer Society, 406–418. https://doi.org/10.1109/HPCA.2016.7446082

[23] Fangfei Liu and Ruby B. Lee. 2014. Random Fill Cache Architecture. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*. IEEE Computer Society, 203–215. https://doi.org/10.1109/MICRO.2014.28

[24] 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, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 605–622. https://doi.org/10.1109/SP.2015.43

[25] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. PHANTOM: practical oblivious computation in a secure processor. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM, 311–324. https://doi.org/10.1145/2508859.2516692

[26] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations in SGX. In *Topics in Cryptology - CT-RSA 2018 - The Cryptographers' Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10808)*, Nigel P. Smart (Ed.). Springer, 21–44. https://doi.org/10.1007/978-3-319-76953-0_2

[27] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3860)*, David Pointcheval (Ed.). Springer, 1–20. https://doi.org/10.1007/11605805_1

[28] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. 2021. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 645–662. https://www.usenix.org/conference/usenixsecurity21/presentation/paccagnella

[29] Dan Page. 2005. Partitioned cache architecture as a side-channel defence mechanism. *Cryptology ePrint Archive* (2005).

[30] Colin Percival. 2005. Cache missing for fun and profit.

[31] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 565–581. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl

[32] Moinuddin K. Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. IEEE Computer Society, 775–787. https://doi.org/10.1109/MICRO.2018.00068

[33] Moinuddin K Qureshi. 2019. New attacks and defense for encrypted-address cache. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 360–371.

[34] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 431–446. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane

[35] Martin Schoeberl, Wolfgang Puffitsch, and Benedikt Huber. 2009. Towards time-predictable data caches for chip-multiprocessors. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer, 180–191.

[36] Ali Shafiee, Akhila Gundu, Manjunath Shevgoor, Rajeev Balasubramonian, and Mohit Tiwari. 2015. Avoiding information leakage in the memory controller with fixed service policies. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, Milos Prvulovic (Ed.). ACM, 89–101. https://doi.org/10.1145/2830772.2830795

[37] Luigi Soares and Fernando Magno Quintão Pereira. 2021. Memory-Safe Elimination of Side Channels. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 200–210. https://doi.org/10.1109/CGO51591.2021.9370305

[38] Wei Song, Boya Li, Zihan Xue, Zhenzhen Li, Wenhao Wang, and Peng Liu. 2021. Randomized last-level caches are still vulnerable to cache side-channel attacks! But we can fix it. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 955–969.

[39] 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. *Journal of the ACM (JACM)* 65, 4 (2018), 1–26.

[40] Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W Fletcher. 2021. Opening pandora's box: A systematic study of new ways microarchitecture can leak private data. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 347–360.

[41] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. 2022. MeshUp: Stateless cache side-channel attack on CPU mesh. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1506–1524.

[42] Yao Wang, Andrew Ferraiuolo, and G. Edward Suh. 2014. Timing channel protection for a shared memory controller. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*. IEEE Computer Society, 225–236. https://doi.org/10.1109/HPCA.2014.6835934

[43] Yao Wang and G Edward Suh. 2012. Efficient timing channel protection for on-chip networks. In *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*. IEEE, 142–151.

[44] Zhenghong Wang and Ruby B. Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, Dean M. Tullsen and Brad Calder (Eds.). ACM, 494–505. https://doi.org/10.1145/1250662.1250723

[45] Hassan M. G. Wassel, Ying Gao, Jason Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. 2013. SurfNoC: a low latency and provably non-interfering approach to secure networks-on-chip. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, Avi Mendelson (Ed.). ACM, 583–594. https://doi.org/10.1145/2485922.2485972

[46] 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 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 603–620. https://www.usenix.org/conference/usenixsecurity18/presentation/weiser

[47] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 15–26. https://doi.org/10.1145/3213846.3213851

[48] Qiumin Xu, Hoda Naghibijouybari, Shibo Wang, Nael B. Abu-Ghazaleh, and Murali Annavaram. 2019. GPUGuard: mitigating contention based side and covert channel attacks on GPUs. In *Proceedings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, June 26-28, 2019*, Rudolf Eigenmann, Chen Ding, and Sally A. McKee (Eds.). ACM, 497–509. https://doi.org/10.1145/3330345.3330389

[49] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 888–904.

[50] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B Lee, and Gernot Heiser. 2015. Mapping the Intel last-level cache. *Cryptology ePrint Archive* (2015).

[51] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2016. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. In *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9813)*, Benedikt Gierlichs and Axel Y. Poschmann (Eds.). Springer, 346–367. https://doi.org/10.1007/978-3-662-53140-2_17

[52] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W Fletcher. 2018. Data oblivious ISA extensions for side channel-resistant and high performance computing. *Cryptology ePrint Archive* (2018).

[53] Yanqi Zhou, Sameer Wagh, Prateek Mittal, and David Wentzlaff. 2017. Camouflage: Memory Traffic Shaping to Mitigate Timing Attacks. In *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*. IEEE Computer Society, 337–348. https://doi.org/10.1109/HPCA.2017.36