Can randomized mapping secure instruction caches from side-channel attacks?

Fangfei Liu
Princeton University
Princeton, NJ, 08544 USA
fangfeil@princeton.edu

Hao Wu
Princeton University
Princeton, NJ, 08544 USA
haow@princeton.edu

Ruby B. Lee
Princeton University
Princeton, NJ, 08544 USA
rblee@princeton.edu

ABSTRACT

Information leakage through cache side channels is a serious threat in computer systems. The leak of secret cryptographic keys voids the protections provided by strong cryptography and software virtualization. Past cache side channel defenses focused almost entirely on data caches. Recently, instruction cache based side-channel attacks have been demonstrated to be practical – even in a Cloud Computing environment across two virtual machines. Unlike data caches, instruction caches leak information through secret-dependent execution paths. In this paper, we propose to use a classification matrix to quantitatively characterize the vulnerability of an instruction cache to software side channel attacks. We use this quantitative analysis to answer the open question: can randomized mapping proposed for thwarting data cache side channel attacks secure instruction caches? We further study the performance impact of the randomized mapping approach for the instruction cache.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: Design Styles—*Cache memories*

General Terms

Security

Keywords

Cache side channel, SVM, randomized mapping

1. INTRODUCTION

Cache side channels are serious threats to computer systems, from smartphones to multi-tenant cloud computing servers. They can be exploited by attackers to leak cryptographic keys, nullifying any protection provided by strong cryptography. Since they attack the underlying hardware caches, they also void the software memory isolation protections provided by virtual machines. Many real world attacks have been successfully demonstrated on various platforms [17, 11, 18, 16, 12, 10, 15, 22, 21].

Past work on cache side channels tended to focus on the data cache (D-cache). D-cache attacks exploit the interaction of *secret-dependent data access patterns* of a cipher with the underlying D-cache to leak secret information. The behavior of the D-cache enables an attacker to indirectly infer the memory address of a security-critical access, and in the case of a cipher, bits of the secret key.

In instruction cache (I-cache) attacks, however, it is the interaction of *secret-dependent execution paths* with the underlying I-cache that leaks the secret information [9, 10]. Recently, Zhang et al. successfully applied I-cache based side-channel attacks to the noisy, multi-tenant Cloud Computing environment, showing that an attacker virtual machine (VM) can steal the private key used in a victim VM [22]. This elevates I-cache based side-channel attacks to a serious practical threat.

Past work on defenses focused on securing the D-cache against D-cache based side-channel attacks [19, 20]. They showed that randomizing memory-to-cache mapping is effective against contention based attacks [19, 20], without performance degradation. In this paper, we explore the open question: "Are randomized mapping approaches for D-cache also suitable for securing I-caches from cache side-channel attacks?"

We first analyze whether and how the contention based attack techniques for D-cache attacks can be applied to the I-cache. This is because constructing a side-channel attack on the I-cache has many challenges beyond that of D-cache attacks. We find that the access-based attacks that can capture the execution trace of a program are more of a threat for the I-cache than timing-based attacks.

A key insight on I-cache attacks is that they rely on differentiating the footprints of different execution paths in the instruction cache. We therefore propose to use a classification matrix derived from machine learning classification to quantitatively characterize the vulnerability of an I-cache to these side-channel attacks.

We apply our classification matrix to analyze two variations of randomized mapping with different degrees of randomness: a fully-associative cache with a random replacement algorithm, and the Random-Fixed mapping scheme, which is the core mechanism in Newcache [20] – the randomized mapping scheme with the best performance. We find that the fully-associative cache with random replacement algorithm can make the cache footprints of different branches indistinguishable and the Random-Fixed mapping

HASP'15, June 13 2015, Portland, OR, USA
 ISBN 978-1-4503-3483-9/15/06 ...$15.00
DOI: http://dx.doi.org/10.1145/2768566.2768570

can achieve the same indistinguishability as the fully-associative cache, with much less power consumption and access latency.

Our main contributions are:

- Quantitative characterization of the relative vulnerability of different systems to I-cache side-channel attacks using an SVM classification matrix.
- An analysis of the best defense proposed in past work, the Newcache secure data cache architecture, to shed insight into its basic security features, and a comprehensive evaluation of its suitability for use as a secure I-cache to mitigate I-cache side-channel attacks (section 4).
- Evaluation of the performance of Random-Fixed mapping scheme for instruction caches (section 5).

To the best of our knowledge, our work is the first work to analyze how to secure the I-cache against side-channel attacks. We discuss D-cache side channel attacks and defenses in section 2. We introduce I-cache attacks and our classification matrix in section 3. We characterize the effectiveness of randomized mapping schemes in 4, and discuss its impact on performance and power in section 5.

2. BACKGROUND

2.1 Attacks on Data Caches

The majority of cache side channel attacks are D-cache attacks. They rely on *secret-dependent memory indexing*, which can be found in many cryptographic algorithms. Listing 1 shows a code snippet in AES. It first XORs the input *in* with the round key *rk*, and then uses the result to index the AES tables (Te0,...,Te3). If the attacker can get the index to the lookup table, he can immediately get the key.

Listing 1: secret-dependent table lookup

```
s0 = GETU32(in      )^ rk[0];
s1 = GETU32(in + 4) ^ rk[1];
s2 = GETU32(in + 8) ^ rk[2];
s3 = GETU32(in + 12) ^ rk[3];
t0 = Te0[s0>>24]^Te1[(s1>>16)&0xff]
     ^Te2[(s2>>8)&0xff]^Te3[s3&0xff]^rk[4];
```

In order to infer the index of the victim's table lookup, the most common way is to exploit the cache contention (conflict misses) between the victim and the attacker. If an attacker discovers which cache set he can access to contend with the victim's table lookup, he can easily reverse engineer the memory address of the table lookup, due to the fixed memory-to-cache mapping of conventional caches. Prime-Probe and Evict-Time are two well-known attack techniques on a D-cache that can recover a secret key.

Prime-Probe Attack: This is an access-based attack since the attacker can infer the victim's memory accesses by measuring how these accesses impact the attacker's own data. It works as follows:

Prime: An attacker A fills one or more cache sets with its own data.

Idle: A waits for a prespecified Prime-Probe interval while the victim process V gains control of the processor and utilizes the cache.

Probe: A gains control of the processor again, and measures the time to access the same cache sets to learn V's cache activity.

If V uses some cache sets during the Prime-Probe interval, some of A's cache lines in these cache sets will be evicted, which causes a longer load time for those cache sets during A's Probe phase. It is straightforward to do "prime" and "probe" for the D-cache–the attacker only needs to load data from an array of the same size as the data cache.

Evict-Time Attack: This is a timing-based attack since the attacker can observe the total execution time of the victim's security-critical operation. The attacker A works as follows:

Evict: A fills one specific cache set with his own data.

Time: A triggers the victim process to perform the security-critical operation and measures the victim's total execution time.

If the victim accesses the evicted cache set, his execution time tends to be statistically higher than when he does not access the evicted cache set (but may access other data in the lookup table), due to the victim having a cache miss.

2.2 Defenses for D-cache Attacks

Several hardware solutions have been proposed to secure the D-cache against side-channel attacks [19, 20]. Compared with software countermeasures, hardware solutions can target the root cause of the attacks, and thus are more general, without degrading performance. *Cache partitioning* and *randomizing memory-to-cache mapping* are two general approaches to secure the data cache [19] against the contention based side-channel attacks. In this paper, we focus on the randomization based approaches since they are reported to have little or no performance degradation [19, 20], while many partitioning approaches have some performance degradation due to smaller effective cache sizes. The randomization based approaches allow cache contention, but the attacker cannot reverse engineer the memory address from a cache set since any memory address can be randomly mapped to a cache set. In particular, we are interested in Newcache [20], which has been shown to have the best performance among various randomization based approaches.

Newcache: We observe that the core mechanism of Newcache can be denoted a *Random-Fixed Mapping*. It introduces a level of indirection in the memory-to-cache mapping using the concept of a Logical Direct Mapped (LDM) cache (see Figure 1). The LDM cache does not physically exist and can be larger than the physical cache. Conceptually, a memory block to cache line mapping first uses part of the memory address, called index bits, to lookup the LDM cache as for a direct-mapped cache and then follows the mapping (if this exists) from the corresponding LDM cache entry to locate the physical cache line. The main guarantee for security is that the mapping from the LDM cache to the physical cache is fully-associative, randomized and can be changed dynamically. We use physical memory address to avoid aliasing issues.

Can we reuse the same design to secure other caches? This paper answers the open question: "*Is Newcache, proposed for D-caches, also suitable for securing I-caches from side-channel attacks targeting the I-cache?*"

3. I-CACHE ATTACKS

The answer to this question is by no means obvious because of the fundamental difference between I-cache and D-cache attacks. Unlike D-cache attacks which exploit the secret-dependent memory indexing of load and store instructions, I-cache attacks exploit the secret-dependent instruc-

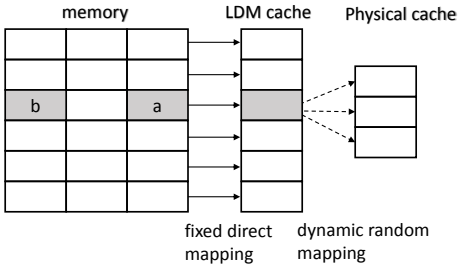

Figure 1: Random-Fixed Mapping in Newcache.

tion paths of instruction execution. Listing 1 illustrates a code segment with secret-dependent instruction paths. If the attacker can distinguish which code path is executed, he can immediately learn the secret. Although the implementation of the cipher tries to avoid any secret-dependent instruction paths, this is very hard to achieve, in practice, since the implementation also has to be optimized for performance. Also, secret-dependent instruction paths is not specific to cryptography, but can also be found in various non-cryptographic applications that process secrets like passwords, credit card information and personally identifying information.

Listing 2: secret-dependent instruction paths

```
if (secret==1) { code block 1; }
else           { code block 2; }
```

Before delving into the analysis of the effectiveness of the randomized mapping against I-cache attacks, we first study whether the D-cache attack techniques can be applied to I-cache attacks.

3.1 Prime-Probe Attack

Prime-Probe attacks are the only attacks that have been shown against the I-cache. However, a Prime-Probe attack for the I-cache is not as easy as that for the D-cache. First, priming or probing a specific set in the I-cache is more tricky than just loading data from an attacker's array. Second, derivation of victim cache usage requires distinguishing cache footprints left in the I-cache by different secret-dependent code paths, rather than just timing how long it takes to probe a D-cache set. An I-cache footprint consists of all the cache sets that the memory lines containing the code block (including all the functions called by it) maps to. The Prime-Probe attack enables the attacker to determine which cache sets have been accessed by the victim, and hence its cache footprint.

First, we show how to prime or probe exactly one set in the I-cache, as shown in Figure 2. This shows a cache with a size of S sets·W ways·B bytes, where B is the block size. Memory addresses that are $B \cdot S$ bytes apart will be mapped to the same cache set. To prime one cache set, the attacker needs to allocate W cache blocks (the shaded blocks in Figure 2), where each cache block is $B \cdot S$ bytes away from the previous block. Inside each cache block is the instruction **jmp B · S**, a relative jump instruction that jumps to an instruction that is $\mathbf{B \cdot S}$ bytes away, which is the next block, except for the last block, which has a return(**ret**) instruction. When priming the cache set, the attacker's main program jumps to the address of the first block. Then the W cache blocks will be fetched into the cache set one by one, and the last block **ret** returns to the main program. We can use the same mechanism to prime the remaining $S-1$ sets. The attacker can allocate a contiguous memory chunk that has the same size as the instruction cache to prime the whole I-cache. Probing is just like Priming, except that the **rdtsc** instruction, placed at the beginning and end of accessing each I-cache set, is used to measure the time to probe each I-cache set.

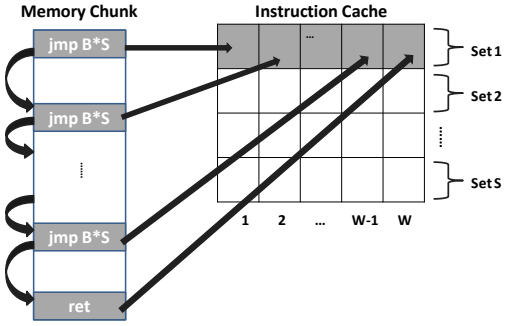

Figure 2: Instruction Cache Set Prime and Probe

As a concrete example, we show the Prime-Probe attack against many public-key algorithms. The private key in a public-private key-pair is used for longer-term digital identity, including digital signatures. Leaking the private key to an attacker can lead to masquerading attacks, fake server or client authentications, and fake signatures – causing very serious security breaches.

The main computation in many of these public-key algorithms, like RSA, El-Gamal, etc., is the modular exponentiation operation. To illustrate, we use the implementation of modular exponentiation in libgcrypt v1.5.3 [3], a widely used cryptographic library, and also used in the Cloud Computing I-cache attack[22]. It implements a square-and-multiply algorithm as shown in Algorithm 1. We label S, R, M to stand for calls to the functions, Square, ModReduce and Mult, respectively, inside the algorithm.

Algorithm 1 Square-and-Multiply Algorithm in libgcrypt

```
procedure SquareMul(base, expo, mod)
    Let e_n, ..., e_1 be the bits of expo, and e_n = 1
    for i ← n − 1, 1 do
        y ← Square(y, base)                    ▷ (S)
        y ← ModReduce(y, mod, base)            ▷ (R)
        if e_i = 1 then
            y ← Mult(y, mod, base)             ▷ (M)
            y ← ModReduce(y, mod, base)        ▷ (R)
        end if
    end for
end procedure
```

The sequence of function calls in one execution of modular exponentiation with square-and-multiply can leak information about the exponent *expo*, which is the private key of many public-key ciphers (ElGamal, RSA etc.). Note that the most siginificant exponent bit e_n is always 1, so the information leaked is from e_{n-1} to e_1. For example, the sequence **(SRMR)(SR)** corresponds to $expo = 110_2 = 6_{10}$. **SRMR** leaks information $e_2 = 1$, and **SR** gives $e_1 = 0$.

In a real attack on a real machine, the attacker and victim processes are run alternately via normal preemptive

scheduling. The attack is conducted on our lab testbed equipped with quad-core Intel Xeon E5-1410 processors and Linux kernel 3.5.0-43. Xeon E5-1410 has a $32kB$, 8-way Set-Associative (SA) I-cache with $64B$ block size, thus we have 64 sets for the I-cache. We use a method similar to [15] to trick the Linux complete fair scheduler (CFS) to get an attacker Prime-Probe interval of about $2.5\mu s$, so that basically one victim's M or S operation can execute during this interval. The victim just performs modular exponentiation depicted in Algorithm 1, with an exponent unkown to the attacker. Each time the attacker performs a Prime-Probe, he gets a cache footprint, which is a vector of timings, one timing per cache set. For our test setup, this is a 64-element vector. After getting the trace of the footprints online, the attacker can do off-line analysis to classify each footprint as an S, M or R operation, to infer the exponent key bits. For our purpose of evaluating the vulnerability of an I-cache to this attack, we label each S, R or M operation in Algorithm 1.

Cache Footprint Classifier. To quantitatively describe how well an operation can be correctly classified based on its cache footprint, we use libsvm[4] to train a multi-label SVM (Support Vector Machine) classifier. We choose its default SVM type C-SVC [13, 14] and a linear kernel function. An SVM is a supervised machine learning tool that, when fed with enough training samples with different labels, builds a model that can categorize new testing samples into one of the labels. For this case, a label is one of S, R and M; a sample is a cache footprint with feature dimension of 64, and the ith feature ($1 \leq i \leq 64$) is the probe time for cache set i. Thus, before an SVM can classify new instances, we need to train it with a set of sample-label pairs. Similar to [22], we set the victim's exponent to be all 1's, which gives a sequence of SRMRSRMR... operations. We also choose a training ratio of 1:1:2 for the operation types S:M:R to bias a little bit towards R, which avoids too many R misclassificatons. We collect a trace of the attacker's Prime-Probe footprints, and we use the hooking function trick from [22] to label those footprints. In the experiment, we use 40,000 of these footprints to train an SVM classifier, and another 12,000 samples to do the testing.

Figure 3 shows 6000 attacker's Prime-Probe trials collected by the attacker. To give a figure with visually clear patterns, we choose the first 2000 footprints with a S, M or R label and pile them together based on their labels. So trials 1-2000 are all S operations; 2001-4000 are all M operations and 4001-6000 are all R operations, and we can see different patterns for these three different operations.

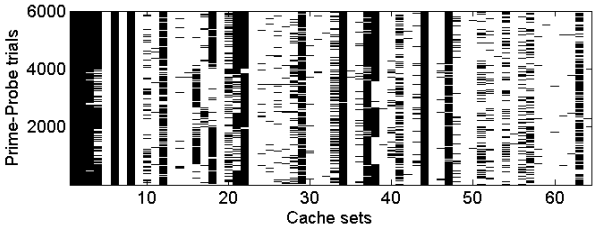

Figure 3: Cache footprint patterns for S, M and R operations

Table 1 shows the classification matrix for the attack. The **diagonal** entries are the correct classifications, and the classification accuracy is 90.2%. The high classification accuracy indicates a high key-bits recovery by subsequent off-line stages of the attack [22]. This classification matrix is more accurate than the visual patterns in Figure 3.

Table 1: SVM Classification Matrix for Real Attacks on Real Machines

	Classification			Classification Accuracy
	Square	Multiply	Reduce	
Op: Square	3479 (0.87)	189 (0.05)	332 (0.08)	90.2%
Op: Multiply	375 (0.09)	3587 (0.90)	38(0.01)	
Op: Reduce	92(0.02)	148 (0.04)	3760 (0.94)	

3.2 Evict-Time Attacks

There are no I-cache attacks exploiting the Evict-Time technique. Ideally, to apply the Evict-Time technique to the I-cache, the attacker can evict one specific cache set in one of the code paths, say, in code block 1. However, since code block 2 may be much longer, the total execution time is not necessarily higher even when the victim executes code block 1 and experiences a cache miss. Unlike the D-cache where each load takes more or less the same time under the same conditions, two code paths may have different lengths and thus take different execution times. In fact, one may argue that if the timing difference between two execution paths is different, the attacker can directly infer the secret from the total execution time, without needing the Evict-Time attack. Furthermore, Evict-Time attacks generally leak less information than Prime-Probe attacks for the I-cache. If the secret-dependent branch is within a loop (e.g. Square-and-Multiply exponentiation as shown in Algorithm 1), a Prime-Probe attack can capture each execution of the branch. But for an Evict-Time attack, the evicted line can only impact the first execution of that code path. After both code paths are cached, there will be no timing differences due to cache misses. In summary, access-based attacks (e.g. Prime-Probe attacks) that can capture the execution trace of a program are the main threats for the I-cache. Hence, we focus our analysis on the Prime-Probe attacks.

4. RANDOMIZE MEMORY-TO-CACHE MAPPING

The reason why the randomized mapping can defeat the D-cache attacks is that even though the attacker still has cache contention with the victim, he cannot infer the memory address of the victim's table lookup by observing the contention. Since I-cache attacks rely on differentiating the I-cache footprints of different execution paths, it is important to characterize whether or not the I-cache footprints are still distinguishable after randomized memory-to-cache mapping. We use the SVM classification matrix for this characterization.

4.1 FA cache with random replacement algorithm

We first analyze the memory-to-cache mapping scheme, achieved using a fully-associative (FA) cache with a random replacement algorithm.

We performed the attack using gem5 [7], a cycle-accurate simulator. gem5 is able to run a full operating system, libraries and applications, while modeling new hardware, enabling us to simulate our new I-cache hardware design and test real-world attacks. We use a two-level cache hierarchy

Table 3: Classification Matrices for 8-way SA Cache (LRU and random replacement policy) and FA Cache

	Classification (SA, LRU)			Classification (SA, random)			Classification (FA)			Accuracy		
	Square	Multiply	Reduce	Square	Multiply	Reduce	Square	Multiply	Reduce	LRU	random	FA
S	3985 (1.00)	0 (0.00)	15 (0.00)	3758 (0.94)	7 (0.00)	235 (0.06)	1058 (0.26)	996 (0.25)	1946 (0.48)	99.7%	93.2%	40.5%
M	1 (0.00)	3991 (1.00)	8(0.00)	5 (0.00)	3606 (0.90)	389 (0.10)	1108 (0.27)	1105 (0.28)	1787 (0.45)			
R	8(0.00)	6 (0.00)	3986 (1.00)	76 (0.02)	107 (0.03)	3817 (0.95)	705 (0.18)	599 (0.15)	2696 (0.67)			

Table 2: Baseline Simulator Configurations

Parameter	Value
L1 data cache associativity, and size	8-way SA, 32 KB
L1 instruction cache associativity, and size	8-way SA, 32 KB
L2 cache associativity, and size	8-way SA, 2 MB
Cache line size	64 B
L1 hit latency	1 cycle
L2 hit latency	20 cycles
Memory size, and latency	2 GB, 200 cycles

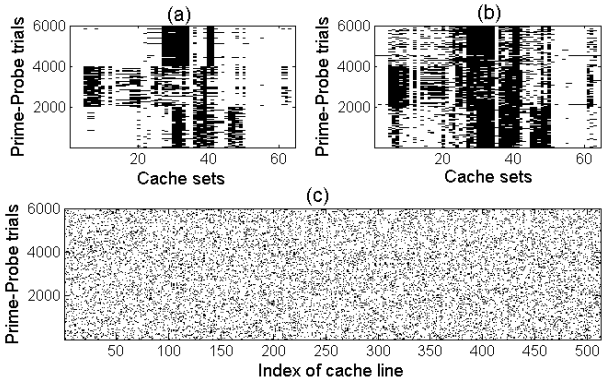

Figure 4: Cache footprint patterns of modular exponentiation on gem5 simulator, (a) 8-way SA cache with LRU replacement policy, (b) 8-way SA cache with random replacement policy, (c) Fully-associative (FA) cache with random replacement policy. Darker point represents longer time.

in our simulation with baseline configurations shown in Table 2. Similar to the real attacks we did on real machines, we set the victim process to repeatedly perform modular exponentiations. Since gem5 does not support a high precision timer, it is very hard to achieve fine-grained preemption as in the real machine. Hence we emulate the Prime-Probe attacks by hacking the simulator to execute dummy memory accesses for the probe operations at some fixed time interval. The time interval is chosen so that the victim can only run for a very short time interval, similar to the real attack.

The figures and SVM classification matrices are obtained in the same way as in section 3.1. Figure 4 shows the visual patterns of 6000 footprints, divided into 2000 footprints each for S, M and R. For comparison, we show results for an SA cache with LRU replacement (a) and random replacement (b). We can clearly distinguish the operations from each other in an SA cache with LRU replacement in Figure 4(a). In Figure 4(b), although introducing more noise than the LRU policy, the random replacement policy can only randomize eviction of cache lines within a set, but not over the entire cache. Hence the fixed, linear mapping between the memory addresses and cache sets is still maintained. However, no visual pattern can be found for the fully associative cache with a random replacement policy (Figure 4(c)).

Similar to section 3.1, we use these footprints as training and testing inputs to an SVM classifier. Note that instead of Priming each set in a 32kB 8-way SA cache, we Prime-Probe each cache line in the FA cache, which gives a feature dimension of 512 (i.e., SVM input vector of 512 elements). Table 3 gives the classification matrix for these three cache configurations. The numbers in parenthesis are probabilities adding up to 1 for each row. Random replacement within each set of an 8-way SA cache only degrades the SVM classification accuracy from 99.7% to 93.2%, while using randomization across a FA cache drops the classification accuracy to only 40.5%. According to [23], it would be extremely difficult for further offline analysis to extract the exponent key bits correctly under 40.5 % accuracy. Note that ideally, if the S, M, and R operations are completely indistinguishable, the classification accuracy should be 33.3% (instead of 0%). Our results show, both visually and quantitatively, that, the randomized mapping approach (in a fully-associative cache) is also effective against I-cache attacks. This is because the I-cache footprints, which are a set of cache sets that are used by a certain execution path, can contain arbitrary random cache sets due to the randomized memory-to-cache mapping.

4.2 Random-Fixed Mapping

We now consider the Random-Fixed Mapping scheme used by Newcache. Basically, Newcache can dynamically randomize the cache line eviction over the entire cache (like a FA cache), while using some static fixed mapping to reduce the cache access time and power consumption.

If we take a closer look at Figure 1, we can have a better idea why we denoted this a Random-Fixed Mapping.

Since the LDM cache can be larger than the physical cache, not every entry has a mapping to a physical cache line. If a cache access finds that there is no mapping from the corresponding LDM cache entry to a physical cache line, the LDM entry can establish a new mapping to any of the physical cache lines randomly (replacing the old mapping to this cache line). This is equivalent to randomly replacing a cache line over the entire cache.

Random-Fixed Mapping still allows the existence of some deterministic contention due to its fixed memory-to-LDM linear mapping; two memory addresses may map to the same LDM cache entry and thus share the same mapping to the physical cache. Consider the memory lines, a and b, in Figure 1. If the memory line a is already in the cache and the memory line b is not, when accessing the memory line b, b will replace the memory line a in a fixed way, since it will follow the random mapping previously established by a. When the size of the LDM cache increases, the impact of such fixed mappings will decrease. Specifically, when the size of the LDM cache is the same as the entire physical memory, the Random-Fixed Mapping becomes the fully-associative random mapping without any fixed linear mapping.

We perform the same Prime-Probe attack on Random-Fixed Mapping caches, as we did for the FA cache in Fig-

Table 4: Classification Matrices for Newcache with LDM size equal to 1X, 4X and 16X cachesize

	Classification (1X cachesize)			Classification (4X cachesize)			Classification (16X cachesize)			Accuracy		
	Square	Multiply	Reduce	Square	Multiply	Reduce	Square	Multiply	Reduce	1X	4X	16X
S	3978 (0.99)	0 (0.00)	22 (0.01)	3840 (0.96)	5 (0.00)	155 (0.04)	1143 (0.29)	940 (0.24)	1917 (0.47)	98.7%	96.4%	41.0%
M	1 (0.00)	3983 (1.00)	17 (0.00)	5 (0.00)	3850 (0.96)	145 (0.04)	1154 (0.29)	1098 (0.27)	1748 (0.44)			
R	10(0.00)	107 (0.03)	3883 (0.97)	62 (0.02)	61 (0.02)	3877 (0.97)	696 (0.17)	620 (0.16)	2684 (0.67)			

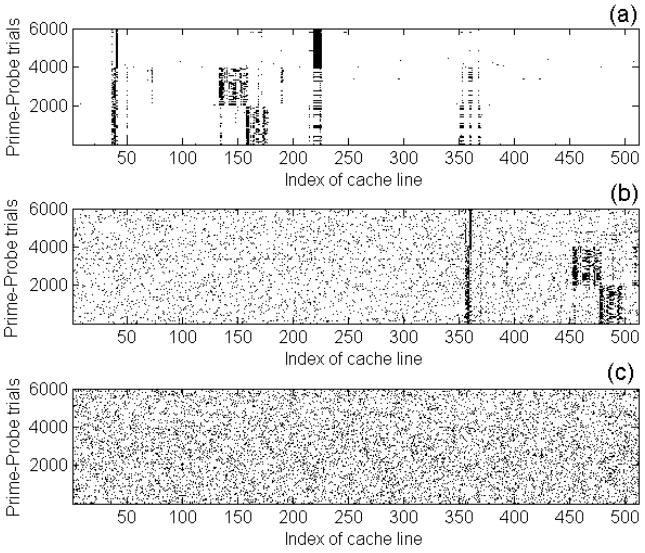

Figure 5: Cache footprint patterns of modular exponentiation on gem5 simulator for Newcache with different LDM cache sizes, LDM size equals (a) cachesize, (b) 4X cachesize, (c) 16X cachesize.

ure 4(c). The simulator configurations are the same as the baseline configurations in Table 2, except for the associativity of the L1 instruction cache. We varied the size of the ephemeral LDM cache to be 1X, 4X or 16X the size of the physical cache. We collect the Prime-Probe footprints, and input them as training and testing samples to an SVM classifier. Figure 5 gives the visual patterns of Prime-Probe trials similar to section 3.1 for these three cache configurations, and Table 4 gives the classification matrix.

When the size of the LDM cache equals the size of the physical cache, it is essentially a direct-mapped (DM) cache. This is why a clear pattern following the S, R, M operations can be seen in Figure 5(a). In this case, at the steady state, every entry in the LDM cache will have a mapping to a physical cache line and the mapping cannot be updated any more. Therefore, all the following memory accesses will follow the fixed, linear mapping. The amount of fixed, linear mapping decreases as the size of the LDM cache increases. When the LDM cache size is increased to 16 times the physical cache size, no clear pattern for the cache footprints can be seen (Figure 5(c)). In Table 4, the SVM classification accuracy decreases from 98.7% to 41.0%, as the size of the LDM cache increases from 1X to 16X the size of the physical cache. Comparing Figure5(b) and Table 4, we find that when the LDM cache = 4X physical cache size, a lot of noise is introduced into the cache patterns. But if the attacker's prime-probe code happens to get even a few index conflicts with the victim operations' codes, the fixed linear mapping

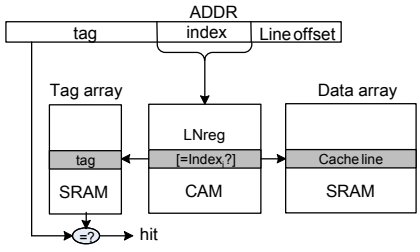

Figure 6: Holistic implementation of Newcache.

will give him a very high classification accuracy. By increasing the LDM cache size, we cannot eliminate the fixed mapping entirely, but we can reduce the probability of the index conflicts.

The reason the LDM cache can be larger than the physical cache is because once we support dynamic randomized mapping to the physical cache lines, we need to implement indirect mapping of memory to physical cache lines. This is shown as Line Number registers (LNregs) in Figure 6. Once there are LNregs, it is easy to increase the width of the LNregs, which essentially determines the size of the ephemeral LDM cache relative to the physical cache. If the physical cache contains 2^n cache lines, the width of LNregs can be $n + k$ where k is called the *number of extra index bits*, which means the LDM cache is 2^k larger than the size of the physical cache. Figure 5(a), (b) and (c) correspond to Newcache with $k = 0$, $k = 2$ and $k = 4$ extra index bits, respectively. These k extra cache index bits improve both security and performance.

5. POWER AND PERFORMANCE EVALUATION

We now study the performance impact of Newcache used as the I-cache, since it is likely to be different from that of the D-cache.

5.1 Holistic Hardware Implementation

Although conceptually Newcache introduces a level of indirection, we show that with clever, holistic hardware optimization across microarchitecture and circuit layers, it need not result in a longer cache access latency. Figure 6 shows the holistic implementation of Newcache. The row decoder in a Direct Mapped (DM) cache is replaced with a small content-addressable-memory (CAM) array, similar to the concept of an inverted page table. Each CAM entry functions as a Line Number register (LNreg), containing the index to the LDM cache. If there is a match, the corresponding row of the tag array and the data array are accessed in parallel. The tag array contains the remaining part of the address and some status bits. Newcache can be much faster and consume less power than the FA cache because only a small portion of the address is used for the CAM's associative search.

Table 5: Latency and power comparison

	8-way SA cache	FA cache	Newcache						
			k=0	k=1	k=2	k=3	k=4	k=5	k=6
Latency (ps)	1020	1190	924	938	938	952	952	966	966
Power (mW)	104	108	92	93	93	94	94	94	94

Power and Latency: We further study the power and latency of Newcache by implementing it in a testchip, using 65nm CMOS process. The cache size is 32 KB. We use post-layout extraction to obtain net lists of our prototype with accurate wire capacitances. The post-layout HSPICE simulation results for the access latency and power with different numbers of extra index bits k are summarized in Table 5. We also show the results for the 8-way SA cache and the FA cache. Table 5 shows that we can design a secure instruction cache that consumes less power than conventional SA caches, and in fact is even faster. The 8-way SA cache is more power-hungry than our secure cache because it has to read out all the 8 tags, but Newcache only needs to read out 1 tag, and most of the power of the cache is consumed by accessing the data and tag arrays. (Both caches read out only 1 data line.) The CAM power only contributes a small portion of the total power. Also, we used clever circuit level optimizations (for both Newcache and FA cache), e.g.,hierarchical NAND-type CAM, which detects match instead of mismatch, and consumes much less power than conventional NOR-type CAM.

5.2 System Performance

Table 6: Summary of workloads

Workload	Description
bzip2	file compression
gcc	gcc compiler
povray	image rendering
h264ref	video compression
xalancbmk	XML processing
perlbench	perl interpreter
apache [1]	Web server Client: apache benchmark Send 1000 https request with concurrency of 10
ffserver [2]	Streaming server Client: openRTSP [5] Send 30 different remote connection requests to the ffserver for media file streaming
tomcat [8]	Java application server Client: apache benchmark Send 10 requests each for 11 URLs

We now study the performance impact of Newcache used as the I-cache using the gem5 simulator. The baseline simulator configurations are the same as Table 2. We picked six workloads from the SPEC CPU 2006 benchmark suite [6], and three real-time server workloads for our performance evaluation (See Table 6).

Most of the SPEC CPU 2006 benchmarks do not have large instruction working sets – we picked six with relatively larger instruction working sets. We picked some server workloads as well because they usually have larger instruction working sets. The server workloads are widely used real-world Cloud Computing workloads.

Figure 7 shows the instruction misses per kilo instructions (MPKI) for SA caches and Newcache with different lengths of extra index bits, k. The results are normalized to the MPKI of the 8-way SA cache with LRU replacement

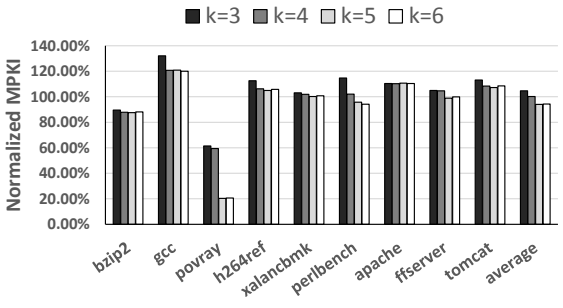

Figure 7: Normalized I-cache Misses Per Kilo Instructions (MPKI) for Newcache with different lengths of extra index bits, k. Lower is better.

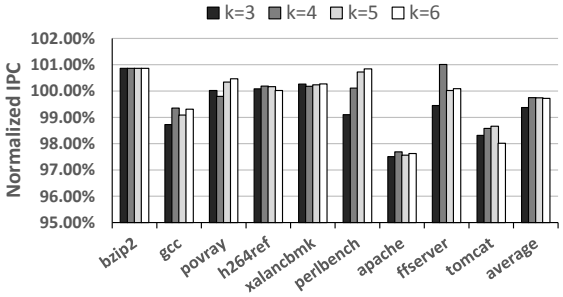

Figure 8: Normalized Instructions Per Cycle (IPC) for Newcache with different lengths of extra index bits, k. Higher is better.

policy – typical of today's L1 caches. Newcache has about the same MPKI performance as the conventional 8-way SA cache. For some benchmarks like **bzip2**, **povray** and **perlbench**, Newcache has better performance. On average, the variation is within 10%. Increasing the length of the index bits may reduce the miss rate, but the impact is small when further increasing the number of extra index bits, k, beyond 5. (Compare Figure 5 where k=4 is sufficient for security.)

Figure 8 shows that the impact to overall performance, in terms of instructions per cycle (IPC), is even smaller than for the I-cache MPKI. This is because the absolute instruction misses are very low, and the use of a non-blocking cache allows instruction fetching during a cache miss, hiding some miss latency. The largest performance degradation is less than 2.5% (**apache**). Some workloads (**bzip2, povray, perlbench**) can benefit from the reduced I-cache MPKI and improve the overall performance by 0.8%. On average, the performance degradation is less than 0.3% when k is larger than 3.

6. RELATED WORK

While no hardware defenses have been proposed, some

software defenses for I-cache attacks have been proposed. Much effort involved writing side-channel resistant cryptographic algorithms without secret-dependent execution paths. For example, in the modular exponentiation in libgcrypt v1.5.3, the multiplication is performed regardless of the value of the exponent bit, if the exponent is stored in the secure memory (which needs to be explicitly allocated by the applications). However, this tends to degrade the performance by at least 20% for a random exponent. In general, writing side-channel resistant cryptographic algorithms tends to be ad-hoc, i.e., different for each cryptographic algorithm, and usually suffers from large performance penalties. Moreover, the I-cache attacks are also applicable to non-cryptographic settings where there are few mechanisms for preventing side-channel information leakage.

A compiler is also able to remove key-dependent instruction paths either through predicated execution or if-conversion. The basic idea is to fetch instructions in both code paths and instructions with predicates equal to zero will be nullified during execution. However, only very few architectures (e.g., Intel IA-64) support full predication, and if-conversions may cause significant performance degradation. Furthermore, lack of access to source code typically prevents recompilation of legacy applications.

7. CONCLUSIONS

We analyzed cache side-channel attacks on the I-cache. We detailed how to perform the Prime-Probe attack on I-caches. We analyzed why timing attacks on the I-cache are not serious threats, whereas access based side-channel attacks are. We proposed to use a classification matrix to quantitatively characterize the vulnerability of an instruction cache to the access based side-channel attacks. We then applied an SVM classification matrix to study whether the randomized mapping approach, proposed for the D-cache, can be used to defeat side-channel attacks on the I-cache. In particular, we analyzed the core mechanism of Newcache – which we call the Random-Fixed Mapping scheme. We find that randomized mapping indeed works well for securing an I-cache against contention-based Prime-Probe attacks, without degrading performance. Hence, a Newcache replacement for both the I-cache and D-cache in processors will prevent cache side-channel attacks without degrading either system performance or cache physical performance.

Acknowledgment

This work was supported in part by DHS/AFRL grant FA8750-12-2-0295. We thank Ken Mai and his students for their help in the Newcache testchip circuit implementation, from which we extracted and interpolated the latency and power numbers for the different cache configurations.

8. REFERENCES

[1] Apache. http://www.apache.org/.
[2] ffserver. https://www.ffmpeg.org/ffserver.html.
[3] libgcrypt. http://www.gnu.org/software/libgcrypt/.
[4] libsvm. http://www.csie.ntu.edu.tw/ cjlin/libsvm/.
[5] openRTSP. http://www.live555.com/openRTSP/.
[6] SPEC CPU 2006. http://www.spec.org/cpu2006/.
[7] The gem5 Simulator System. http://www.gem5.org.
[8] tomcat. http://tomcat.apache.org/.
[9] O. Aciiçmez. Yet Another Microarchitectural Attack: Exploiting I-cache. In *ACM Workshop on Computer Security Architecture*, pages 11–18, October 2007.
[10] O. Aciiçmez, B. B. Brumley, and P. Grabher. New Results on Instruction Cache Attacks. In *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems (CHES'10)*, pages 110–124, 2010.
[11] D. J. Bernstein. Cache-timing Attacks on AES. Technical report, 2005.
[12] J. Bonneau and I. Mironov. Cache-Collision Timing Attacks against AES. In *Proceedings of Cryptographic Hardware and Embedded Systems (CHES'06)*, pages 201–215, 2006.
[13] B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 144–152, New York, NY, USA, 1992. ACM.
[14] C. Cortes and V. Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, Sept. 1995.
[15] D. Gullasch, E. Bangerter, and S. Krenn. Cache Games — Bringing Access-Based Cache Attacks on AES to Practice. In *Proceedings of IEEE Symposium on Security and Privacy (SP'11)*, pages 490–505, 2011.
[16] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: the Case of AES. In *Proceedings of The Cryptographers' Track at the RSA conference on Topics in Cryptology (CT-RSA'06)*, pages 1–20, 2006.
[17] D. Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *IACR Cryptology ePrint Archive*, page 169, 2002.
[18] C. Percival. Cache Missing for Fun and Profit. In *Proc. of BSDCan*, 2005.
[19] Z. Wang and R. B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of ACM/IEEE International Symposium on Computer Architecture (ISCA'07)*, pages 494–505, 2007.
[20] Z. Wang and R. B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO'08)*, pages 83–93, 2008.
[21] Y. Yarom and K. Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. Cryptology ePrint Archive, Report 2013/448, 2013.
[22] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 305–316, 2012.
[23] Y. Zhang and M. K. Reiter. Duppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*, pages 827–838, 2013.