2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

Random Fill Cache Architecture

Fangfei Liu and Ruby B. Lee
Princeton Architecture Laboratory for Multimedia and Security (PALMS)
Department of Electrical Engineering, Princeton University
Princeton, NJ 08544, USA
{*fangfeil, rblee*}*@princeton.edu*

***Abstract*—Correctly functioning caches have been shown to leak critical secrets like encryption keys, through various types of cache side-channel attacks. This nullifies the security provided by strong encryption and allows confidentiality breaches, impersonation attacks and fake services. Hence, future cache designs must consider security, ideally without degrading performance and power efficiency. We introduce a new classification of cache side channel attacks: contention based attacks and reuse based attacks. Previous secure cache designs target only contention based attacks, and we show that they cannot defend against reuse based attacks. We show the surprising insight that the fundamental demand fetch policy of a cache is a security vulnerability that causes the success of reuse based attacks. We propose a novel random fill cache architecture that replaces demand fetch with random cache fill within a configurable neighborhood window. We show that our random fill cache does not degrade performance, and in fact, improves the performance for some types of applications. We also show that it provides information-theoretic security against reuse based attacks.**

***Keywords*-cache; security; side channel attacks; cache collision attacks; secure caches; computer architecture.**

I. Introduction

Recent findings on cache side channel attacks [1]–[7] have shown that correctly functioning caches may leak critical secrets like cryptographic keys, nullifying any protection provided by strong cryptography. These attacks are easy to perform and are effective on all platforms, from embedded systems to cloud servers, that use hardware caches. Therefore, future cache designs must take into account security, ideally without degrading performance and power efficiency.

In cache side channel attacks, an attacker exploits the large timing difference between cache hits and cache misses to infer the key-dependent (i.e., security-critical) memory addresses, and hence the secret information, during the execution of cryptographic programs. We introduce a new classification of cache side channel attacks, depending on how the attacker infers memory addresses: contention based attacks versus reuse based attacks. In contention based attacks [2], [3], [5], the key-dependent memory accesses may contend for the same cache set with the attacker's memory accesses, and result in eviction of one by the other, in a *deterministic* way. This enables the attacker to infer the memory address according to which cache set it maps to. In contrast, the reuse based attacks [4], [6], [8], [9] do not rely on any resource contention. Instead, they only exploit the reuse of a previously accessed (and cached) security-critical data to correlate the addresses of two memory accesses. We point out that reuse of the cached data is exactly the purpose of a cache, therefore reuse based attacks strike at the heart of a cache and are much harder to defend against.

Several recent work [8], [10]–[14] investigated how to design secure caches to provide built-in defenses against cache side channel attacks. Wang and Lee proposed two general design approaches [11]: the partition-based approach [8], [11], [13], [14] that eliminates the cache contention, and the randomization-based approach [10]–[12] that randomizes the cache contention. However, these approaches only target contention based attacks and are not effective in defeating reuse based attacks. There are also some efforts that try to achieve constant execution time by either not loading security-critical data into the cache at all, or trying to ensure all cache hits whenever security-critical data is accessed, by frequently preloading or reloading all security-critical data [8], [14], [15]. This approach may potentially defeat the reuse based attacks, but at the cost of significant performance degradation, and sometimes enabling other types of attacks.

In this paper, we try to find a general approach against reuse based attacks, as a complement to existing secure cache design approaches. We show that, contrary to conventional wisdom, constant execution time is not the necessary condition to defeat reuse based attacks. Surprisingly, we find that the fundamental *demand fetch* policy of a cache is a security vulnerability that causes the success of reuse based attacks. With the demand fetch policy, the cache fill is always correlated with a demand memory access, hence the state of a cache reveals information about previous memory accesses. Hence, we propose a general approach against reuse based attacks: re-design the cache fill strategy so that it is de-correlated with the demand memory access. We propose a novel random fill cache architecture with a new security-aware cache fill strategy. The random fill cache architecture takes advantage of the random access pattern found in cryptographic algorithms. Hence, it does not degrade performance. In fact, it is more general and flexible than the demand fetch strategy, and even enables performance improvements for some types of applications. Our main contributions are:

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.28

IEEE computer society

- A new classification of cache side channel attacks as contention based and reuse based attacks,
- A new general approach for securing caches against reuse based attacks: the cache fill strategy must be re-designed to de-correlate the cache fill and demand memory accesses,
- A novel random fill cache architecture with a flexible cache fill strategy, which replaces the demand fetch with random cache fill within a configurable neighborhood window,
- An information-theoretic proof of the security provided by our random fill cache architecture,
- Performance evaluation of the proposed cache architecture and study of the broader performance implications of the random cache fill strategy to programs that are not demand-fetch amenable.

The rest of the paper is organized as follows: Section II gives some background on cache side channel attacks and section III discusses past work. We introduce our new random fill cache architecture in section IV. We provide an information-theoretic proof of the security provided by our random fill cache architecture in section V. We evaluate the performance of our cache architecture in section VI. We discuss the broader performance implications of our cache architecture in section VII. We compare security and performance with past work in section VIII and conclude in section IX.

II. Background

A. Overview of Cache Side Channel Attacks

The majority of cache side channel attacks exploit the interaction of the **key-dependent data flow** in a program with the underlying cache (mostly L1 data cache) to learn the secret information. We primarily consider the information flow in which the secret information is directly modulated onto the memory address, in the form of key-dependent table lookups. This is commonly found in the software implementation of cryptographic algorithms. For example, the substitution box (S-box) in the block ciphers (e.g., Data Encryption Standard (DES), Advanced Encryption Standard (AES), Blowfish), and the multipliers table in the public-key algorithms (e.g., RSA) are all implemented as lookup tables indexed by a linear function of the secret key. The attacker is an unprivileged user-level process that aims to infer the key-dependent memory addresses, indirectly through the cache behavior.

B. Classification of Cache Side Channel Attacks

Table I summarizes the classification of all known cache side channel attacks. Cache side channel attacks have been conventionally classified as access-driven attacks and timing-driven attacks [1], based on what can be measured by the attacker. In the access-driven attacks, the attacker can observe which cache lines the victim has accessed by measuring the impact of the victim's cache accesses to the attacker's own accesses. In the timing-driven attacks, the attacker can measure the execution time of the victim process. However, this classification is not helpful in identifying root causes and potential countermeasures. We introduce a new classification: contention based attacks and reuse based attacks, based on how the attacker infers the memory address.

Table I
CLASSIFICATION OF CACHE SIDE CHANNEL ATTACKS

	Contention based Attacks	**Reuse based Attacks**
Access-driven Attacks	Prime-Probe Attacks	Flush-Reload Attacks
Timing-driven Attacks	Evict-Time Attacks	Cache collision Attacks

1) Contention based Attacks: The attacker may contend for the same cache set with the victim process and the contention results in eviction of one's cache line by the other. If the contention and eviction is *deterministic*, the attacker can infer the memory address of the victim according to which cache set it maps to. Figure 1 illustrates how this works. There are two variations of contention based attacks:

Prime-Probe Attack [3], [5]: The attacker repeats the following operations: 1) Prime: the attacker fills one or more cache sets with his own data. 2) Idle: the attacker waits for a pre-specified Prime-Probe interval while the victim process is running and utilizing the cache. 3) Probe: the attacker process runs again and measures the time to load each set of his data. The Probe phase primes the cache for subsequent observations. If the victim process uses some cache sets during the Prime-Probe interval, some of the attacker's cache lines in these cache sets will be evicted, which causes cache misses and thus a longer load time during the Probe phase.

Evict-Time Attack [5]: The attacker repeats the following operations: 1) Evict: the attacker fills one specific cache set with his own data and hence evicts the victim's data in that cache set. 2) Time: the attacker triggers the victim process to perform a cryptographic operation, and measures the total execution time. If the victim accesses the evicted data, his execution time tends to be statistically higher, due to the victim having a cache miss.

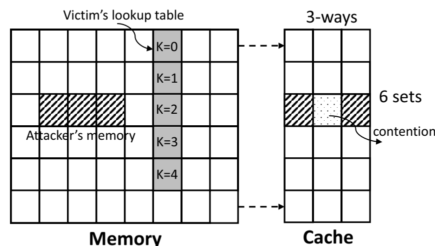

Figure 1. How cache contention can be used to infer information. Each square represents a cache line. Each memory line of the victim's lookup table is indexed by a different value of key bits K. The attacker occupies one cache set. The contention with the victim tells the attacker that the victim process has accessed the memory line corresponding to $K = 2$.

2) Reuse based Attacks: Contention based attacks rely on cache contention to learn where a memory line is placed in the cache. However, *reuse based attacks do not care about the location of a memory line in the cache. Instead, they only rely on the fact that a previously accessed data will be cached, hence the reuse of the same data by a later memory access will result in a cache hit.* There are two variations of reuse based attacks:

Flush-Reload Attack [6]: The attacker and the victim process may share some address space. In particular, security-critical data such as the lookup tables can be shared through a shared library. The attacker repeats the following operations: 1) Flush: the attacker flushes the security-critical data out of the cache to eliminate the impact of any previous accesses to the security-critical data. 2) Idle: the attacker waits for a pre-specified Flush-Reload interval while the victim process is running and utilizing the cache. 3) Reload: the attacker process runs again and measures the time to reload the security-critical data. If the victim process accesses some security-critical data during the Flush-Reload interval, the attacker will get a significantly lower reload time, since it will hit in the cache.

Cache Collision Attack [4]: The Flush-Reload Attack exploits reuse of the shared data between the victim and the attacker. However, sharing of security-critical data is not common and can easily be disabled by not declaring the security-critical data as read-only. The more serious threat of reuse based attacks is to exploit the reuse of the cached security-critical data *within* the victim process, which represents the intrinsic information leakage of a program. The reuse of data is commonly called *cache collision*, meaning that two memory accesses reference the same memory line. The basic idea of a cache collision attack is to exploit the impact of cache collision on the victim's aggregated execution time.

Premise of Cache Collision Attacks: Consider a series of security-critical accesses to lookup table T:

$$...T[\bullet], T[\bullet], T[x_i], T[\bullet], T[\bullet], T[x_j], T[\bullet], T[\bullet]... \qquad (1)$$

Denote the memory block address of x as $\langle x \rangle$. For any pair of accesses to x_i and x_j, when $\langle x_i \rangle = \langle x_j \rangle$, access to x_j will hit in the cache, whereas when $\langle x_i \rangle \neq \langle x_j \rangle$, access to x_j may or may not hit in the cache, depending on the rest of the sequence and prior cache contents. Statistically, the execution time of the cryptographic operation when $\langle x_i \rangle = \langle x_j \rangle$ will be less than that when $\langle x_i \rangle \neq \langle x_j \rangle$. Assume x_i and x_j both depend on the key K, where $x_i = f(K)$, and $x_j = g(K)$. If the attacker learns $\langle x_i \rangle = \langle x_j \rangle$ from the timing measurement, he can establish a relationship for the secret key as $\langle f(K) \rangle = \langle g(K) \rangle$.

3) Discussion: As pointed out by Wang and Lee [11], the root cause of contention based attacks is the *deterministic memory-to-cache mappings, causing deterministic cache contention*. The root cause of reuse based attacks is more fundamental, since reuse of the data is the primary goal of a cache. We observe that the hidden assumption of the reuse based attacks is that the access to a security-critical data will bring the requested memory line into the cache (if not yet cached), which is exactly the *demand fetch* policy of all existing caches, that take advantage of both temporal and spatial locality of a program. *With the demand fetch policy, cache fills are always correlated with memory accesses in a deterministic way, and the state of the cache can "remember" information of previous demand accesses.*

Resource contention of shared micro-architectural components (either memoryless or storage components) [10], [16]–[18] have been well-known as sources of information leakage. Reuse based attacks are fundamentally different since they do not rely on any resource contention, and represent new threats specific to storage structures such as cache and buffer structures that exploit the locality principle to store recently-used data of a larger storage structure.

C. Case Study: Cache Collision Attacks against AES

As a concrete example, we show how the cache collision attack works to extract the AES encryption keys (e.g., in the OpenSSL implementation of AES). AES is a block cipher that has 128-bit blocks with three possible key sizes: 128, 192, 256 bits. Depending on the key size, AES performs 10, 12 or 14 rounds, respectively [19]. We use 128-bit keys in our discussion. The output of each round will be the input for the next round, and the operations in each round are implemented as table lookups for performance reasons. OpenSSL uses ten 1-KB lookup tables, five for encryption and five for decryption. Four tables are used in each round except that the final round uses a different lookup table.

To perform the attack, the attacker sends random plaintext blocks to the victim to do AES encryption, and measures the time for each block encryption. Before triggering the next block encryption, the attacker cleans the cache so that each block encryption starts from a clean cache. Cache collision attacks assume that the attacker either knows the plaintext (first-round attack) or the ciphertext (final-round attack).

First-round attack: The index of the first round table lookup x_i is related with the key byte k_i and plaintext byte p_i as $x_i = k_i \oplus p_i$. If the attacker learns that two memory accesses collide, i.e., $\langle x_i \rangle = \langle x_j \rangle$, he can infer that $\langle k_i \oplus k_j \rangle = \langle p_i \oplus p_j \rangle$.

Final-round attack: The index of a final round table lookup x_u^{10} of table T_4 is related with the final round key byte k_i^{10} and ciphertext byte c_i as $T_4[x_u^{10}] \oplus k_i^{10} = c_i$. Hence, by learning two table lookups x_u^{10} and x_w^{10} collide in the final round table lookups, the attacker can infer that $k_i^{10} \oplus k_j^{10} = c_i \oplus c_j$.

Since a cache collision of two memory accesses x_i and x_j means a lower execution time on average, the attacker can aggregate the time measurements according to the value of the XORed plaintext (or ciphertext) bytes and find the

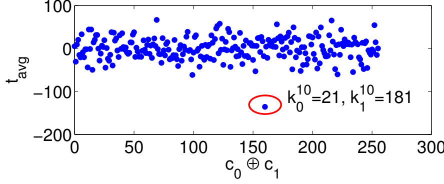

Figure 2. Timing characteristic chart for $c_0 \oplus c_1$. The minimum average encryption time occurs at $c_0 \oplus c_1 = 160$, implying $k_0^{10} \oplus k_1^{10} = 160$.

minimum encryption time. Figure 2 shows the average encryption time for the samples with the same value of XORed $c_0 \oplus c_1$. The data is collected by running 2^{17} block encryptions on the cycle-accurate gem5 simulator [20]. The attacker can easily find the point with minimum average encryption time, then he can infer $k_0^{10} \oplus k_1^{10} = c_0 \oplus c_1 = 160$. He can get similar timing characteristics for 15 XORed ciphertext bytes ($c_0 \oplus c_i$, $i = 1, ..., 15$), and thus infer 15 XORed key bytes to recover the full 16-bytes key (by guessing just one key byte k_0^{10}).

III. Past Work

A. Secure Cache Solutions

Designing secure caches can provide much higher performance, and often greater security, than software solutions for mitigating cache side channel attacks. Two general approaches that have been used are partitioning the cache and randomizing the memory-to-cache mapping.

Partitioning the Cache: The cache is partitioned into different zones for different processes and each process can only access the cache blocks in its zone, thus eliminating contention between a victim process and an attacker process. Partitioning can be achieved statically or dynamically.

NoMo [13] cache uses static cache partitioning by simply reserving one or more ways of a set for each hardware thread. However, it only works for the case when the victim and the attacker processes are executing simultaneously in a simultaneous multi-threading (SMT) processor.

PLcache [11] performs finer-grained, dynamic partitioning, which does not statically reserve any cache lines for a process; instead it locks a protected cache line into the cache and does not allow it to be evicted by another process. Each cache line is extended with the process identifier and a locking status bit. The architectural support also includes special load/store instructions for fine-granularity locking/unlocking. The special load/store instructions are similar to the normal load/store instructions, except that if the memory access hits in the cache or causes a cache line to be fetched into the cache, the locking status bit is set (lock) or cleared (unlock).

Randomizing memory-to-cache mapping: RPcache [11] and Newcache [12] are cache designs using memory-to-cache mapping randomization. In RPcache, there is a permutation table for each trust domain, which can permute the index bits to the cache set. If a process wants to replace cache line X in cache set S, which belongs to a process in another trust domain, a cache line Y in a randomly selected cache set S' will be evicted instead of evicting cache line X. The cache indices of S and S' will be swapped in the process' permutation table and other cache lines in S and S' belonging to the process are invalidated. Therefore, the attacker cannot get useful information from the cache line eviction and replacement.

Newcache [12], [21] can randomize the mapping to each single cache line by adopting a logical direct-mapped cache architecture, hence avoiding the swapping of cache sets and invalidating of other lines in these swapped cache sets – resulting in better performance. Newcache introduces a remapping table as a level of indirection, which stores the mapping from the index bits of the address to a real cache line. Protected processes have different remapping tables, while all unprotected processes share the same remapping table. Newcache can avoid many conflict misses by using a longer index (additional bits) than needed for the actual size of the physical cache – again improving performance. The remapping table is dynamically updated and randomized using a security-aware replacement algorithm, hence randomizing the cache contention between the victim and the attacker.

Both Partitioning and Randomization based approaches only target contention based attacks, and cannot defeat reuse based attacks. These approaches differ from the conventional set-associative caches mainly in where the memory line can be placed in the cache. Consider, for example, the data reuse between memory accesses x_i and x_j, in the cache collision attack. For the Partitioning based approaches, access to x_i will bring the memory line $\langle x_i \rangle$ into its own cache partition, and accesses to x_j will hit in the cache when $\langle x_i \rangle = \langle x_j \rangle$. Similarly, in randomization based approaches, if memory line $\langle x_i \rangle$ was brought into a random location in the cache, the second memory access would still result in a cache hit if $\langle x_i \rangle = \langle x_j \rangle$. Hence, the root cause of cache collision attacks still holds.

B. Constant Execution Time Solutions

Achieving constant execution time in cache accesses [4], [8], [14] could potentially eliminate cache side channel attacks based on the timing difference of cache hits and misses, including reuse based attacks. One drastic approach is to disable the cache for security-critical accesses [8] so that all accesses to security-critical data miss in the cache. This will severely degrade performance. Getting all cache accesses to be hits is ideal, and this can be done using other on-chip storage [22], rather than a cache.

Constant time using a cache can also be approached by "preloading" all the security-critical data into the cache, so that all accesses to them are cache hits. This has been done by rewriting the software cipher to preload all the critical data before each round [15]. However, performance

is significantly degraded, and the preloaded data can still be evicted within a round of encryption. With some hardware support, performance may be improved. [14] proposed to use the "informing loads" technique [23] to perform the "preloading". Critical data is loaded using informing load instructions, and on a cache miss, a user-level exception handler is invoked to perform the preloading. Security-critical data can also be preloaded during context switches [14] and then "locked" in the cache by the cache line locking mechanism of PLcache [11]. "PLcache+preload" is simpler than "informing loads" in terms of hardware and software changes, and has better performance due to less frequent invocation of the preloading routine on context switches rather than cache misses.

Unfortunately, "preloading" based approaches still have scalability and performance issues, and may cause new security problems, which we will discuss in detail in sections VI and VIII.

In this paper, we try to find a general approach against reuse based attacks, as a complement to existing secure cache design approaches, which only defend against contention based attacks. Contrary to conventional wisdom, we also show that constant execution time is not the necessary condition to defeat cache collision attacks and other reuse based attacks. Our defense strategy leads to a simpler hardware based solution, without the need to frequently preload or reload all the security-critical data into the cache.

IV. Random Fill Cache Architecture

A. Random Cache Fill Strategy

Our key insight is that the root cause of reuse based attacks suggests that the cache fill strategy has to be re-designed to de-correlate the cache fill and the demand memory access. We propose using a random cache fill strategy to dynamically achieve the de-correlation. On a cache miss, the missing data is sent to the processor without filling the cache. To still get performance from the cache, we fill the cache with randomized fetches within a configurable neighborhood window of the missing memory line instead. The idea is partially motivated by our observation that accesses to the security-critical data in cryptographic programs usually have random patterns, due to the nonlinearity of the lookup tables (e.g., S-box) and to the random keys. Therefore, randomly fetching the neighborhood memory lines is as good as demand fetching the missing memory line. The random fetching within the spatial locality of the neighboring memory locations is like prefetching, and hence performance may not be degraded, and could even be improved in some cases.

The random cache fill strategy represents a more general and flexible cache fill strategy than the demand fetch policy, and the degree of de-correlation can be configured by changing the random fill window size. We will show in section V that our random cache fill strategy can provide an information-theoretic security assurance against reuse based attacks by choosing a proper random fill window size. As a cache fill strategy, it can be built on any existing secure cache architecture (e.g., Newcache [12], [21]) to provide built-in security against all known cache side channel attacks.

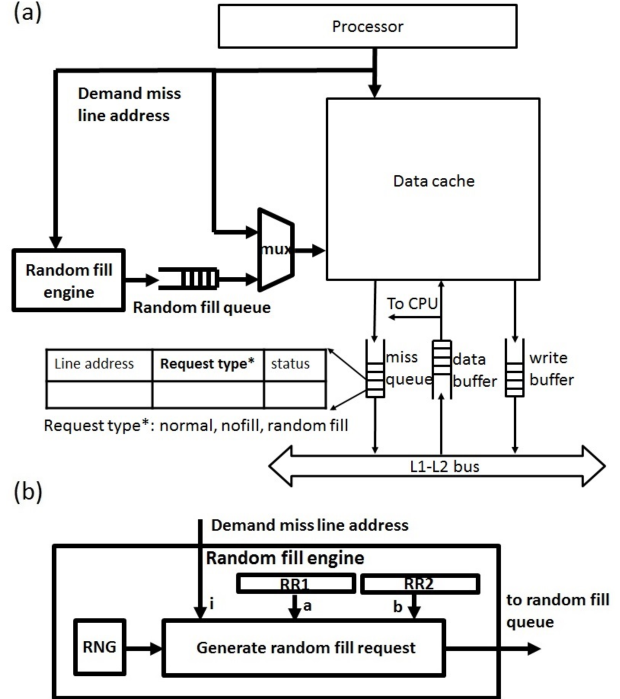

Figure 3. (a) block diagram of random fill cache, (b) blow up of random fill engine

B. Random Fill Cache Architecture

A block diagram of the random fill cache architecture is shown in Figure 3(a). We focus on the L1 data cache since cache side-channels are most effective (fastest) in L1 data caches. It is built upon a conventional non-blocking cache and the hardware addition is very small (highlighted in bold in Figure 3), essentially a random fill engine, a queue and a multiplexer.

1) Hardware for No Demand Fill: In a non-blocking and write-back cache, an entry in the miss queue records the missing memory line address and the status of the request. We add a field to miss queue entries to indicate the request type: normal, nofill or random fill:

- *Normal request* is a demand fetch as in a conventional cache that does demand fill; it fills the cache with the missing line, and the data returned will be sent to the processor.
- *Nofill request* is a demand fetch that directly forwards returned data to the processor while not filling the cache. This leverages the critical word first technique typically implemented to reduce the cache miss latency,

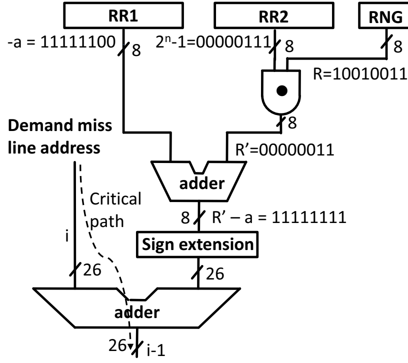

Figure 4. Efficient generation of the address of the random fill request. The example shows a random fill window $[i-4, i+3]$. RR_1 stores the lower bound $-a = -4$ and RR_2 stores the window size mask $2^3 - 1$. Both the range registers and RNG are 8-bits in width, and the generated random fill request is $i-1$.

so no extra hardware is required to implement the forwarding of data.

- *Random fill request* only fills the cache but does not send any data to the processor.

2) Random Fill Engine (Figure 3(b)): Upon a cache miss, the demand requested memory line will not be filled into the cache. Instead, the random fill engine generates a random fill request with an address within a neighborhood window $[i-a, i+b]$ which is a memory lines before and b memory lines after the demand request for memory line i. The two boundaries a and b are stored in two new range registers, $RR1$ and $RR2$, which bound the range of the random number generated from a free running random number generator (RNG). For example, the RNG can be implemented as a pseudo random number generator with a truly random seed. The use of RNG does not impact the cycle time since it is used only during a cache miss and hence is not in the critical path of the processor's pipeline. Furthermore, the random number can be generated ahead of time and buffered. Note that when the range registers are set to zero, randomized cache fill is essentially disabled. In this case, the demand request will be sent as a normal request and no random fill request is generated. The random fill request goes to a random fill queue (a First In First Out (FIFO) buffer) where it waits for idle cycles to lookup the tag array of the data cache. If the random fill request hits in the cache, it is dropped. Otherwise a random fill request is issued and put into the miss queue.

Table II
ALTERNATIVE SYSTEM CALLS FOR CONFIGURING RANDOM FILL WINDOW DYNAMICALLY (ONLY 1 NEEDED)

Declaration	Description
set_RR(int a, int b)	set range register RR_1 and RR_2 to the given value a and b, respectively
set_window(int lowerBound, int n)	set the lower bound and size of the random fill window to $lowerBound$ and 2^n, respectively

3) System Interface: The two range registers, $RR1$ and $RR2$, are configurable by the operating system (OS). As shown in Table II, the OS provides a system call *set_RR* to set the range registers by the compiler and/or applications. This system call provides a fine-granularity control of the use of the random fill cache. By default, the two range registers are set to zero and the random fill cache works just like the conventional demand-fetch cache. The system call can be inserted before the cryptographic operations either by the compiler or by the applications to enable randomized cache fill. They can be disabled afterwards by another call to *set_RR*. The range registers are part of the context of the processor and need to be saved to, and restored from, the process control block (PCB) for a context switch.

4) Optimization: Since it may be non-trivial to generate a random number within an arbitrary bound, we also propose an optimization that constrains bounds a and b so that $a + b + 1 = 2^n$, i.e., the window size is a power of two. Instead of *set_RR*, a different system call *set_window* is implemented: this takes the lower bound of the random fill window (i.e., $-a$) and the logarithm of the window size (i.e., n) as parameters. Instead of directly storing a and b, the range registers store the lower bound $-a$ and a mask for the window (i.e., $2^n - 1$), as shown in Figure 4. The masked random number is $R' = 3$, which when added to the lower bound -4 gives the bounded random number -1. Since the bounded random number can be computed ahead of time, the critical path only consists of one adder that adds the demand miss line address i and the bounded random number (as shown by the dotted arrow).

V. SECURITY EVALUATION

As shown in Table I, reuse based attacks consist of cache collision attacks and Flush-Reload attacks, which correspond to two information leakage channels: the timing channel and the storage channel. By definition, the timing channel exploits the timing characteristics of events to transfer information [24], [25], whereas the storage channel transfers information through the setting of bits by one program and the reading of those bits by another [24]. We show that our random cache fill strategy is able to completely close the known timing channel and provide a strong information-theoretic security assurance against the storage channel, when the random fill window of the victim process is properly chosen.

A. Timing Channel

We analyze known cache collision attacks that exploit the impact of the reuse of security-critical data on the aggregated time. Data reuse always comes in a pair of memory accesses (x_i and x_j, where x_i precedes x_j). We abstract the impact as the difference of the expected execution time under the conditions of cache collision (μ_1) and no collision (μ_2), respectively:

Table III
$P_1 - P_2$ AND THE NUMBER OF MEASUREMENTS FOR A SUCCESSFUL CACHE COLLISION ATTACK, FOR VARIOUS WINDOW SIZES

window size:		**size=1**	size=2	size=4	size=8	size=16	size=32
Random fill + 4-way SA	$P_1 - P_2$	**0.652**	0.332	0.127	0.044	0.012	0.006
	# measurements	**65,000**	1,866,000	16,653,000	no success after trying 2^{24} measurements		
Random fill + Newcache	$P_1 - P_2$	**0.576**	0.292	0.119	0.045	0.016	0.007
	# measurements	**244,000**	2,106,000	no success after trying 2^{24} measurements			

$$\begin{aligned} \mu_1 &= \mu_0 + P_1 \cdot t_{hit} + (1 - P_1) \cdot t_{miss} \\ \mu_2 &= \mu_0 + P_2 \cdot t_{hit} + (1 - P_2) \cdot t_{miss} \end{aligned} \tag{2}$$

where μ_0 is the expectation of the aggregated time excluding the security-critical access x_j, t_{hit} and t_{miss} are the cache hit and miss latencies, respectively. P_1 and P_2 are the cache hit probabilities under the conditions of cache collision and no collision, defined as follows:

$$\begin{aligned} P_1 &= P\left(x_j \text{ hit} | \langle x_i \rangle = \langle x_j \rangle\right) \\ P_2 &= P\left(x_j \text{ hit} | \langle x_i \rangle \neq \langle x_j \rangle\right) \end{aligned} \tag{3}$$

The expectation of the timing difference ($\mu_2 - \mu_1$) is the signal that an attacker wants to extract, which depends on P_1 and P_2 as follows:

$$\mu_2 - \mu_1 = (P_1 - P_2)(t_{miss} - t_{hit}) \tag{4}$$

This means that no information can be extracted from the timing characteristics if $P_1 - P_2 = 0$ for any arbitrary pair of security-critical accesses. In fact, $P_1 - P_2$ (or $\mu_2 - \mu_1$) directly reflects the difficulty of the attack. The number of measurements required for a successful attack is related to $P_1 - P_2$ as:

$$N \approx \frac{2Z_\alpha^2}{\left(\dfrac{(P_1 - P_2)(t_{miss} - t_{hit})}{\sigma_T}\right)^2} \tag{5}$$

where α is the desired likelihood of discovering the secret key, and represents how we define a successful attack. Z_α is the quantile of the standard normal distribution for a probability α. σ_T is the variance of the execution time. Equation (5) is obtained using a derivation similar to that in [26] and [27]. It indicates that when $P_1 - P_2 = 0$, the attack cannot succeed (infinite number of measurements are required).

The purpose of the random cache fill strategy is to zero out the signal that an attacker can extract, not just simply add noise to the attacker's measurements. In the following, we show how the random cache fill strategy achieves $P_1 - P_2 = 0$. Assume the security-critical data is contained in a contiguous memory region with M cache lines, starting at M_0. Consider two memory accesses x_i and x_j to the security-critical memory lines i and j where $i, j \in [M_0, M_0 + M - 1]$, respectively. Further assume 1) the memory line i is not cached yet (i.e., the cache warm-up phase) and the memory access x_i initiates a random fill within window $[i - a, i + b]$, 2) there are no other accesses to memory line i in between x_i and x_j, 3) the cache state is clean. This represents the best case for the attacker. Then we can calculate the two conditional probabilities:

$$\begin{aligned} P_1 &= \frac{1}{a+b+1} \\ P_2 &= \begin{cases} \dfrac{1}{a+b+1}, & j \in [i-a, i+b] \\ 0, & j \notin [i-a, i+b] \end{cases} \end{aligned} \tag{6}$$

Equation (6) indicates that for arbitrary i and j, if j is in the random fill window of i, $P_1 - P_2 = 0$ always holds. The sufficient and necessary condition that gives $j \in [i-a, i+b]$ is $a, b \geq M - 1$. This means that when the random fill window is sufficient to cover the whole lookup table, the random fill cache ensures that $P_1 - P_2 = 0$ for any pair of security-critical accesses, and hence completely closes the timing channel. However, if we examine $P_1 - P_2$ for the conventional set-associative cache, and for the Partitioning and Randomization based secure caches, we find that $P_1 - P_2 \approx 1$ always holds under the same assumptions, since the memory access x_i will always bring the memory line i into the cache.

Note that a cryptographic program may contain multiple independent lookup tables, so *the total size of all the security-critical data may not be small – this is why other solutions which require pre-loading all the security-critical data may fail to scale when the size of security-critical data is large.* However, an individual table is usually small and the random fetch window size is determined by the individual table size, and thus is usually relatively small.

Case Study – Cache Collision Attacks against AES: In fact, the timing channel can be substantially mitigated even when the window size is small. In this section, we use cache collision attacks against AES as a case study to investigate how the random fill window size impacts the security of a random fill cache. Consider the final round AES table T_4. It is 1 KB in size and contains 16 cache lines (assume cache line size is 64 bytes). So, a random fill window with $a = b = 15$ is large enough to cover the whole AES table for any table lookup. There are 16 table lookups to T_4 for each block encryption. We use Monte Carlo simulation to calculate the average $P_1 - P_2$ for all the table lookup pairs within the 16 table lookups. We perform 100,000 trials in the Monte Carlo simulation and each trial does AES encryption of one block of random plaintext.

Table III shows the average $P_1 - P_2$ for different random fill window sizes. We include results for two cases: 1) the random fill cache built upon a conventional 4-way set-associative (SA) cache (32 KB in size); 2) the random fill cache built upon Newcache (with the same cache size as the SA cache). The first column with "size=1" in Table III also represents the 4-way SA cache and Newcache with the demand fetch policy. We use a bidirectional random fill window $[i - 2^n, i + 2^n - 1]$, because the randomized table lookups in cryptographic algorithms do not favor the forward direction over the backward direction, so a bidirectional random fill window has the best security.

We make the following observations: 1) Both SA cache and Newcache have a large $P_1 - P_2$, thus are vulnerable to cache collision attacks (the first column with "size=1"). 2) Our random cache fill strategy is effective against cache collision attacks for both SA cache and Newcache. In fact, as a cache fill strategy, it can be built on any cache architecture. 3) $P_1 - P_2$ drops dramatically as the window size increases. Note that due to the inaccuracy of the Monte Carlo method, we cannot achieve exactly $P_1 - P_2 = 0$ when the window size is 32.

We further verify our conclusions by performing real cache collision attacks [4]. The simulator configuration is similar to what we use for the performance evaluation (Table IV) except we minimize the impact of a non-blocking cache by using only 1 miss queue entry for 1 non-blocking cache miss. This configuration favors the attacker since we find that it requires about 1 order of magnitude less samples compared to the baseline configuration in Table IV, which has 4 miss queue entries. The results are shown in Table III. When the window size increases, the number of measurements required increases drastically. When the window size is larger than 4, the attacks all fail after collecting 2^{24} measurements. (It takes more than three weeks of continuous simulation on gem5 to collect 2^{24} measurements). Note that attacking Newcache requires slightly more measurements than attacking the SA cache because cache collision attacks require each measurement to start from a clean cache, and completely cleaning Newcache is harder than cleaning the SA cache, due to Newcache's random replacement algorithm.

B. Storage Channel

For the storage channel, e.g., in a Flush-Reload attack, we can directly calculate the channel capacity using the channel model in [11]. Similar to the proof for the timing channel, we assume the security-critical data is contained in a contiguous memory region with M cache lines, starting at M_0. The victim process is the sender who accesses security-critical memory line $i \in [M_0, M_0 + M - 1]$, which can be represented by a random variable S. Let j be the memory line that is randomly filled into the cache due to the access of i, then we have $j \in [M_0 - a, M_0 + M - 1 + b]$ for the random fill

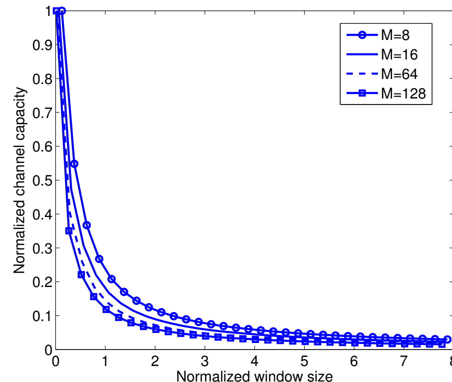

Figure 5. Channel capacity for different window sizes. The window size is normalized to the size of the security-critical data (contains M cache lines). The channel capacity is normalized to the demand fetch case.

window of the form $[i - a, i + b]$. Due to the boundary effect, memory lines that are outside the security-critical region may be brought into the cache. Here, we assume the best case for the attacker: the attacker can access the data outside the security-critical region and hence can determine which memory line is brought into the cache. Therefore, the attacker is the receiver who can exactly observe the symbol j, which can be represented by a random variable R. Then the conditional probability that the attacker receives a symbol j given the victim process sends symbol i is

$$P_{ij} = P(R = j | S = i) = \begin{cases} \frac{1}{W} & i - a \le j \le i + b \\ 0 & otherwise \end{cases} \quad (7)$$

where W is the window size: $W = a + b + 1$.

The channel capacity is the mutual information of S and R when S satisfies a uniform distribution, i.e., $P(S = i) = 1/M$. We have the channel capacity as [28]

$$\begin{aligned} C &= \sum_{i,j} P(S = i, R = j) log \frac{P(S = i, R = j)}{P(S = i) \cdot P(R = j)} \\ &= \sum_{i,j} \frac{1}{M} P_{ij} \cdot log \frac{M \cdot P_{ij}}{\sum_{i} P_{ij}} \end{aligned} \quad (8)$$

where M is the number of cache lines of the security-critical data, P_{ij} is the conditional probability defined in Equation (7).

Figure 5 shows how the window size impacts the channel capacity for various sizes of security-critical data. The channel capacity is normalized to the demand fetch case and the window size is normalized to the size of the security-critical data. Due to the boundary effect, the storage channel cannot be completely closed. However, we find that the channel capacity drops dramatically as the window size increases. For example, the channel capacity is already reduced by more than one order of magnitude when the window size is twice the size of the security-critical region. The impact

Table IV
SIMULATOR CONFIGURATIONS

Parameter	Value
ISA	ALPHA
Processor type	4-way out-of-order
L1 instruction cache	4-way 32 KB
L2 cache	8-way 2 MB
Cache line size	64 bytes
Cache replacement algorithm	LRU
miss queue entries	4
L1/L2 hit latency	1 cycle / 20 cycles
DRAM frequency/channels	DDR3-1600/1

of the boundary effect is smaller for larger security-critical regions. As mentioned in section II, the Flush-Reload attack requires the attacker and the victim to share the security-critical data, which can be easily disabled by not declaring the security-critical data as read-only, hence the information leakage through the storage channel is a less serious threat than the information leakage through the timing channel. Nevertheless, our random fill cache still provides a strong information-theoretic security assurance against this type of side channel leakage even when sharing is allowed.

VI. PERFORMANCE EVALUATION

We implemented our random fill cache on a cycle-accurate simulator, gem5 [20]. We use a 4-way, out-of-order processor with two levels of caches in our performance evaluation, and the baseline configuration is shown in Table IV. The baseline cache is a set-associative cache with least recently used (LRU) replacement algorithm and without a prefetcher. The DRAM models the detailed timing of a single channel DDR3-1600.

Performance impact on cryptographic algorithm: We first study how the random fill cache performs for cryptographic algorithms. Our workload is the OpenSSL's AES encryption that takes a 32 KB random input and does a cipher block chaining (CBC) mode of encryption. The results are shown in Figure 6 for various cache sizes and associativity. The Instruction Per Cycle (IPC) metric is normalized to the baseline demand-fetched cache with the same cache size and associativity. We also compared the performance with two previous constant-time solutions against the cache collision attacks: the "disable cache" approach disables the cache for security-critical accesses to the 5 AES tables for encryption. The "PLcache+preload" approach pre-loads all the 5 AES tables and uses the locking mechanism provided by the PLcache to lock these tables in the cache. We chose the "PLcache+preload" approach because it has better performance than the "informing loads" approach [14]. The random fill window is configured to be of the form $[i - 16, i + 15]$, which can cover the whole table for any pair of security-critical accesses to the table.

As can be expected, the "disable cache" solution degrades the performance by 45% for all the cache configurations, since security-critical accesses contribute about 24% of

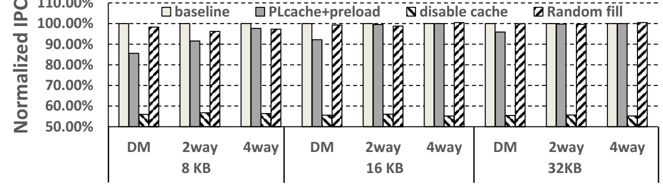

Figure 6. Performance impact on cryptographic program

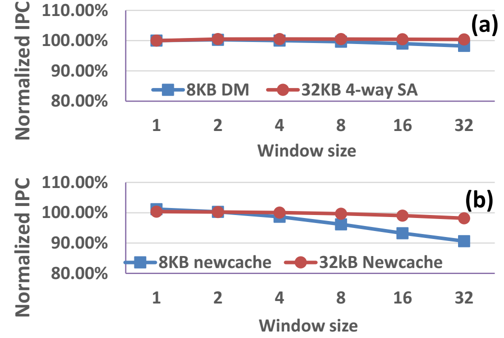

Figure 7. Impact of window size for the random cache fill strategy built on (a) SA cache, (b) Newcache, when running AES

the total accesses to the data cache. The performance of "PLcache+preload" is sensitive to the cache size and associativity. When the cache size is small (8KB), more than half of the cache lines are locked for the security-critical data. It incurs 15% degradation for the direct-mapped (DM) cache and 3% degradation for the 4-way SA cache when the cache size is 8 KB. "PLcache+preload" also does not work well when the associativity is low; it incurs more than 4% degradation for the DM cache even when the cache size is 32 KB. The random fill cache is less sensitive to the cache size and associativity. The performance degradation is less than 3.5% even when the cache size is 8 KB, and there is no degradation for larger caches.

Note that our conclusions are not specific to AES. The reason why the random fill cache can maintain good performance is that it still takes advantage of the spatial locality of the key-dependent data flow. The slight performance degradation when the cache size is small is due to fetching of unused data that is outside the security-critical region. We also study how the performance is impacted when both the L1 and L2 caches are random fill caches. We find that the performance impact is negligible since the L2 cache is large and can better tolerate the potential cache pollution due to the random fill of unused data.

Impact of window size: Figure 7(a) shows that when the random fill cache is built upon the SA cache, the performance is not sensitive to the window size for both the worst case (8 KB DM cache) and the best case (32 KB 4-way SA cache). When built upon Newcache (Figure 7(b)), our random fill cache works slightly worse than for the SA cache, when running AES. As the window size increases,

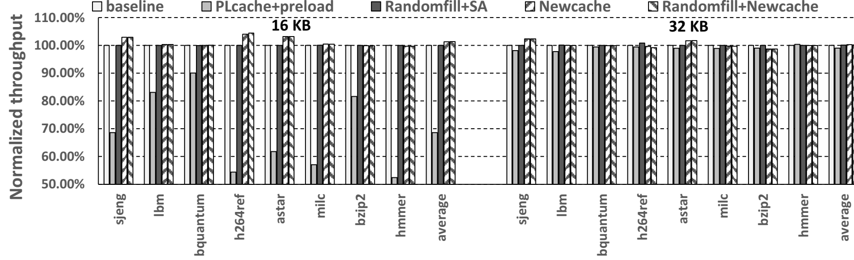

Figure 8. Performance impact on other programs running concurrently

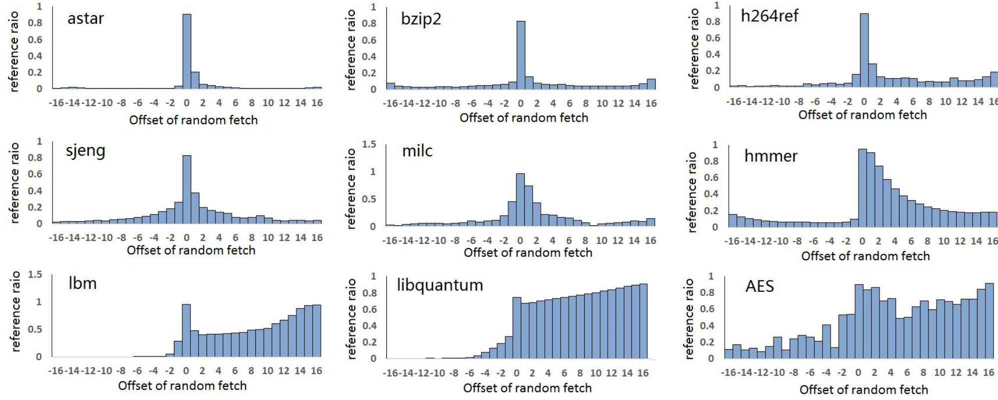

Figure 9. Effectiveness of random cache fill strategy for general workloads

the performance also decreases (maximum degradation is 9% when the window size is 32 for the 8 KB cache). This is because when the window size is large, it is more likely to fetch unused data into the cache and evict frequently-used data through the random replacement algorithm.

Performance impact on other concurrent programs: Since a random fill window of $[0, 0]$ is used by default, there is no impact to a non-cryptographic program when it runs alone. Instead, we study how the performance of a non-cryptographic program is impacted when it runs concurrently with a cryptographic algorithm in a simultaneous multi-threading (SMT) processor. To stress the cache, the cryptographic program continuously does both AES decryption and encryption of 32 KB random data. In this case, the security-critical data includes 10 AES tables for both encrytion and decryption. A bidirectional random fill window with a size of 32 lines is used for the crpytographic program. The non-cryptographic programs are 8 SPEC2006 benchmarks and are run for 2 billion instructions with reference inputs. We consider two cache configurations: 16 KB DM cache and 32 KB 4-way SA cache. The results are the normalized throughput (IPC) for the non-cryptographic program.

As shown in Figure 8, for all the benchmarks, we observe no impact of the random fill cache on the throughput of the non-cryptographic programs, for both the case when the random fill cache is built on the SA cache and when it is built on the Newcache. This is because our random fill cache does not need to reserve any cache lines; for the non-cryptographic algorithms, random fill request is no different than a demand fetch request. In contrast, “PLcache+preload” incurs significant impact on the non-cryptographic programs. When the cache size is relatively small (16 KB), the performance degradation is 32% on average. Even when cache size is 32 KB, it still incurs an average degradation of 1%. Therefore, scalability is a serious problem for the “PLcache+preload” approach. When the size of the security-critical data is large (relative to the cache size), the cache available for other data is reduced – causing performance degradation of both the cryptographic algorithm and other concurrently running programs. The scalability issue is especially problematic for the L1 data cache. L1 data cache is usually small but cache side channel attacks are most effective on the L1 data cache, rather than on the L2/L3 caches.

VII. Performance Benefits of Random Fill Cache

Although our random fill cache is proposed for security, it also provides architectural support for a more flexible and general cache fill strategy than the demand fetch policy. We now study the extent to which a general non-cryptographic program can benefit from the random cache fill strategy to improve its performance. The performance implication of random fill is that it can take advantage of spatial locality beyond a cache line, while the demand fetch strategy can only take advantage of spatial locality within a cache line.

We first investigate the effectiveness of the random cache fill strategy through profiling. Specifically, we study how

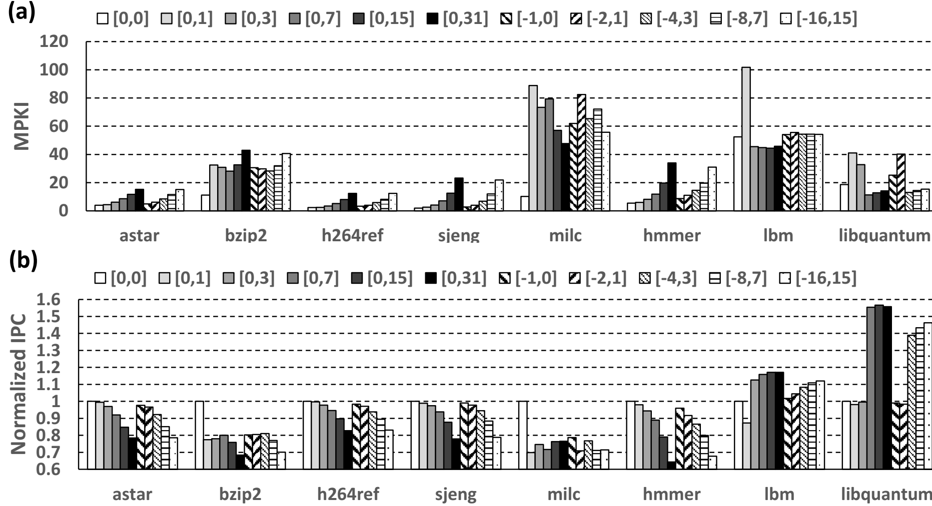

Figure 10. (a) L1 data cache MPKI and (b) IPC for random fill cache with different random fill window sizes. $[-a, b]$ indicates a range of a memory lines before, to b memory lines after, the demand requested memory line

the performance of non-cryptographic programs is impacted when a L1 data cache exploits random fill strategy. The baseline configuration is the same as that in Table IV. We tag each randomly filled memory line with an offset (denoted as d) with respect to the associated demand requested memory line. The effectiveness (reference ratio) of the random fill is defined as the ratio of the number of cache lines that are referenced before being evicted after being brought into the cache, over the total number of fetched memory lines:

$$Eff(d) = \frac{N_{referenced}(d)}{N_{fetched}(d)} \tag{9}$$

Note that the profiling method essentially provides a way to sample the spatial locality of a program. The profiling results are shown in Figure 9, with a maximum d of ± 16. We find that many workloads only have spatial locality spanning about four neighborhood cache lines or less. Therefore, demand fetch will work well for these workloads. However, there are some benchmarks that do not perform well under demand fetch. For example, lots of irregular streaming access patterns can be found in **libquantum** and **lbm**, which shows wider spatial locality beyond a cache line, especially in the forward direction. The random access patterns, as can be found in AES, also show wide spatial locality beyond a cache line.

Figure 10 shows the L1 data cache Misses Per Kilo Instructions (MPKI) and IPC for the SPEC benchmarks with different random fill window sizes (both forward and bidirectional windows). The MPKI counts the number of cache misses that cause a data fetch request to the L2 cache, excluding outstanding misses to the same cache line. Note that the random fill window $[0, 0]$ means demand fetch only (first bar for each benchmark in Figure 10). We simulate two billion instructions using the reference inputs. We insert the system call for setting the range registers of the random fill cache at the beginning of the program, which essentially enables random fill for all the memory accesses.

The MPKI results agree well with the profiling results in Figure 9. Figure 10(a) shows that for benchmarks without wide spatial locality, a larger random fill window tends to increase the L1 cache MPKI, compared to demand fetch. Two exceptions are **lbm** and **libquantum**. In these two benchmarks with irregular streaming patterns, a larger random fill window actually reduces the L1 cache MPKI compared to demand fetch, especially with a forward window. Figure 10(b) shows that as the MPKI increases for larger window sizes, the overall performance in IPC decreases, as expected – except for **lbm** and **libquantum**.

For these streaming applications, we notice that the IPC seems to improve more than the L1 MPKI decreases. For **libquantum**, the best performance is achieved when the random fill window is $[0, 15]$: the MPKI is reduced by 31% while the IPC is increased by 57%. This is likely due to the fact that our random fill cache reduces L2 MPKI in addition to reducing L1 data cache MPKI. Consider the following case: if a demand request to $X[i]$ misses in both the L1 data cache and the L2 cache, and the random fill request triggered by the demand request also misses in the L2 cache, both the cache line containing $X[i]$ and one of its neighboring cache lines will be brought into the L2 cache. Since these two cache lines brought into the L2 cache are likely to be used in the future, the L2 cache miss rate may be significantly reduced. Meanwhile, although the cache line containing $X[i]$ does not fill the cache, which may cause extra cache misses for the subsequent accesses to the same cache line, this need not increase the overall L1 MPKI because a random fill request for a neighboring cache line is generated and fetched into the L1 data cache (if not already there), and this is likely to be referenced in the near future, especially for irregular streaming applications like **libquantum**, as shown in Figure

9. Also, the extra cache misses for the accesses $X[i+1]$, etc., in the same cache line as $X[i]$, do not take a whole cache miss latency in non-blocking caches (like the one we simulate with 4 miss queue entries), if they occur during the time $X[i]$ is being fetched or in the miss queue.

Note that a random fill cache does increase L2 cache and memory traffic due to extra random fill requests. For **lbm** and **libquantum**, the traffic to the L2 cache is increased by 48% and 56%, but the traffic to the memory is increased only by 0.03% and 22%, respectively.

Note that the streaming patterns in **lbm** and **libquantum** are irregular and may be too complex for a simple hardware prefetcher [29]. The use of our random fill cache may give better performance than the demand-fetched cache with a simple prefetcher for these benchmarks. For example, we compare the result with a commonly used tagged prefetcher [30], that associates a 1-bit tag with the cache line to detect when a demand-fetched or prefetched cache line is referenced for the first time, to fetch the next sequential line. We find that the tagged prefetcher can only improve the IPC performance by 11% for **lbm** and 26% for **libquantum**, while our random fill cache improves IPC by 17% for **lbm** and 57% for **libquantum**. Further performance improvements with the random fill cache may be possible by getting spatial locality profiles for different phases of the program, and setting the appropriate window size for each phase.

While more sophisticated prefetchers can certainly give better performance, our goal here is to show that design-for-security need not necessarily degrade performance, but may even improve performance, as shown by the random fill cache for streaming applications like **libquantum** and **lbm**.

VIII. Comparison with Past Work

Our random fill cache provides architectural support for a security-critical program to protect itself against reuse based attacks, by properly configuring its own random fill window size. A random fill cache hardly incurs any performance degradation, and can sometimes even improve the performance of programs that have irregular streaming patterns. The hardware addition is very small, and only the cache controller needs to be changed slightly. Also, only trivial software changes are required: to set the window size at the beginning of the cryptographic routine or the security-critical or streaming program.

We now compare our Random Fill cache with the past work described in section III. The Partition and memory-to-cache mapping Randomization based secure cache designs only target the contention based attacks and cannot defeat the reuse based attacks, since they still exploit the demand fetch policy, which we have identified as the root cause of reuse based attacks. Our random fill cache can complement these prior secure cache designs.

Although the constant execution time solutions, like "PLcache+preload" and "informing loads" [14], may also defeat reuse based attacks, they may also introduce new security problems which could potentially be more dangerous than even cache side-channel attacks. (We refer to both these techniques as hardware-assisted preloading, or just preloading based approaches.) For example, the "informing loads" approach may create a new vulnerability, since now an attacker can essentially supply malware to be executed on every cache miss. Also, the user-level exception handler for each cache miss is not protected and can easily be attacked. Both these hardware-assisted preloading based approaches are also vulnerable to Denial-of-service (DoS) attacks. For example, an attacker can abuse the locking mechanism of PLcache by locking a lot of cache lines to prevent other processes from utilizing the cache. For the "informing loads" approach, if the attacker frequently evicts the security-critical data, the exception handler for informing loads will be frequently invoked, leading to huge slowdown for the victim. In other words, the defense mechanism itself can be abused by the attacker to create more havoc.

In contrast, the configurable random fill window in our solution cannot be abused by an attacker, since using a large random fill window for his own attack process only makes his measurements harder, and the attacker cannot set the victim's window size. Also, since the random fill cache does not need to preload all the security-critical data, it has much lower performance overhead, and also better scalability to larger amounts of security-critical data (section VI), than the preloading based approaches. Furthermore, our random fill cache is simpler in both hardware and software changes required, and has a much simpler programming model than the preloading based approaches. For example, the "informing loads" approach requires writing and installing of a correct and incorruptible user-level exception handler to preload all the security-critical data, and also rewriting of each cipher to use the new informing load instructions.

IX. Conclusions

Reuse based cache side channel attacks are serious new sources of information leakage in the microprocessor, in addition to the better-known contention based side channel attacks. They do not rely on any resource contention and are threats especially relevant to storage structures (like caches and TLBs) which exploit the locality of data accesses to store data from larger storage structures. We found that the fundamental demand fetch policy in conventional caches is the security vulnerability that causes the success of reuse based attacks. We proposed a random fill cache architecture, which is able to dynamically de-correlate the cache fill with the demand memory access. We proved that the random fill cache provides information-theoretic security against reuse based attacks. We showed that our random fill cache incurs very slight performance degradation for cryptographic algorithms and has no performance impact on concurrent non-security-critical programs. A very interesting result is

that our random fill strategy can be built on existing secure cache designs, e.g., Newcache, to provide comprehensive defenses against all known cache side channel attacks – without degrading performance. Furthermore, our random fill cache provides a more general cache fill strategy than the demand fetch strategy, and can provide performance benefit to some applications that are not demand-fetch amenable, by exploiting spatial locality beyond a cache line.

ACKNOWLEDGMENT

We thank the reviewers for their helpful comments. This work was supported in part by DHS/AFRL FA8750-12-2-0295 and NSF CNS-1218817.

REFERENCES

[1] D. Page, "Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel," Cryptology ePrint Archive, Report 2002/169, 2002.

[2] D. J. Bernstein, "Cache-timing Attacks on AES," Tech. Rep., 2005.

[3] C. Percival, "Cache Missing for Fun and Profit," in *The Technical BSD Conference (BSDCan'05)*, 2005.

[4] J. Bonneau and I. Mironov, "Cache-Collision Timing Attacks against AES," in *Cryptographic Hardware and Embedded Systems (CHES'06)*, 2006, pp. 201–215.

[5] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *Cryptographers' Track at the RSA Conference (CT-RSA'06)*, 2006, pp. 1–20.

[6] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games — Bringing Access-Based Cache Attacks on AES to Practice," in *Proc. IEEE Symposium on Security and Privacy (SP'11)*, 2011, pp. 490–505.

[7] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM Side Channels and Their Use to Extract Private Keys," in *Proc. ACM conference on Computer and Communications Security (CCS'12)*, 2012, pp. 305–316.

[8] D. Page, "Partitioned Cache Architecture as a Side-Channel Defence Mechanism," Cryptology ePrint Archive, Report 2005/280, 2005.

[9] A. Bogdanov, T. Eisenbarth, C. Paar, and M. Wienecke, "Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs," in *Cryptographers' Track at the RSA Conference (CT-RSA'10)*, 2010, pp. 235–251.

[10] Z. Wang and R. B. Lee, "Covert and Side Channels Due to Processor Architecture," in *Proc. Annual Computer Security Applications Conference (ACSAC'06)*, 2006, pp. 473–482.

[11] Z. Wang and R. B. Lee, "New Cache Designs for Thwarting Software Cache-based Side Channel Attacks," in *Proc. ACM/IEEE International Symposium on Computer Architecture (ISCA'07)*, 2007, pp. 494–505.

[12] Z. Wang and R. B. Lee, "A Novel Cache Architecture with Enhanced Performance and Security," in *Proc. IEEE/ACM International Symposium on Microarchitecture (MICRO'08)*, 2008, pp. 83–93.

[13] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Trans. on Architecture and Code Optim.*, vol. 8, no. 4, pp. 1–21, 2012.

[14] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou, "Hardware-Software Integrated Approaches to Defend against Software Cache-based Side Channel Attacks," in *Proc. IEEE International Symposium on High Performance Computer Architecture (HPCA'09)*, 2009, pp. 393–404.

[15] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert, "Software Mitigations to Hedge AES against Cache-based Software Side Channel Vulnerabilities," Cryptology ePrint Archive, Report 2006/052, 2006.

[16] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing Channel Protection for a Shared Memory Controller," in *Proc. IEEE International Symposium on High Performance Computer Architecture (HPCA'14)*, 2014, pp. 225–236.

[17] Y. Wang and G. E. Suh, "Efficient Timing Channel Protection for On-chip Networks," in *Proc. IEEE/ACM Intl. Symposium on Networks-on-Chip (NOCS'12)*, 2012, pp. 142–151.

[18] H. M. G. Wassel, Y. Gao, J. K. Oberg, T. Huffmire, R. Kastner, F. T. Chong, and T. Sherwood, "SurfNoC: A Low Latency and Provably Non-interfering Approach to Secure Networks-on-chip," in *Proc. ACM/IEEE International Symposium on Computer Architecture (ISCA'13)*, 2013, pp. 583–594.

[19] R. B. Lee, *Security Basics for Computer Architects*. Synthesis Lectures on Computer Architecture, Morgan Claypool, 2013.

[20] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[21] F. Liu and R. B. Lee, "Security Testing of a Secure Cache Design," in *Hardware and Architectural Support for Security and Privacy (HASP'13)*, 2013.

[22] R. B. Lee and Y.-Y. Chen, "Processor Accelerator for AES," in *Proc. IEEE Symposium on Application Specific Processors (SASP'10)*, 2010, pp. 16–21.

[23] M. Martonosi, M. D. Smith, T. C. Mowry, and M. Horowitz, "Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors," in *Proc. ACM/IEEE International Symposium on Computer Architecture (ISCA'96)*, 1996, pp. 260–260.

[24] J. C. Wray, "An Analysis of Covert Timing Channels," *Journal of Computer Security*, vol. 1, no. 3, pp. 219–232, 1992.

[25] Z. Wang, "Information Leakage Due to Cache and Processor Architectures," PhD Thesis, Electrical Engineering Department, Princeton University, 2012.

[26] S. Mangard, "Hardware Countermeasures against DPA—A Statistical Analysis of Their Effectiveness," in *Cryptographers' Track at the RSA Conference (CT-RSA'04)*, 2004, pp. 222–235.

[27] K. Tiri, O. Aciicmez, M. Neve, and F. Andersen, "An Analytical Model for Time-driven Cache Attacks," in *Fast Software Encryption (FSE'07)*, 2007, pp. 399–413.

[28] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. Wiley-Interscience, 1991.

[29] J. Mars, D. Williams, D. Upton, S. Ghosh, and K. Hazelwood, "A Reactive Unobtrusive Prefetcher for Multicore and Manycore Architectures," in *Workshop on Software and Hardware Challenges of Manycore Platforms (SHCMP'08)*, 2008.

[30] S. P. Vanderwiel and D. J. Lilja, "Data Prefetch Mechanisms," *ACM Comput. Surv.*, vol. 32, no. 2, pp. 174–199, 2000.