Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs

Adi Fuchs
Princeton Electrical Engineering Department
Princeton University, NJ 08544
adif@princeton.edu

Ruby B. Lee
Princeton Electrical Engineering Department
Princeton University, NJ 08544
rblee@princeton.edu

Abstract

Caches are integral parts in modern computers; they leverage the memory access patterns of a program to mitigate the gap between the fast processors and slow memory components.

Unfortunately, the behavior of caches can be exploited by attackers to infer the program's memory access patterns, by carrying out cache-based side-channel attacks, which can leak critical information.

Secure caches that were proposed employ cache partitioning or randomized memory-to-cache mapping techniques to prevent these attacks. Such techniques may add to the complexity of cache designs.

In this work, we suggest the use of specialized prefetching algorithms for the purpose of protecting from cache-based side-channel attacks. Our prefetchers can be combined with conventional set associative cache designs, are simple to employ, and require low incremental hardware overhead costs, if the base prefetching scheme is already employed.

We integrated our prefetching policies with commonly used GHB and stride prefetching schemes, and compared their performance with the standard implementations of those schemes, on both conventional and secure cache designs. More specifically, our results show that the use of our secure prefetching policy delivers original prefetching performance when integrated with a stride prefetcher. Finally, we demonstrate how a disruptive prefetching scheme can protect the cache from an access based side-channel attack.

Categories and Subject Descriptors B.3 [*Memory Structures*]: Cache memories

; D.2.8 [*Computer Systems Organization*]: General—Security and protection

SYSTOR '15, May 26–28, 2015, Haifa, Israel.

http://dx.doi.org/10.1145/2757667.2757672

Keywords Cache prefetching, Side-Channel Attacks, Secure hardware, Computer architecture. Microarchitecture

1. Introduction

On-chip memory caching solutions are ubiquitous, as they serve as an integral part of almost every computation-based system. Caches enable high application performance by storing a program's code or data likely to be accessed in the near future. Caching algorithms predict future data or code based on the program's behavior, and therefore couple the behavior of the caching module with the behavior of the program.

Unfortunately, by analyzing on-chip cache behavior, an attacker can exploit this behavioral coupling to carry out *cache-based side-channel attacks*, which leverage distinct patterns in the program's behavior to extract the victim's secrets.

Cache-based side-channel attacks (Page 2002; Bernstein 2005; Percival 2005; Osvik et al. 2006; Tromer et al. 2010; Bonneau and Mironov 2006; Yarom and Falkner 2013) typically combine the timing measurements collected by malicious code with the knowledge of the inner workings of cryptographic algorithms (e.g., AES (Daemen and Rijmen 1999)) to reveal the attacked program's sensitive data. In order for such attacks to be feasible, several conditions must be met: (i) an attacker should possess the knowledge of a victim program's cryptographic algorithm, which uses memory and caches (ii) the attacker should exploit the fact that cache designs tend to have fixed memory-to-cache mapping and behavior, and (iii) there is a significant difference between the cache hit and miss times. If these conditions are met, it is possible to conduct a cache-based attack using the victim program's data access sequence, extract secret information from the attacked program, such as a cryptographic key, and obtain sensitive information.

Secure cache solutions should ideally keep the program's secrets secured in the presence of side-channel attacks, and have a minimal impact on performance. Existing secure cache proposals often deflect cache-based attacks by preventing the reflection of cache-mapping between processes. These designs typically employ cache *partitioning* or cache

remapping, thus eliminating possible projections of the victim process' cache footprint to an attacking process' observations of the shared cache state, and more generally, preventing information leakage across different trust domains.

In this work, we investigate an approach using security-enhanced *cache prefetching* policies to protect from cache-based side-channel attacks. We leverage the traditional prefetcher purpose of boosting cache performance, and combine it with a memory-to-cache mapping aware scheme that can produce cache access patterns that are not useful to the attacker. Our prefetchers are designed to mitigate cache-based side-channel attacks, and thus serve as cheap and highly-performing alternatives to proposed secure cache designs. We suggest two new extensions to standard prefetching schemes: (i) a randomized prefetching policy, that varies in the level of aggressiveness and prefetch ordering, and (ii) a "set-balancer" extension, that attempts to balance the load induced by the prefetched lines across all cache sets.

We evaluate our prefetching policies with both stride(Fu et al. 1992) and GHB D/GC (Nesbit and Smith 2004) prefetchers, and integrate them with both a standard cache design and the security-aware Newcache (Wang and Lee 2008) design. The main contributions of this work are as follows:

- We suggest several cache prefetching policies that can obfuscate a victim's cache accesses from cache side-channel attackers. We integrate our policies with commonly used prefetching schemes and compare them with prefetchers connected to both a standard set-associative cache and a security-aware cache.
- We show when a cache prefetching policy is capable of disrupting a cache-based Prime+Probe side-channel attack (Osvik et al. 2006; Tromer et al. 2010) , and we also discuss the reasons and conditions where a prefetcher is incapable of disrupting such an attack.

The rest of this paper is organized as follows: Section 2 provides the background for the main issues dealt with in this work: (i) a brief survey of cache-based side-channel attacks, (ii) an overview of recent secure cache designs, and (iii) the role of prefetchers in a modern computer system, and brief description of the commonly used schemes we employed. Section 3 outlines the basic algorithms and motivation for our prefetcher policies. Section 4 outlines the performance experiments conducted for our randomized and set balanced prefetching policies integrated with two common prefetching schemes. Section 5 demonstrates how using our prefetching schemes we can defeat an access based "Prime+Probe" attack, and also discusses when it is impossible to do so. Section 6 summarizes this work, and possible future directions.

2. Background and Related Work

Today's personal computing based devices extend beyond traditional PCs. As cloud services, personalized content, and smartphones are becoming more ubiquitous, more hardware is used by commercial companies to store personal user information, increasing the motivation for attackers to conduct hardware-based thefts and gain access to sensitive information. There have been several studies, both commercial and non-commercial, on data-driven attacks, and ways to mitigate them by software or hardware. In this section, we focus on cache side-channel attacks, and security-aware cache designs, and present some of the recent work done in these fields.

When suggesting a new security-aware design, one must first establish the threat model the design wishes to protect from, and the Trusted Computing Base (TCB) (Lee 2013) which contains the system's trusted modules. The TCB provides the foundations for the capabilities that a secure design relies on.

In this work, our *threat model* is that of an attacker concurrently running a malicious process that carries out an attack by periodically polling a cache shared by both the victim and attacker processes. The secure design presented in this work relies on a data cache prefetcher connected to an L1 data cache, therefore, our TCB contains the processor core, including its Level 1 caches (i.e. L1 instruction and data caches) and the data cache prefetcher. We also assume that the Level 2 and higher caches, their controllers, and software memory management all behave correctly and are not under the control of an attacker. This assumption essentially means that we assume these hardware (and software) components are correctly implemented and have no malicious hardware (or software) embedded in them.

We discuss the various classes of cache-based side-channel attacks, focusing on the access-based "Prime+Probe" side-channel attack, demonstrated in this work as a case study. We then present some of the recent work on secure cache designs. Finally, we discuss how cache prefetchers achieve performance, focusing on two commonly used implementations, on which we based our prefetching scheme.

2.1 Side-Channel Attacks

Side-channel attacks exploit secrets that can be unintentionally leaked via shared resources; e.g., shared caches, shared physical cloud server (Wu et al. 2012; Ristenpart et al. 2009), shared memory and storage systems (Jana and Shmatikov 2012), etc. With the growing popularity of parallel computing, the ubiquity of shared resources is increasing.

Several studies have investigated side-channel attacks in various domains. Jana and Shmatikov (Jana and Shmatikov 2012) suggested a user-level attack that leverages the memory usage table in an operating system. Their attack attempts to capture the timely evolution of an application's memory

working set and infer its contents using a large database constructed from application footprints.

Percival suggested a cache-based side-channel attack on the RSA encryption in an SMT environment (Percival 2005). This attack repeatedly occupies the entire cache with the attacker's lines and reads the system timer to detect which lines are missing from the cache, and by that inferring some of the bits in the victim program's RSA key. Bernstein's attack on AES (Bernstein 2005) assumes the attacker has no knowledge of the victim's memory accesses. This attack also uses the timing disparity between a cache hit and a cache miss: the attacker attempts to encrypt a large number of plaintext blocks and measures the time taken for each block encryption. The Prime+Probe attack on AES (Osvik et al. 2006; Tromer et al. 2010) assumes a separate victim memory space, and, therefore, does not assume shared AES tables between the victim and the attacker. As demonstrated in this work, this attack occupies all cache lines, and uses cache mapping conflicts to conduct an attack to measure which cache sets are used by the victim. By doing this, it is possible to infer the memory address (and the table index) used by the victim, and thus the bits of the secret key used by the victim to access AES tables.

Cache based side-channel attacks are roughly divided into two classes:

- **Timing-based attacks**: in these attacks, the attacker exploits the difference in timing between cache hits and cache misses and measures the timing difference for a whole operation (e.g., block encryption) on each chunk of data. By knowing typical system times (for example, after constructing a large learning database extracted from a large possible number of cache states) the attacker can infer some or all of the encrypted data's key bits.
- **Access-based attacks**: in these attacks, the attacker and victim share the same cache, but not necessarily the same address space. The attacker exploits the fixed memory-to-cache-set mapping, and polls on the cache lines to measure typical cache access times, in order to find which sets the victim has used.

The *Prime+Probe* attack addressed by this work is an *access-based* side-channel attack, with which we target the AES encryption algorithm. In the attack, an attacker and a victim run concurrently on the same machine using the same cache. The attacker periodically fills the data cache with an array, and then periodically probes the cache by reading the array members, thus checking whether each cache line hits or misses. If the access time is high, this means that the victim program evicted the attacker's cache line. By gathering cache state samples, the attacker can post-process the gathered data to construct the access time of the victim's encryption process.

For example, as we demonstrate in Section 5.1, since AES-128 relies on factoring of 16 byte keys and 16 byte plain-text data blocks, it is possible to construct a database containing the typical cache set access times of all 256 possible values of $data_i$ given key_j for each $i, j \in (1..16)$, and compare the gathered cache probe times and extract the values of $data_i, key_j$ for the plain-text data and key bytes used by the victim. In the rest of this paper, we tend to focus on only "Prime+Probe" attacks on the data cache, and do not consider other kinds of cache side-channel attacks.

We will show that using our prefetching policies we were able to disrupt such an attack, and prevent an attacker from obtaining useful probe data.

2.2 Secure Cache Designs

Secure caches have been proposed to defend against cache side-channel attacks. They have used cache partitioning techniques or cache randomized-mapping techniques. Secure cache designs disrupt the inter-process observability of a system's cache; the goal is to prevent an attacking process from being able to manipulate the system in a way that will expose other processes' behavior by leveraging the cache's memory-to-set mapping. Among recently studied cache designs are:

- *The Partition-Locked Cache*: A locking-based cache design (PLCache) (Wang and Lee 2007) that prevents the eviction of the security-critical cache lines of the victim process by another process. It performs selective, dynamic partitioning by presenting two techniques for cache line locking: ISA-based and region-based. Both techniques require the involvement of the operating system to prevent other malicious processes from misusing the cache locking mechanism, e.g., a DoS attack (lock all lines so no other processes will be able to use them). PLCache, therefore, assumes the operating system is trusted.
- *The Random-Permutation Cache(RPCache)*: A remapping-based cache design (Wang and Lee 2007) that uses a per-domain permutation table for randomized cache set mapping. RPCache also employs a clever replacement technique which calculates an on-the-fly set remapping when a secure line is being evicted, in order to prevent collisions from leaking data that can expose remapping independent behavior. This approach requires the storing of a permutation table for each trust domain and adding the domain ID field to the cache tag.
- *The Newcache design*: Newcache (Wang and Lee 2008) is a different secure cache design based on randomized memory-to-cache mapping that appears to have the best performance and security of the proposed secure cache designs. Newcache adds a level of indirection for remapping memory to cache addresses, based on an inverse mapping table implemented as line number registers (LNregs). This is a fully-associative mapping from an ephemeral large direct-mapped cache, achieved by in-

creasing the number of bits in each LNreg. The purpose of the hybrid direct-mapped and fully-associative mapping of Newcache is to improve performance or power while achieving dynamic randomized memory-to-cacheset address mapping. The designers claim equivalent cache miss rates and cache access latencies compared to conventional set-associative L1 caches of the same size. We used the gem5 implementation of Newcache used in the secure cache evaluation conducted by Liu and Lee (Liu and Lee 2013). We integrated Newcache with some of the prefetchers suggested in this work, and its performance is evaluated in Section 4.2.

- *Random-Fill cache design*: This is the most recent implementation of a secure cache design (Liu and Lee 2014). Random fill cache was the first to integrate randomized fetching techniques for the purpose of security. Our work shows that by adding directed fetching techniques to randomized fetches, it is possible to integrate our policy with commercially used prefetchers and conventional set associative caches, while achieving performance as well as mitigating some cache-based side-channel attacks.

  Cache line protection proposals extend beyond the scope of secure cache designs; Kim et al. proposed StealthMem (Kim et al. 2012), a system-level cache locking scheme protecting from cross-VM side-channel attacks in a cloud environment. Zhang and Reiter proposed Dppel (Zhang and Reiter 2013), an OS-based technique that consists of a background kernel process, which intentionally evicts some of the cache lines used by the processes in a targeted VM.

While a correct implementation and correct usage of the secure cache designs mentioned above can eliminate the feasibility of cache side-channel attacks, they require one or more of the following: a new cache structure, a change in the processor's instruction set architecture, or reliance on the operating system to manage and constrain process behavior. In addition, for remapping-based cache designs, the effectiveness of cache prefetching mechanisms might be adversely affected, if the cache prefetcher is oblivious of the secure cache's remapping. By implementing sophisticated prefetching policies, we wish to avoid these issues, and achieve a scheme that has high performance, low incremental hardware overhead, and also improves security.

2.3 Cache Prefetchers

Cache prefetching is a technique that further ameliorates the effect of memory latency on performance, as provided by the cache hierarchy.

With the growth in the CPU-memory gap, commonly termed "the memory wall" (Wulf and McKee 1995), the potential loss of performance induced by the memory wall becomes more pronounced, and prefetchers become more common.

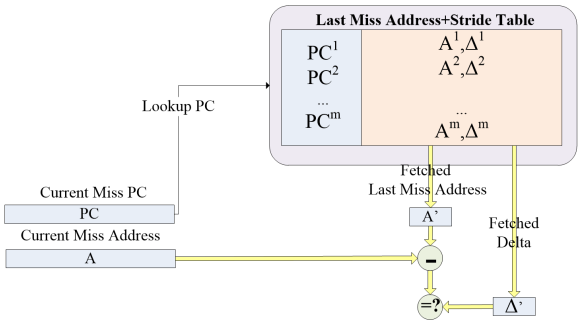

Figure 1. Stride Prefetching scheme

Prefetchers attempt to identify distinct program streams, and predict each stream's future memory accesses based on past stream characteristics. By eliminating some of the demand misses, or at least saving some of their average waiting time, the prefetcher reduces the overall number of cycles in which the processor waits for memory data to arrive.

We now briefly discuss the design and logical flow of the Stride(Fu et al. 1992) and GHB D/GC (Nesbit and Smith 2004) prefetching schemes upon which we build and evaluate our new security-enhanced prefetching policies in this work. We chose to evaluate these schemes since they are found in commonly used commercial processors (Doweck 2006; ARM 2010).

2.3.1 Stride Prefetcher

The Stride prefetching scheme records tuples of {PC, address} generated by cache misses, and attempts to compose streams of address deltas (or "strides") manifested from the same program location (identified by the PC). Figure 1 depicts a typical lookup and prediction process. Each stride table entry is indexed by the miss PC and contains a tuple of {last miss address, stride}. Let A be the current miss address A, and PC be the miss PC. If PC matches a stored PC', and the stride of the current and stored miss address A' matches the stored stride Δ', i.e. $\Delta = A - A'$ equals $\Delta' = A' - A''$ the stride prefetcher assumes steady state for the miss PC and predicts the next miss address for that PC will be $A + \Delta$.

2.3.2 GHB G/DC Pretecher

The global history buffer (GHB) with global delta correlation prefetching scheme tracks cache miss addresses, and records vectors of address deltas. Figure 2 depicts a general flow of the GHB G/DC prefetcher. Delta vectors are recorded into a GHB table with m entries. Each vector contains n deltas, divided into k "history deltas"; these deltas are used as index to the GHB entry. The entry's value is the sequence of the succeeding $(n - k)$ "future deltas". By capturing a recurring sequence of history deltas and fetching the subsequent sequence of predicted deltas, the GHB G/DC

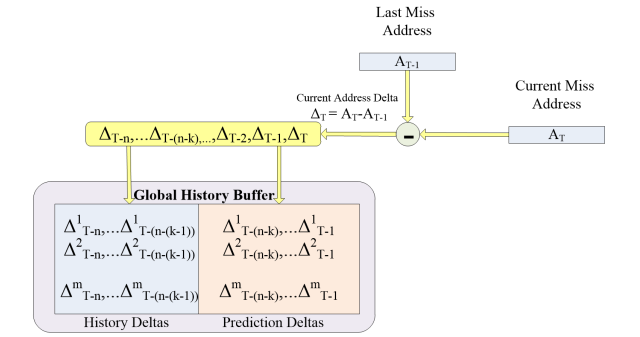

Figure 2. GHB G/DC Prefetching scheme

prefetcher attempts to correlate occurrences of a program's temporal-spatial streams.

3. Disruptive Prefetching

While prefetchers are traditionally used for performance purposes, we study whether in the context of cache-based side-channel attacks, can prefetchers disrupt the stability of cache occupancy, leveraged by side-channel attackers? Can we formulate a way to use prefetchers as means to improve security? Will it be possible to use our security aware prefetchers with a conventional, non-secure, cache design? If the answer is yes to these questions, our prefetching schemes might serve as a cheap, simple, and highly-performing alternative to sophisticated secure cache designs.

The key motivation for this disruptive use of prefetchers is the fact that a prefetcher disrupts a program's cache footprint, by scheduling memory accesses not yet requested by the program (and sometimes not requested at all). We can think of prefetching accesses as noise added to the original memory access sequence, and we wish to see what type of noise would be useful to mitigate access-based side-channel attacks.

It is important to note that standard prefetching schemes are not suitable as means for handling cache side-channel attacks, since altering the stability of cache line occupancy might not suffice to prevent side-channel attacks; if an attacker has prior knowledge of the cache mapping, and the prefetching policy is deterministic, it is possible for the attacker to model the prefetcher as well, and attempt to mimic the cache behavior in the presence of the prefetcher. In this section, we suggest ways to alter the prefetching policy, in order to make the cache behavior less predictable, while maintaining reasonable performance.

3.1 Prefetching Degree

One of the behavioral properties of a prefetcher is its aggressiveness, which can be quantified by the *prefetching degree* that is defined as follows: given a detected stream S of addresses: $\{A_{S,1},A_{S,2},...,A_{S,N}\}$ the degree D is the length of the stream's extrapolation, meaning the number of future addresses $\{A_{S,N+1},A_{S,N+2},...,A_{S,N+D}\}$ predicted and fetched by the prefetcher. A high degree, or aggressive prefetching, indicates high confidence in the stream prediction and can potentially deliver high performance if done in a timely manner. On the other hand, this means a higher degree of speculation. Therefore, if prediction is not accurate enough, it is possible that not all D addresses will be used, and fetching them can pollute the cache and harm performance. Prefetching degree, therefore, has an effect on performance, and should be carefully tuned.

3.2 Random Prefetching Policy

Algorithm 1 Random Prefetching Policy

function CACHE_MISS(ADDRESS ADDR, PC PC)
  $D \leftarrow rand(1, .., MAX\_DEGREE)$
  if ($\{ADDR, PC\}$ match stream S) **then**
    ;HIT: randomize default prefetcher policy
    $\{A_{S,N+1}, ..., A_{S,N+D}\} \leftarrow$ (predicted next D addresses)
  else
    ;MISS: randomize "next-line" fallback policy
    $\{A_{S,N+1}, ..., A_{S,N+D}\} \leftarrow$ (D cache lines addresses following ADDR)
    $pf\_addresses \leftarrow rand\_permutation\{A_{S,N+1}, ..., A_{S,N+D}\}$
  end if
  $T \leftarrow 0$
  for $pf\_addr \in pf\_addresses$ **do**
    mark pf_addr for prefetch at time T
    $T \leftarrow T + 1$
  end for
end function

In order to reduce the predictability of the cache access pattern, we wish to apply a randomized prefetching policy. Our policy works as follows: when a cache miss is detected, and the prefetcher can associate the miss with a matching stream, we randomize both the length of the extrapolated stream (i.e., the prefetching degree D), and the order in which the addresses $\{A_{S,N+1}, ..., A_{S,N+D}\}$ are queued for prefetching. In case the default prefetching policy fails to match a stream, we incorporate a fallback "next-line" policy that prefetches the D succeeding cache lines, i.e., $\{A_{S,N} + 1, ..., A_{S,N} + D\}$, in a randomized order. We do this to trigger accesses to adjacent cache sets, and prevent an attacker from being able to pinpoint the exact sets that are accessed by the victim program. The pseudo-code for our prefetching policy is outlined in Algorithm 1.

The prefetching degree has a high impact on prefetching policy. A high degree is likely to have inaccurate prefetching that could pollute the cache. Furthermore, when attempting to randomize the order in which an extrapolated stream is being prefetched, we affect the prefetcher's timeliness (i.e. scheduling the next predicted line after lines predicted for later use). These two key insights limit the maximal possible degree in which our prefetcher can deliver high performance. On the other hand, we wish to have a degree that is high enough to achieve a sufficient level of randomness, to make side-channel attacks infeasible. In our experiments, we found that a maximal degree of 10 serves as a good "sweet

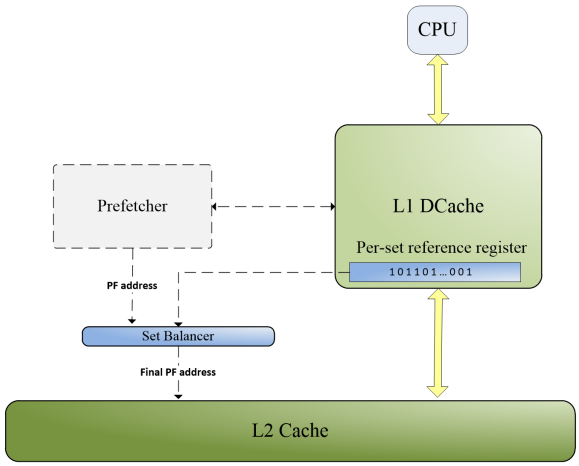

Figure 3. Cache Prefetcher with Set-Balancer Prefetching

spot", as it did not pollute the cache on most workloads tested, and made it harder to conduct side-channel attacks.

3.3 Set-Balancer Prefetching Policy

Algorithm 2 Set-Balancer Prefetching Policy

```
ref_set ← 0, 0, ..., 0
function SET_BALANCED(ADDRESS ADDR)
                        ;finds the address mapped to the closest un-accessed set
    ret_addr ← ADDR
    if ref_set = 1, 1, .., 1 then
        ref_set ← 0, 0, ..., 0
    end if
    orig_set ← addr.cache_set
    target_set ←{ closest s to orig_set with ref_set[s]=0 }
    ret_addr.cache_set ← target_set
    return ret_addr
end function
function CACHE_MISS(ADDRESS ADDR, PC PC)
    misses ← misses + 1
    if ({ADDR, PC} match stream S) then
        pf_addresses ← (predicted next D addresses)
        for pf_addr ∈ pf_addresses do
            final_pf_addr ← SET_BALANCED(pf_addr)
            mark final_pf_addr for prefetch at time T
            set ← final_pf_addr.cache_set
            ref_set[set] ← 1
            T ← T + 1
        end for
    end if
end function
function CACHE_HIT(ADDRESS ADDR, PC PC)
    set ← addr.cache_set
    ref_set[set] ← 1
    if misses mod 16 = 0 then
                        ;aggressive relaxation: PF on hit every 16th cache miss
        pf_addresses ← (D lines addresses following ADDR)
        for pf_addr ∈ pf_addresses do
            final_pf_addr ← SET_BALANCED(pf_addr)
            mark final_pf_addr for prefetch at time T
            set ← final_pf_addr.cache_set
            ref_set[set] ← 1
            T ← T + 1
        end for
    end if
end function
```

In order to understand the motivation behind a set-balancer policy, we must carefully examine the threat model of access-based side-channel attacks. In this model, a malicious program polls the cache lines while a victim program is concurrently executed.

These accesses provide the attacker with periodic snapshots of cache state, from which the victim's temporal cache access traces can be extracted. As conventional caches rely on fixed address-to-cache mapping, the distribution of cache sets accessed is connected to the victim's data structure accesses, which is often a function of the bits of cryptographic keys used to access a table, e.g., in the AES encryption process.

We wish to tackle this problem by suggesting a prefetching scheme that attempts to achieve a *uniform* cache set access pattern at any given time, since it is implausible to extract useful information from a uniform distribution. Our goal is to create a prefetcher that keeps track of currently accessed cache sets, and invokes prefetching requests to un-accessed sets. Our prefetcher employs this technique in order to distribute the program's memory accesses across all cache sets at any given time, so when a malicious process polls that cache, it will not be able to pinpoint specific cache sets that are occupied by the victim program. Our policy is integrated with basic prefetching schemes and alters an address targeted for prefetching to the address mapped to the nearest un-accessed cache-set. Our scheme relies on a register holding a a bit for each cache set, which is set upon reference to that set. If needed, our scheme changes an address marked for prefetching to an address mapped to the closest un-accessed set. Once all sets are accessed, all per-set bits are cleared.

Figure 3 and Algorithm 2 depict how a set-balancer can be integrated with a standard prefetching scheme. The per-set reference register (ref_set in Algorithm 2) is set upon cache access or when an address is marked for prefetching. Before an address can be marked for prefetching, in case it refers to an address already referenced (i.e. the corresponding bit is set in the reference register), the set-balancer attempts to find the closest address that is mapped to an unmapped set. By integrating this directed prefetching policy with an existing prefetcher scheme, we attempt to achieve an approximation for a uniform cache set occupancy at any given time. Since prefetching on every access is too aggressive, we've added an aggressiveness relaxation factor to prevent cache trashing; the on-hit next line prefetch policy is set only every 16th cache miss. This assures that too many lines will not be prefetched for the common case of low miss rate, but on the other hand it will be triggered by a high miss rate that is typical for access based attacks (in which the attacker evicts the victim's lines).

3.3.1 Set-Balancing Example

The best way to demonstrate the concept of set-balanced prefetching is by a simplified example. Assume we have a cache of 4 sets, with 16 byte lines. The cache is connected to a conservative (degree=1) delta prefetcher with

Memory Reference	Calculated PF Address	Reference Register	Final PF Address
LD 0x1020(set=2)	-	0,0,0,0→0,0,1,0	-
LD 0x1040(set=0)	-	0,0,1,0→1,0,1,0	-
LD 0x1060(set=2)	0x1080(set=0)	1,0,1,0	0x1090(set=1)
PF 0x1090(set=1)	-	1,0,1,0→1,1,1,0	-
LD 0x2020(set=2)	-	1,1,1,0	-
LD 0x2040(set=0)	-	1,1,1,0	-
LD 0x2060(set=2)	0x2080(set=0)	1,1,1,0	0x20B0(set=3)
PF 0x20B0(set=3)	-	1,1,1,0→0,0,0,0	-
LD 0x2070(set=3)	-	0,0,0,0→0,0,0,1	-
LD 0x2080(set=0)	-	0,0,0,1→1,0,0,1	-
LD 0x2090(set=1)	0x20A0(set=2)	1,0,0,1→1,1,0,1	0x20A0(set=2)
PF 0x20A0(set=2)	-	1,1,0,1→0,0,0,0	-

Table 1. Set-Balanced Prefetching Example

a set-balanced prefetching policy. Table 1 depicts a program's memory trace; each reference is a load instruction (LD) which is part of the program or a prefetch access (PF) invoked by the prefetcher. Each time the prefetcher calculates an address, the set balancer evaluates the reference register to determine which set has not been referenced yet; the invoked (final) prefetch address is mapped to the nearest un-accessed set.

We assume the cache is initially empty, so all accesses are misses. When encountering the load sequence of {0x1020,0x1040, 0x1060} the prefetcher calculates 2 consecutive 0x20 deltas, therefore predicts the next fetched address to be $0x1060 + 0x20 = 0x1080$, but since it is mapped to cache set #2 which is already marked as accessed by the reference register, the set-balancing policy changes it to the nearest un-accessed set, which is set #3, which results in 0x1090 being the final address to be prefetched. By employing this technique the set-balancer attempts to distribute all prefetched accesses in a uniform manner across all cache sets, using prefetch accesses to obfuscate the cache footprint produced by the program's memory access pattern. In this example address 0x1090 is fetched but not accessed by the program, causing cache pollution and a possible loss in performance. However, as we demonstrate in Section 4, the performance of a set-balanced prefetcher is comparable to the standard cache prefetchers.

4. Performance Evaluation

4.1 Methodology

For the purpose of evaluation, the gem5 simulator (Binkert et al. 2011) was used. We simulate a single Out-Of-Order microprocessor connected to a 2-level cache hierarchy over a 2 GB physical memory. Figure 4 shows a top-level block diagram of the simulated system. To test our randomized-prefetching policy we applied it to two commonly used cache prefetching schemes: a stride prefetcher (Fu et al. 1992), and a GHB-based prefetcher with global delta-correlation (GHB G/DC (Nesbit and Smith 2004)). Table 2 summarizes the simulation parameters used.

We ran 17 different workloads taken from the SPEC CPU2006 benchmark suite (Standard Performance Evaluation Corporation 2006). For the purpose of steady state measurements, for each benchmark the simulator skipped a given number of instructions to hit the memory intensive region of the benchmark. From within the memory intensive regions, the actual simulation recorded the first 500 Million instructions. We used Jaleel's SPEC workload analysis (Jaleel 2009) to roughly determine the number of instructions to skip for each benchmark.

Parameter	*Value*
General	
Instruction Set Architecture	amd64 (x64)
CPU clock rate	2.0 GHz
CPU configuration	8-wide OOO
L1 DCache size	32KB
L1 DCache line size	64B
L1 DCache associativity	4 way LRU
L1 DCache hit latency	4 cycles (2 tag+2 data)
L1 ICache size	32KB
L1 ICache line size	64B
L1 ICache associativity	2 way LRU
L1 ICache hit latency	4 cycles (2 tag+2 data)
L2 cache size	2MB
L2 cache line size	64B
L2 cache hit latency	20 cycles
Physical memory size	2048MB
Physical memory latency	93 cycles
Stride Prefetcher (if applicable)	
Table entries	256 fully associative
GHB G/DC Prefetcher (if applicable)	
Table entries	256 fully associative
History Length	3 Last miss address deltas

Table 2. Simulation Parameters

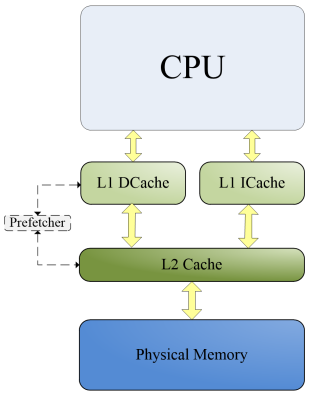

Figure 4. Top-level system diagram

4.2 Performance Results

We will now show the performance results of a randomized prefetcher and a randomized prefetcher with set-balancing. To evaluate the randomized prefetching and randomized set-balanced schemes, we implemented prefetching schemes against three different configurations.

- Base configuration - a standard L1 data-cache that does not possess any security-aware mechanism, and it is not connected to any prefetcher

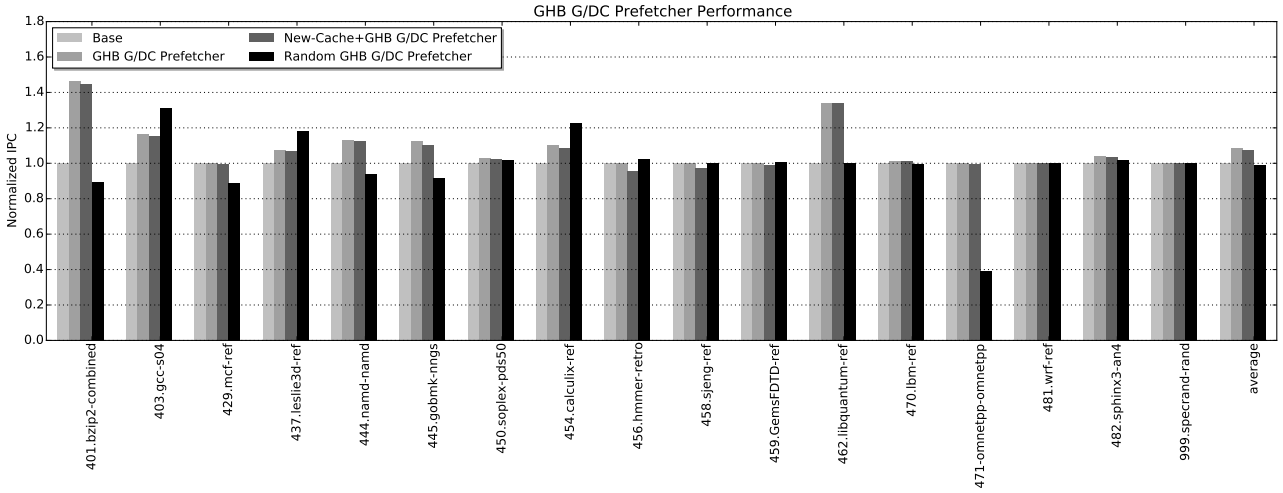

Figure 5. GHB-Based Randomized Prefetcher Performance

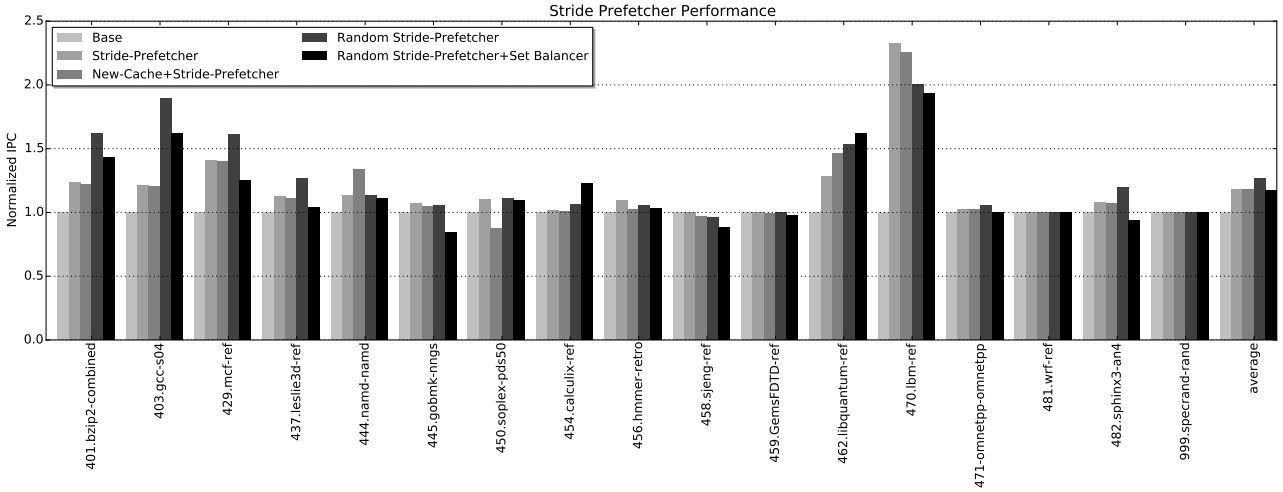

Figure 6. Stride-Based Randomized Prefetcher Performance

- Standard Prefetcher - a standard L1 data-cache connected to the basic configuration of a cache prefetching scheme (either GHB G/DC or Stride).
- Newcache + Prefetcher - the secured cache design suggested by Wang and Lee (Wang and Lee 2008), connected to a GHB/stride prefetching scheme.

Figure 5 depicts the performance results achieved for GHB prefetcher simulations, and Figure 6 shows the results achieved for stride prefetcher simulations. IPC was normalized to the base configuration's IPC.

The purpose of the fallback policy was not to gain performance, but to maintain a high prefetching aggressiveness, in case the default prefetching policy does not succeed in matching. We saw that the performance difference when not employing a fallback "next-line" policy was negligible, e.g., a randomized stride prefetcher without the fallback policy achieved an average speedup of 26% instead of a speedup of 27% gained by a stride prefetcher with a fallback policy. We did not show the results for this policy for brevity reasons.

4.2.1 Discussion

We see from the performance results that while a randomized GHB G/DC prefetcher (Figure 5) often performed poorly and achieved an average slowdown of 1% for all workloads tested, a randomized stride prefetcher achieved the best performance for 7 of the 17 workloads tested and achieved an average speedup of 27%, compared to the 18%

achieved by the second-best performing, standard stride prefetcher.

This indicates that the randomized GHB G/DC pollutes the cache with unused lines, due to the noisy miss pattern induced by the randomization.

There are two main reasons for which randomization worked better for a stride prefetcher: (i) **Localized stream construction:** since a stride prefetcher is PC-directed, stream randomization was employed *locally and separately* for each PC. When stream randomization is applied for global memory stream correlation, as in the case of GHB G/DC, it creates a noisy pattern that disrupts the correlation process, and induces global miss patterns that are translated to unstable delta vectors recorded under the same history deltas. A stride prefetcher, on the other hand, attempts to extrapolate memory streams locally by PC-based partitioning. Adding randomization to the process, introduces locally employed noise, that is corrected and regenerated, periodically, for long memory streams. The superior performance of the randomized stride prefetcher was achieved due to the aggressive prefetching of the randomized policy. (ii) **Faster stream detection:** as outlined in Figures 1 and 2, both prefetchers use recorded history tables to detect memory streams. However, the difference is that a GHB G/DC relies on a sequence of several history deltas, and as mentioned in Table 2, our simulation uses 3 history delta sequences as index. The stride prefetcher, on the other hand, matches a current delta with a single history delta, enabling easier and faster detection of a stabilized, non-noisy, stream.

We also see that in both GHB and Stride prefetcher schemes the integration with Newcache achieves a speedup comparable to the speedup achieved with the conventional set-associative cache. This is interesting and unexpected, showing that Newcache does not degrade the performance of a standard prefetcher, as we originally feared. Newcache achieves the same speedup of 18% for the standard stride prefetcher. We initially thought that the performance of Newcache with a prefetcher would be hurt by the randomized remapping caused by Newcache replacement. Surprisingly, this rarely happened and is noticeable only in one case - when running 'soplex-pds50' for the stride prefetcher. In fact, in two cases (for 'namd-namd' and 'libquantum'), the stride prefetcher with Newcache performed better than the stride prefetcher with the conventional set-associative cache.

We conclude that it is both the localized stream partitioning that constrains randomized prefetching noise, and faster detection by short histories, which make a stride prefetcher more suitable for a randomized prefetching policy.

Finally, we see that a randomized stride prefetcher policy with set-balancing achieves a speedup of 18%, comparable to the standard prefetcher configuration, but less than the speedup achieved by the randomized stride prefetcher (without set-balancing). The reason is the added prefetching aggressiveness that causes some amount of cache pollution by prefetched lines that are not later accessed by the program. Clear evidence of that are 'gobmk', 'sjeng', and 'sphinx'benchmarks in which the randomized prefetcher with set-balancer achieves worse performance than that of the "Base" configuration. This clearly indicates that the prefetcher is inaccurate and has an adverse effect on performance. An interesting case is the 'libquantum' benchmark which is the only benchmark for which the the addition of the Set-balancing policy achieves the best performance. This may be because the set-balancing policy prefetches future accessed lines by accessing neighboring lines that have not been accessed recently, when trying to achieve a uniform access distribution over the cache sets.

Although, as anticipated, the set-balancer did not achieve a performance gain comparable to the randomized prefetching scheme on which it was built, it did gain performance comparable to the standard configuration. We show next the important role of the set-balancer based prefetcher in mitigating cache side-channel attacks.

Algorithm 3 Prime+Probe Attack

```
initialize cache_occupying_array Arr
byte chunk[16]
k ← 128 bit AES key
f ← open(file_to_cipher)
repeat
                                        ;Attacker code: Prime phase
    for set ∈ (0..cache_sets − 1) do
        index ← set × cache_line_size
        for i ∈ (0..num_ways − 1) do
            Read Arr[index+cache_way_size × i]
        end for
    end for
                                        ;Victim code: AES Encryption
    chunk ← read(f, 16_bytes)
    AES_encrypt(chunk,k)
                                        ;Attacker code: Probe phase
    for set ∈ (0..cache_sets − 1) do
        for i ∈ (0..num_ways − 1) do
            index ← set × cache_line_size
            start_time ← time()
            load Arr[index+cache_way_size × i]
            end_time ← time()
            record (end_time − start_time)
        end for
    end for
until EOF(f)
```

5. Security Evaluation

In this section, we evaluate how our new prefetcher policies deal with a real side-channel attack, determine whether prefetchers can be used to defeat a side-channel attack, and under what conditions.

5.1 Case Study: Prime+Probe Attack

To demonstrate a prefetcher's ability to mitigate cache-based side-channel attacks, we ran a synthetic code implementing the *"Prime+Probe"* cache-based side-channel attack (Osvik et al. 2006; Tromer et al. 2010) on an application that encrypts a file via AES encryption.

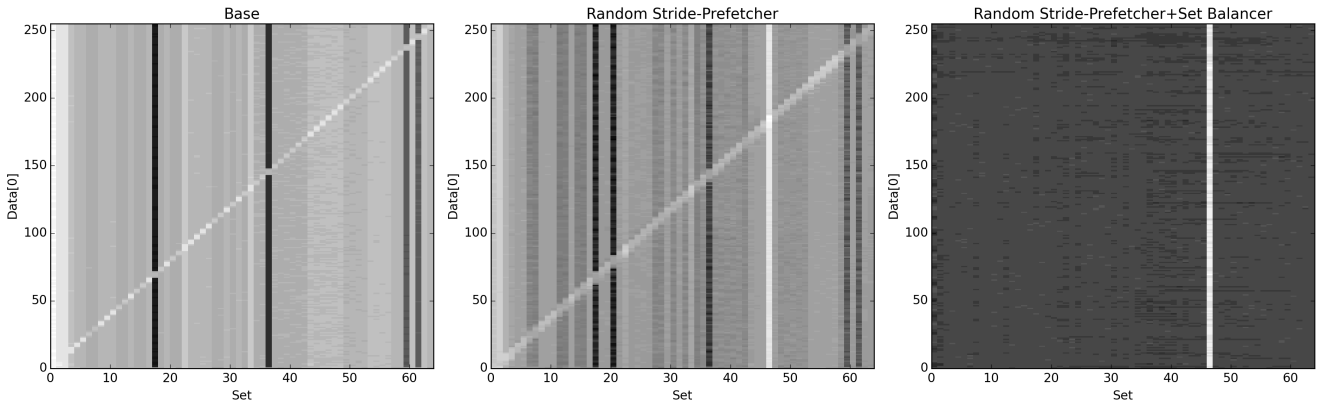

Figure 7. Prime+Probe attack cache set access times

In a Prime+Probe attack, the attacker does not have access to the victim application's memory space, and therefore attempts to construct the cache footprint by tracking the "hot" cache sets used by the victim, as they indicate which AES table entries are accessed using the outcome of the victim's plaintext byte XOR'ed with an AES key byte (round 1 AES attack). The attacker polls the cache by repeatedly executing the attack code (i.e. the cache fill and cache probe phases) while the victim application (that executes AES encryption) is concurrently executed. During the online phase of the attack, many samples of the encryption time of a plaintext block are taken. Then, in the offline phase of the attack, a "heat map" is constructed, giving the encryption times for each (plaintext byte value, cache set number), for each of the 16 key bytes. Figure 7 shows the heat map for one key byte, where a lighter spot indicates a longer execution time. Algorithm 3 depicts the pseudo-code for the Prime+Probe attack we used in this work. In the prime phase, the attacker repeatedly loads each cache line with elements read from an array, Arr. In the probe phase the attacker samples the access times for all cache sets, and primes the cache for the next timing sample.

Our code simulates a sequential execution of both the attacker and the victim program, further favouring the attacker. We therefore mimic a sophisticated form of attack in which the attacker can achieve absolute scheduler priority, possibly by timely gaming the scheduler, a technique demonstrated by Ristenpart et al. (Ristenpart et al. 2009). For the purpose of producing a steady state cache footprint, we performed an encryption for a file of 750KB, which produced $\sim$48000 rounds of encryption. In our model after each round the attacker samples the cache state by using the x86's *"rdtsc"* time counter sampling instructions(Paoloni 2010).

We ran our attack on gem5 simulating a system with a 4-way set-associative 16KB L1 Dcache with LRU replacement policy. We tested the results obtained for three different prefetching configurations:

- *base*: the basic cache configuration, not integrated with any prefetching mechanisms.
- *random stride prefetcher*: the basic cache connected to a randomized stride prefetcher.
- *set-balanced random stride prefetcher*: the basic cache connected to a randomized stride prefetcher, with the set balancer prefetching extension.

For brevity purposes we do not show the results of a Newcache design, but a previous study (Liu and Lee 2013) demonstrated how Newcache defeats the cache side-channel attack and produces a noisy pattern for this attack.

The attack presented in Figure 7 shows heat maps containing the average set access times for every possible value of the first encrypted plain-text byte (i.e. data[0]), using an all-zero key.

5.1.1 Discussion

Figure 7 depicts a heat map that contains the distribution of cache set timing with respect to the value of the first plaintext byte. This heat map is an example of an outcome from the Prime+Probe access based timing samples.

In the heatmap produced by the "base" configuration we see some cache sets are more frequently accessed than others; more importantly, we see a diagonal line that can couple the plaintext byte with the cache set. This is the result of parts of the AES encryption data flow being governed by values of the key and the plaintext byte to be encrypted. Both prefetching schemes produce a vertical line at set 46, due to a simulation bias caused by virtual memory mapping and the prefetcher's implementation.

The heatmap shown for the "random stride prefetcher" reveals an interesting insight. While one might have expected a randomized prefetcher to produce a noisy heat map, and although some of the distribution was altered, we still clearly see the diagonal line. The reason for this result is that after many samples, the noise produced by the randomized

prefetcher cannot hide the actual cache access pattern induced by the AES encryption process.

We conclude from a failure of a randomized prefetcher to defeat this attack, that random noise induced by prefetching cannot defeat a well-timed side-channel attack that obtains many samples. This insight motivated our Set-balancer scheme described in Algorithm 2. The set-balancer prefetcher is both aggressive enough to hide the actual access pattern, and targets an even distribution of all memory accesses among cache-sets. We can see that, in fact, the heat map gathered by the attack on a "Random stride prefetcher+Set Balancer" configuration that combines a random prefetcher with a set-balancer extension is too noisy for one to infer from.

Since the prefetcher targets only the L1 D-cache, our security-enhanced prefetching techniques would only apply to attacks on the L1 D-cache, and not to the L1 I-cache, the L2 cache or the L3 cache (Last Level cache in servers).

Also, we do not know if our prefetching techniques can mitigate other side-channel attacks (not Prime and Probe) on the L1 D-cache. This can be future work.

6. Conclusions And Future Work

In this work, we explored the security, performance, and design issues due to cache-based side-channel attacks. We studied current solutions and suggested a new approach leveraging randomized and set-balanced prefetchers as attack-disruptive prefetching policies that do not hurt the performance of a prefetcher, without having to change conventional set-associative cache designs.

We presented a brief survey of side-channel attack classes that exploit the access timing disparity of cache hits and misses, and the static memory-to-cache mapping to leak sensitive information. We then presented the recent secure cache designs that deal with these attacks, discussed their main features, and questioned whether it might be possible to achieve similar features by using a smart prefetching scheme integrated with a conventional set-associative cache.

We described the traditional role of prefetchers in modern computer systems and presented the two basic prefetching schemes evaluated in this work. We then suggested the randomized and set-balanced prefetching policies, as alternatives to secure cache designs. Our approach does not require any architectural changes and relies on simple data-structures.

In our performance study, we saw that the integration of Newcache with a prefetcher achieves a speedup comparable to that of the prefetcher connected to a conventional set-associative cache. We showed that for some cases, the aggressiveness of our new randomized and set-balanced prefetchers performs better than the standard, more-conservative, prefetching scheme. We also saw that the randomized prefetching policy integrates better with a simple PC-based stride prefetcher scheme than with a global delta-correlated scheme, due to both the use of a PC to partition different streams and the use of short stride histories for faster stream learning. We, therefore, conclude that the robustness of a stride prefetcher makes it fit for a randomized prefetching policy.

This work also confirms that Newcache has high performance, and even has the same performance improvement from both stride and global history prefetchers as do conventional caches. A detailed comparison of the hardware complexity of Newcache versus security-enhanced prefetchers could also enlighten the discussion on how best to defeat cache side-channel attacks.

Finally, we demonstrated the ability of a randomized and set-balanced stride prefetcher to disrupt the cache footprint constructed by a Prime+Probe attack, thus creating a noisy, hard-to-classify footprint that does not leak any significant information from the original program running on an insecure cache.

We therefore showed that it is possible to use a prefetcher for the purposes of security and defeat a Prime+Probe side-channel attack, while keeping the prefetcher's original potential performance gain.

This work can set the path for many future studies, on how prefetchers can secure other aspects of computer systems prone to side-channel attacks. It would be interesting to integrate our schemes with secure cache designs and also to evaluate how and whether set-balancing can affect other applications of computer architectures.

7. Acknowledgements

We thank Fangfei Liu for providing us with valuable input on secure cache designs and side-channel attacks; this input greatly helped us leverage the security aspects of this work.

References

ARM (2010). Cortex-A8 Technical Reference Manual. `http://infocenter.arm.com/help/topic/com.arm.doc.subset.cortexa.a8/`.

Bernstein, D. J. (2005). Cache-timing attacks on aes.

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. (2011). The gem5 simulator. *Computer Architecture News*, 39(2):1–7.

Bonneau, J. and Mironov, I. (2006). Cache-Collision Timing Attacks Against AES. *Cryptographic Hardware and Embedded Systems*, 8:201–215.

Daemen, J. and Rijmen, V. (1999). Aes proposal: Rijndael. NIST Web page.

Doweck, J. (2006). White paper: Inside microarchitecture and smart memory access.

Fu, J. W. C., Patel, J. H., and Janssens, B. L. (1992). Stride directed prefetching in scalar processors. In *Intl. Symp. on Microarchitecture (MICRO)*.

Jaleel, A. (2009). Memory characterization of workloads using instrumentation-driven simulation.

Jana, S. and Shmatikov, V. (2012). Memento: Learning secrets from process footprints. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 143–157. IEEE.

Kim, T., Peinado, M., and Mainar-Ruiz, G. (2012). Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security symposium*, pages 189–204.

Lee, R. B. (2013). Security basics for computer architects. *Synthesis Lectures on Computer Architecture*, 8(4):1–111.

Liu, F. and Lee, R. (2014). Random fill cache architecture. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 203–215.

Liu, F. and Lee, R. B. (2013). Security testing of a secure cache design. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 3:1–3:8, New York, NY, USA. ACM.

Nesbit, K. J. and Smith, J. E. (2004). Data cache prefetching using a global history buffer. In *Symp. on High-Performance Computer Architecture (HPCA)*.

Osvik, D. A., Shamir, A., and Tromer, E. (2006). Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology–CT-RSA 2006*, pages 1–20. Springer.

Page, D. (2002). Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002:169.

Paoloni, G. (2010). How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. `http://download.intel.com/embedded/software/IA/324264.pdf`. [Online; accessed 3-March-2013].

Percival, C. (2005). Cache missing for fun and profit.

Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. (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*, pages 199–212. ACM.

Standard Performance Evaluation Corporation (2006). SPEC CPU2006. http://www.spec.org/cpu2006.

Tromer, E., Osvik, D. A., and Shamir, A. (2010). Efficient cache attacks on aes, and countermeasures. *J. Cryptol.*, 23(2):37–71.

Wang, Z. and Lee, R. B. (2007). New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 494–505. ACM.

Wang, Z. and Lee, R. B. (2008). A novel cache architecture with enhanced performance and security. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 83–93. IEEE.

Wu, Z., Xu, Z., and Wang, H. (2012). Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security Symposium*, pages 159–173.

Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24.

Yarom, Y. and Falkner, K. E. (2013). Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. *IACR Cryptology ePrint Archive*, 2013:448.

Zhang, Y. and Reiter, M. K. (2013). Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 827–838. ACM.