Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks

LEONID DOMNITSER, State University of New York at Binghamton
AAMER JALEEL, Intel Corporation, VSSAD
JASON LOEW, NAEL ABU-GHAZALEH, and DMITRY PONOMAREV, State University of New York at Binghamton

We propose a flexibly-partitioned cache design that either drastically weakens or completely eliminates cache-based side channel attacks. The proposed Non-Monopolizable (NoMo) cache dynamically reserves cache lines for active threads and prevents other co-executing threads from evicting reserved lines. Unreserved lines remain available for dynamic sharing among threads. NoMo requires only simple modifications to the cache replacement logic, making it straightforward to adopt. It requires no software support enabling it to automatically protect pre-existing binaries. NoMo results in performance degradation of about 1% on average. We demonstrate that NoMo can provide strong security guarantees for the AES and Blowfish encryption algorithms.

Categories and Subject Descriptors: C.1.0 [**Processor Architectures**]: General

General Terms: Design, Security, Performance

Additional Key Words and Phrases: Side-channel attacks, shared caches, secure architectures

ACM Reference Format:
Domnitser, L., Jaleel, A., Loew, J., Abu-Ghazaleh, N., and Ponomarev, D. 2012. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. ACM Trans. Architec. Code Optim. 8, 4, Article 35 (January 2012), 21 pages.
DOI = 10.1145/2086696.2086714 http://doi.acm.org/10.1145/2086696.2086714

1. INTRODUCTION

In recent years, security has emerged as a key design consideration in computing and communication systems. Security solutions center around the use of cryptographic algorithms, such as symmetric ciphers, public-key ciphers, and hash functions. The strength of modern cryptography makes it infeasible for the attackers to uncover the secret keys using brute-force trials, differential analysis [Biham and Shamir 1991] or linear cryptanalysis [Matsui 1994]. Instead, almost all known attacks today exploit weaknesses in the physical implementation of the system performing the encryption, rather than exploiting the mathematical properties of the cryptographic algorithms themselves.

This material is based on research sponsored by the Air Force Research Laboratory under agreement number FA8750-09-1-0137 and by National Science Foundation grants CNS-1018496 and CNS-0958501. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies and endorsements, either expressed or implied, of Air Force Research Laboratory, National Science Foundation, or the U.S. Government.
Contact author's address: D. Ponomarev, Computer Science Department, State Univeristy of New York at Binghamton, Binghamton, NY 13902-6000; email: dima@cs.binghamton.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2012 ACM 1544-3566/2012/01-ART35 $10.00
DOI 10.1145/2086696.2086714 http://doi.acm.org/10.1145/2086696.2086714

1.1. Side Channel Attacks

A subtle form of vulnerability in the physical implementation of otherwise secure systems is the leakage of information through unintended (or side) channels. The leaked information is called *side channel information*, and attacks exploiting such information are called *side channel attacks* [Bernstein 2005; Kong et al. 2009; Wang and Lee 2007; 2008; Osvik et al. 2005; Tromer et al. 2009; Percival 2005; Bangerter et al. 2011; Side 2009]. Examples of side channels include execution time, power consumption, heat, electromagnetic radiation, or even sound level emanating from a device [Backes et al. 2010]. A large number of side channel attacks have been successfully demonstrated against a range of software and hardware security mechanisms, breaking many cryptographic systems including block ciphers (such as DES, AES, Camellia, IDEA, and Misty1), stream ciphers (such as RC4, RC6, A5/1, and SOBER-t32), public key ciphers (such as RSA-type ciphers, ElGamal-type ciphers, ECC, and XTR), signature schemes, message authentication code schemes, cryptographic protocols, and even the networking subsystems [Side 2009]. Thus, it is critical to build systems that are immune to side channel attacks.

Early side channel attacks were used to break specialized systems such as smart cards. However, side channel attacks that exploit the shared resources in conventional microprocessors have been recently demonstrated [Bangerter et al. 2011; Wang and Lee 2007, 2008; Kong et al. 2009; Tromer et al. 2009; Osvik et al. 2005; Percival 2005; Bernstein 2005; Bonneau and Mironov 2006; Canteaut et al. 2006]. Such attacks are extremely powerful because they do not require the attacker's physical presence to observe the side-channel and can therefore be launched remotely using only legal non-privileged operations. Cache-based software side-channel attacks represent one example of this attack class. The cache accesses performed by a cryptographic process can be monitored by another process sharing the cache (which observes its own cache miss pattern), leaking information about the secret key. Cache-based attacks do not require the attacker's physical presence to observe the side channel, so they can be launched remotely using only legal non-privileged operations.

1.2. Cache-Based Side Channel Attacks

Recent literature classifies cache-based attacks into three groups: *time-driven attacks*, *trace-driven attacks* and *access-driven attacks* [Aciicmez and Koh 2006]. Trace-driven attacks are based on obtaining a profile of cache activity during the encryption and deducing which cache accesses generated by the encryption process resulted in a hit [Aciicmez and Koh 2006]. Time-driven attacks measure the run time of a victim process to deduce information about the secret key [Bernstein 2005; Bonneau and Mironov 2006; Kong et al. 2009; Osvik et al. 2005; Tromer et al. 2009]. The critical insight is that self-contention within the cache varies with the key and the input data, leading to differences in the execution time that can be used to infer the key. These attacks are limited and are relatively easy to circumvent (more details in Section 2). In contrast, access-driven attacks defy straightforward defense mechanisms, because the attacker actively uses the shared cache simultaneously with the victim process, and can monitor the victim's cache behavior in detail [Percival 2005; Osvik et al. 2005; Tromer et al. 2009; Kong et al. 2009; Wang and Lee 2007, 2008; Bangerter et al. 2011]. More specifically, the attacker can force the victim to experience cache misses and observe them through the side channel. In this type of attack, the attacker can glean detailed memory access information, such as the cache sets accessed by the victim program, making the side channel more informative and the attack more dangerous. For example, a recent work [Bangerter et al. 2011] demonstrated an access-driven attack on AES that

recovers the secret key after monitoring 100 encryptions followed by a 3-minute long offline analysis phase. In this paper, we only address this type of attack.

Existing algorithmic solutions (reviewed in Section 7) provide limited protection against access-driven attacks. Architectural solutions [Wang and Lee 2007, 2008; Kong et al. 2009] can shut down the side channel, but require some modifications to the cache design and peripheral circuitry, which may hamper adoption in practice. In addition, most of the hardware schemes proposed so far require OS, compiler, ISA or programming language support. This support is often needed to explicitly mark the critical data (data that needs protection) and limit the scope of protection only to critical cache lines in order to reduce the negative performance impact.

This paper presents a novel approach to mitigating cache-based side channel attacks that is designed purely in hardware and requires no support from the ISA or software layers. The proposed approach is extremely simple, consisting of a small modification to the cache replacement logic, leaving the performance-critical core circuitry of the cache unchanged. The proposed approach provides side-channel security while resulting in minimal performance losses.

1.3. Proposed Solution: Non-Monopolizable (NoMo) Caches

We propose a low-complexity hardware-only design that protects against access-driven cache-based attacks. The proposed solution, called Non-Monopolizable (NoMo) cache, is a simple modification to the cache replacement policy that restricts an attacker to using no more than a predetermined number of lines in each set of a set-associative cache. As a result, the victim's data in the protected cache line(s) of each cache set cannot be replaced by the attacker, preventing the attacker from observing those memory accesses through the side channel. NoMo caches significantly weaken a common class of cache-based side channel attacks in a low-complexity manner, without OS, compiler, programming language or ISA support, and with minimal or no performance degradation to applications sharing the cache. By controlling the sharing restrictions, NoMo designs can tradeoff performance and security. We demonstrate that even minimal cache sharing restrictions provide strong security benefits for AES and Blowfish encryption algorithms, while in the limit NoMo provides static non-overlapping cache partitioning, closing the side channel completely.

Every cache side channel attack involves two phases: the data collection phase through the side channel and the subsequent off-line analysis of this data for the key reconstruction. The secret key reconstruction process is difficult: the side-channel often leaks only some information that is contaminated with noise (information that does not correlate with the key). Despite this fact, in practice, keys have been successfully recovered from side-channel information on different platforms [Tromer et al. 2009; Bangerter et al. 2011]. Thus, the security of the system depends critically on the quantity of information about the key that can be extracted from the side channel [Kopf and Basin 2007; Standaert et al. 2006]. As will be demonstrated in this paper, NoMo significantly reduces the information leakage through the side channel, making key reconstruction (already a difficult process), virtually impossible.

1.4. Contributions

The main contributions and the key results of this paper are the following.

—We propose a new variation of access-driven cache-based attack that exploits the knowledge of underlying cache replacement policy and significantly reduces the number of cache accesses needed by the attack.
—We propose the Non-Monopolizable (NoMo) cache design, a low-complexity hardware approach to mitigate access-driven cache-based side channel attacks. NoMo requires

simple changes to the existing cache design and has minimal performance impact. An attractive feature of the NoMo design is that it does not require any support from the OS, ISA, compiler or programming language.

—We describe variations of the NoMo design, which are defined by the cache sharing restrictions (degree of NoMo). These variations define a spectrum of security-performance trade-offs, where some schemes completely eliminate the side channel at higher performance cost, while others dramatically diminish the side channel at a slightly higher cost to performance.

—We evaluate the security characteristics of NoMo caches using AES and Blowfish encryption and decryption algorithms as examples.

Our performance studies show that for a 32KB 8-way set-associative L1 D-cache[1]: (1) NoMo-2 cache, where applications sharing cache are restricted to use at most 6 out of the 8 ways in each set, only allows 0.6% of critical cache accesses to be observed through the cache side channel. The average performance impact on simulated SPEC 2006 benchmarks is 0.5% (maximum 3%); (2) NoMo-3 completely shuts down the cache side channel for AES and has 0.007% leakage of critical data for Blowfish, but incurs an average performance loss of 0.8% (maximum 4%) for SPEC 2006 benchmarks; (3) NoMo-4 design (non-overlapping static partitioning) provides a complete isolation of applications sharing the cache and thus eliminates cache-based side channel in principle, but that comes at the expense of 1.2% (5% maximum) performance loss on the average. All these designs are attractive choices for secure caches.

The remainder of the paper is organized as follows: in Section 2 we review the AES and Blowfish cryptographic algorithms and existing cache-based attacks. Section 3 presents our assumptions and threat model. In Section 4 we describe the Non-Monopolizable cache design. Section 5 describes our performance and security evaluation methodology, followed by our results in Section 6. We review related work in Section 7, and Section 8 offers our concluding remarks.

2. BACKGROUND AND IMPROVED ATTACK

In this section we describe the AES and Blowfish cryptographic algorithms, classify cache-based side channel attacks, and present the threat model assumed for our study. We also present a variation of an access-driven attack that significantly reduces the number of cache accesses needed by the attacker.

2.1. The Advanced Encryption Standard (AES)

AES, the Advanced Encryption Standard, is a widely used symmetric block cipher. It encrypts and decrypts 128-bit data blocks using either a 128-, 192-, or 256-bit key. Each block is encrypted in 10 rounds of mathematical transformations. To achieve high performance, AES implementations use precomputed lookup tables instead of computing the entire transformation during each round. The indexes to these tables are partially derived from the secret key, thus by detecting the cache sets accessed by the victim (through the side channel observations), the attacker can derive some information about parts of the secret key. By using multiple measurements, the entire key can be successfully reconstructed. The version of the AES code that we use in this study [Daemen and Rijmen 2002] employs five tables (1KB each) for both encryption and decryption. The first four tables are used in the first nine rounds of encryption/decryption, and the fifth table is used during the last round. Separate sets of tables are used for encryption and decryption. More details on the AES encryption algorithm and specific side channel attacks on AES can be found in [Tromer et al.

[1]This cache configuration is representative of Intel's Core i7 processor.

2009; Bangerter et al. 2011]. In summary, the AES table lookups constitute critical cache accesses, and their observation through the cache-based side channel can lead to secret key discovery.

2.2. The Blowfish Encryption

Blowfish is a keyed, symmetric block cipher, included in a large number of cipher suites and encryption products. Blowfish has a 64-bit block size and a variable key length from 32 up to 448 bits. It is a 16-round Feistel cipher and uses large key-dependent S-boxes. Again, just in the case with AES, the accesses to S-boxes are the critical accesses that can reveal the key-related information through the cache side channel.

2.3. Cache-Based Attack Variations

The original idea that cache memory could be used as a side channel during the run of a cryptographic algorithm was proposed by Kelsey et al. [1998]. A comprehensive taxonomy of software cache-based side channel attacks is presented in [Osvik et al. 2005; Tromer et al. 2009; Kong et al. 2009]. Attacks can be categorized into three groups: trace-driven, time-driven and access-driven [Aciicmez and Koh 2006].

Trace-Driven Attacks. In trace-driven attacks, the attacker obtains the detailed cache activity profile during the encryption [Aciicmez and Koh 2006; Zhao and Wang 2010]. This profile includes the outcomes of every memory access issued by the cipher in terms of cache hits and misses. Therefore, the adversary is capable of observing if a particular access to a lookup table yields a hit and can infer information about the lookup indices, which are key-dependent. Consequently, the attacker can use this information to derive the secret key. The limitation of these attacks is that they require access to very detailed profiling information.

Time-Driven Attacks. Time-driven attacks [Tsunoo et al. 2002, 2003; Bernstein 2005; Osvik et al. 2005; Bonneau and Mironov 2006; Canteaut et al. 2006; Kong et al. 2009] are less restrictive than trace-driven attacks, they exploit the relationship between inputs and execution time to deduce information about the secret key. Bernstein [2005] demonstrated a successful attack on AES exploiting the variance in the execution time (due to internal cache interference of the AES algorithm itself) for different inputs. Bonneau and Mironov [2006] further optimized the attack by exploiting cache collision information and relationship between the number of cache collisions (table accesses that hit into the cache) and the execution time.

Time-driven attacks have several shortcomings that make it challenging to implement them in practice. The attack demonstrated by Bernstein [2005] was carried out using a pristine environment (although sophisticated coding techniques that filter out the noise can be used to alleviate this requirement). Another limitation is that the attack requires references of encryption timings for different secret keys in an identical system configuration. The number of such measurements is exponential in the size of the key; although this is a one time setup effort that would allow the extraction of multiple keys from the same environment, it is very expensive. In addition, the attack relies on the ability to execute the timing code synchronously before and after the encryption. Finally, the attack relies on the overall execution time, which is a coarse grained measure shared by many keys (that must then be evaluated to identify the correct one). Since this overall time is measured externally over complex encryption services with various overheads, there is likely to be substantial variability in timing for the same key, substantially increasing the overhead of key recovery [Osvik et al. 2005; Tromer et al. 2009].

Thus, while timing-driven attacks are possible, it is challenging to mount them in practice. In this paper, we target the third class of attacks: access-driven attacks, which

are easier to launch and allow the attacker to extract significantly more information from the cache side channel (e.g., cache sets accessed by the references to critical data and even their order). The richer side-channel exploited by these attacks makes them extremely dangerous and practical [Bangerter et al. 2011]. We describe access-driven attacks in the next section.

Access-Driven Attacks. Access-driven attacks defy straightforward defense mechanisms, because the attacker runs simultaneously with the victim process and manipulates the usage of the shared cache. This directly impacts the cache behavior of the victim process [Percival 2005; Osvik et al. 2005; Wang and Lee 2007, 2008; Tromer et al. 2009; Kong et al. 2009; Bangerter et al. 2011]. The easiest way to launch such an attack is by running the attacker alongside the encryption process on a simultaneously multithreaded processor [Percival 2005; Osvik et al. 2005; Tromer et al. 2009], but it can also be done in a single-threaded environment with appropriate support from the OS [Osvik et al. 2005; Tromer et al. 2009]. The attacker does not need the knowledge of either a plaintext or a ciphertext (note that this is in contrast to traditional cryptanalytic scenarios, where such knowledge is necessary) and it only times its own cache accesses, instead of timing the victim's activity, which is much easier.

If the caches are completely shared, then the attacker can monopolize the entire cache and cause the victim to miss into the cache and evict the attacker's data. When the attacker later experiences its own cache miss, it can determine the set number of the victim's cache access, which resulted in the replacement of the attacker's cache line. From that information, it can derive the lookup table indices and therefore deduce parts of the secret key. This attack is much more powerful than the time-driven attack, because the attacker can obtain the actual order of accesses (if his cache traversal rate is aggressive enough), in addition to the information of which specific table entries were used. Furthermore, the attack can also be synchronized with the boundaries of individual block encryptions. This type of attack was shown to successfully uncover the secret key in the popular OpenSSL implementation of the RSA encryption algorithm [Percival 2005] and also was used to break the AES key [Osvik et al. 2005; Tromer et al. 2009]. Access-driven attacks can be based on collecting individual cache accesses [Percival 2005] or even the frequencies of accesses to individual cache sets [Tromer et al. 2009]. A recent effort [Bangerter et al. 2011] demonstrated a practical access-driven attack that recovers secret AES key in about 3 minutes. Figure 1 shows the C code of a simple access-driven attack. In summary, while the predefined memory-to-cache mapping remains the root cause of the attack [Wang and Lee 2007, 2008], the ability of the attacker to monopolize the cache and cause the victim's evictions makes the access-driven attack far more dangerous.

2.4. Proposed Replacement-Aware Attack Optimization

In the basic access-driven attack [Percival 2005; Osvik et al. 2005; Tromer et al. 2009], the attacker fills the entire cache by accessing all cache blocks. This opens the attack to defenses that observe the rate of cache filling by each process. Moreover, the fact that the whole cache must be continuously visited means that the attacker can only sample accesses to each block at a fairly coarse granularity.

In this section, we propose an optimization to this attack that exploits the knowledge of the cache replacement policy to significantly reduce the number of cache accesses needed by an attacker to check the cache contents. This allows the attack to proceed faster, allowing the attacker to more efficiently monitor the victim's accesses. The optimized attack can substantially improve the attack efficiency, especially if the cache access time is high (for example, as future attacks on lower level caches are devised).

```
#define ASSOC      8
#define NSETS    128
#define LINESIZE  32

#define ARRAYSIZE (ASSOC*NSETS*LINESIZE/sizeof(int))
static int the_array[ARRAYSIZE];

int fine_grain_timer(); //implemented as inline assembly

void time_cache() {
    register int i, time, x;
    for(i = 0; i < ARRAYSIZE; i++) {
        time = fine_grain_timer();
        x = the_array[i];
        time = fine_grain_timer() - time;
        the_array[i] = time;
    }
}
```

Fig. 1. C code for a simple access-driven attack.

The optimized attack relies on the knowledge of the replacement policy to avoid having to access every block in each set of the cache.

Without loss of generality and to simplify the description, we assume an LRU replacement policy. Its important to note that the optimized attack works for all deterministic replacement policies including most policies used on modern microprocessors. In warming up the cache, the attacker accesses the blocks in the set to make the replacement target block (RTB) the same for each set. After warming up the cache, the attacker proceeds by timing the RTB in each set. On a cache hit, the attacker knows that the set has not been accessed and moves to the next set. On a cache miss, the attacker recognizes the victim's access. In this case, the attacker walks the full set to discover if additional accesses were made by the victim. Also, the RTB is adjusted to point to the line to be replaced next for the next round of checks. Note that if the replacement policy is non-deterministic (such as random cache replacement proposed by RPCache [Wang and Lee 2007] and NewCache [Wang and Lee 2008] and implemented by ARM and Loongson processors [ARM 2011; Zhou 2010]), the attacker cannot use this optimized attack and has to rely on a code similar to that shown in Figure 1.

The optimized attack (when it is possible) reduces the number of cache accesses that are needed by the attacker by a factor of the cache associativity for the sets that are not accessed by the victim within an iteration of the attack. The victim typically accesses only a few sets in each iteration due to locality and due to excessive cache misses experienced as a result of the attacker's behavior. A faster attack increases the amount of information that can be obtained by the attacker through the cache side-channel. Moreover, the optimized attack can defeat potential defense mechanisms that are based on controlling the rate of filling the cache. We use this modified attack in our experiments in Section 6.

3. THREAT MODEL, ASSUMPTIONS AND SOLUTION SCOPE

We make the following assumptions with respect to the attack and the attacker's capabilities.

—We consider an access-driven cache side channel. As we describe in Section 2, these are the most difficult attacks to protect against due to the attacker's ability to extract detailed information through the side-channel.
—The attack is mounted on the set-associative L1 data cache of a Simultaneously Multithreaded (SMT) processor. While it is possible, in principle, to launch a side-channel attack on shared L2/L3 caches in a multi-core processor, such an attack is complicated by the fact that most memory accesses are filtered by the private L1 caches and not visible to the attacker unless the attacker exploits the properties of inclusive caches. The size of the cache also challenges the ability of the attacker to capture leaked accesses. L2/L3 attacks and defenses are beyond the scope of this paper.
—We consider an SMT processor with only two threads (which is typical of most existing SMT designs). In general, our technique applies to all situations where the number of threads does not exceed the number of cache ways (which is typical at the L1 cache level for out-of-order processors). We only consider set-associative caches in this study, as they are the norm in today's systems. Our proposal does not apply to direct-mapped caches, but those are rarely used in performance-sensitive designs. Finally, while access-driven attacks are also possible in single-threaded processors, they require OS support [Tromer et al. 2009] and are therefore more challenging to launch. Since the proposed technique is based on cache partitioning, it does not apply as such to superscalar designs.
—We assume that an attacker is synchronized with the cryptographic process at the granularity of individual block encryption. That is, the attacker determines which sets were accessed for each block. This assumption represents a worst case scenario where the attacker uncovers the most information through the side-channel. Such an attack scenario is realistic: an attack can be performed synchronously, in which the attacker triggers the victim's cryptographic operation. For example, the attacker may trigger a server or file system operation that performs encryption or decryption [Tromer et al. 2009]. We note that this assumption is not fundamental to the feasibility of the proposed defense, our goal here is to evaluate security against the worst-case attack scenario.

4. NON-MONOPOLIZABLE CACHES

Access-driven attacks, outlined is Section 2, depend on the ability to evict victim's data from the cache. To thwart such an attack, we propose the Non-Monopolizable cache (NoMo cache in the rest of the paper), a minimally restrictive partial partitioning scheme, which prevents one application (simultaneously co-scheduled on an SMT processor) from monopolizing all lines in any set of a shared cache. NoMo logic is implemented as a simple modification to the replacement policy, so the core cache circuitry need not be reengineered. NoMo is a purely hardware scheme requiring no support from the operating system, compiler, ISA or programming language.

4.1. NoMo Overview

When operating in the NoMo mode, a cache enforces the NoMo invariant: *a running thread is guaranteed to have at least Y lines exclusively reserved in each cache set*. Feasible values of Y are in the range of $[0, \lfloor \frac{N}{M} \rfloor]$, where N is the associativity of the cache, and M is the number of SMT thread contexts. We call Y the degree of non-monopolization, or simply the NoMo degree. A NoMo cache configuration is referred to as NoMo-Y, with NoMo-0 being traditional unconstrained cache sharing, NoMo-$\lfloor \frac{N}{M} \rfloor$ being non-overlapping even cache partitioning, and intermediate configurations representing various levels of sharing flexibility, trading off performance and security.

If a victim (encryption) process uses no more than Y lines in each set, a NoMo-Y cache does not allow any accesses to be observed through the side channel, because the attacker cannot evict any of the Y lines reserved for the victim. If the victim uses more than Y lines in at least some sets, some information can still leak, but the side channel information leaked is dramatically degraded due to the NoMo filtering effect.

NoMo caches require no changes to the cache datapath, and involve only simple changes to the cache controller to implement the NoMo mechanism as a simple extension to the cache replacement policy. In the following subsections, we describe the NoMo implementation and analyze the implications of this design on the attack itself and the side channel leakage resulting from the attack. We consider both a naive attacker and an attacker that is aware of the presence of NoMo.

4.2. Implementation of NoMo Caches

A NoMo design has two major components: replacement logic and mode transition logic. We describe each of these components in this section.

4.2.1. NoMo Replacement Logic. NoMo replacement logic, integrated with an existing replacement policy, forms the basis for protection. We propose way-partitioning to satisfy the NoMo invariant where we statically reserve a subset of cache lines in a set for each application. For example, if two applications are sharing an 8-way cache with a NoMo-2 policy, then two cache ways, say 0 and 1, can be statically reserved for the first application, and two other ways, say 2 and 3, can be statically reserved for the second application. The remaining four ways are dynamically shared.

Static way-partitioning is simple to implement; each thread maintains an N-bit reservation vector (where N is the cache associativity). Each bit in the reservation vector corresponds to a way in the cache. Only if a bit in the reservation vector is set can a thread allocate lines to that way. The logic to implement reservation vector simply amounts to excluding reserved lines from the replacement. This additional logic is minimal and amounts to a single byte per thread for an 8-way cache.

4.2.2. Mode Transition Initiation Logic. When only a single thread is executing, or when security is not a concern, the NoMo policy is not necessary and the cache can be used in unrestricted mode. NoMo mode is switched on when multiple threads start using the cache simultaneously. To determine which threads are actively using the cache, a small M-entry table is maintained (one entry per thread context). Each entry in this table has two fields: process ID and cache access counter. The process ID field stores a virtual address space identifier of a process that owns the corresponding table entry. Note that it is the same identifier that is stored with the cache tags to distinguish the cache lines belonging to different processes. The cache access counter field indicates the number of cycles that have elapsed since the last cache access was performed by this thread. This counter is incremented every cycle and is reset to zero every time the thread accesses the cache. When the counter saturates it is assumed that the corresponding thread is inactive.

The proposed NoMo entry detection mechanism, avoids explicit OS/ISA control of the cache access mode. Thus, NoMo can be implemented entirely at the hardware level.

4.2.3. Mode Transition and Gang Invalidation. When an inactive thread accesses the cache, or when a process ID is changed (during a context switch), a NoMo entry procedure is performed to ensure that no initial leakage occurs when NoMo mode is enabled. During a NoMo entry, Y lines are invalidated in every cache set. We elect to carry out the invalidation up front for all sets, rather than when each set is accessed, to provide immediate leakage protection using simple gang-invalidation circuitry. Without this

gang-invalidation the attacker can detect initial accesses by the new thread, as it observes misses to its data residing in the now exclusive sets as the new thread accesses the same ways. Such initial leakage was shown to provide significant data for the attacker [Kong et al. 2009]. During a NoMo exit, reservations are cleared. The clearing of the reserved ways in this fashion does not have any significant impact on performance, because it happens infrequently, at the granularity of context switches. We performed experiments clearing the entire L1 cache every 30 milliseconds for a processor with 2GHz frequency, and observed only 0.1% loss in performance on the average across our benchmark mixes.

4.3. NoMo-Aware Attacks and Implications on Side Channel

The NoMo design naturally thwarts a straightforward access-driven attack (an attack spanning the entire cache), because the attacker simply cannot access the reserved ways of the victim process, and therefore it would experience a cache miss on at least Y out of every N accesses to the same set. These misses will be encountered in every set, thus exposing no useful information. We assume that the attacker is aware of the NoMo defense mechanism, and modifies the attack to access only the number of lines that it is permitted to access under the NoMo rules. To simplify exposition, and without loss of generality, we assume a 4-way set-associative cache with LRU replacement policy and a NoMo-1 cache.

The NoMo-aware attack performs repetitive accesses to a large array in an effort to cover the entire cache. When the attacker encounters its first cache miss, it is impossible to distinguish whether this miss is a result of an actual access by the victim or is simply an artifact of cache way reservation. Since way reservation affects all sets, no useful information can be determined from these initial misses.

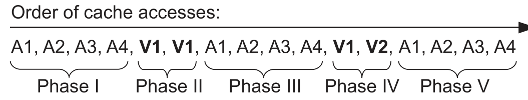
At this point, knowing that at least one of the ways is always reserved for the victim, the attacker can switch the attack mode to repetitively access n-1 instead of all n-ways in each set. Alternatively, the attacker may use the optimized attack described in Section 2.4 where it repetitively accesses the sets in the order of their expected replacement; for LRU, the attacker accesses the least recently used way, then the next one on the LRU stack and so on. This optimized attack minimizes the number of cache accesses needed, by taking into account the knowledge of the replacement policy.

On a miss, the attacker detects an intervening access to the set by the victim. It is possible for the attacker to uncover some of victim's accesses when the victim uses more than the reserved NoMo ways in a particular set. In practice, this form of leakage does not happen often, as we demonstrate in the results section. The main reason is that when the victim hits into the single cache block (within the targeted set) that it owns, or misses into the cache, but replaces the only block that it owns, the attacker gleans no information from this activity by the victim.

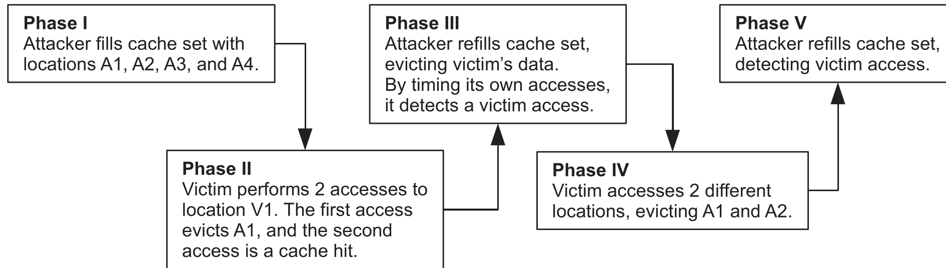
4.4. Example of NoMo Defense

Figure 2 shows an example scenario of a side channel attack. We depict accesses to one set of a 4-way cache. The attacker is able to evict all of the victim's data and capture information about the victim's accesses. Victim accesses that hit into the cache are not exposed, but then the initial access is still exposed. Note that the cache replacement policy used in this figure is LRU.

In Figure 3 we show an equivalent attack attempt on a NoMo-1 cache. The attacker accesses are not exactly the same as in Figure 2 because the attacker is necessarily aware that it is attacking a NoMo cache. As long as the victim process stays within the bounds of its reserved way, the attacker cannot evict its data, and nothing is exposed. Partial exposure is still possible when the victim uses the shared cache ways. For simplicity, these figures show NoMo-1 on a 4-way cache, but our experiments use an



(a) A stream of accesses to a single cache set, *S*. A1, A2, A3, and A4 are 4 different locations in the attacker's memory space that map to set *S*, and their presence in the time-line indicates attacker accesses. Likewise, V1 and V2 are victim memory locations mapping to set *S*, and represent victim accesses. Each phase I through V is a logical grouping of sequential accesses by attacker or victim.



(b) A descriptive timeline of each phase of the attack.

	Way 0	Way 1	Way 2	Way 3
Contents of set *S* after Phase I	A1	A2	A3	A4
Contents of set *S* after Phase II	V1	A2	A3	A4
Contents of set *S* after Phase III	A4	A1	A2	A3
Contents of set *S* after Phase IV	A4	V1	V2	A3
Contents of set *S* after Phase V	A2	A3	A4	A1

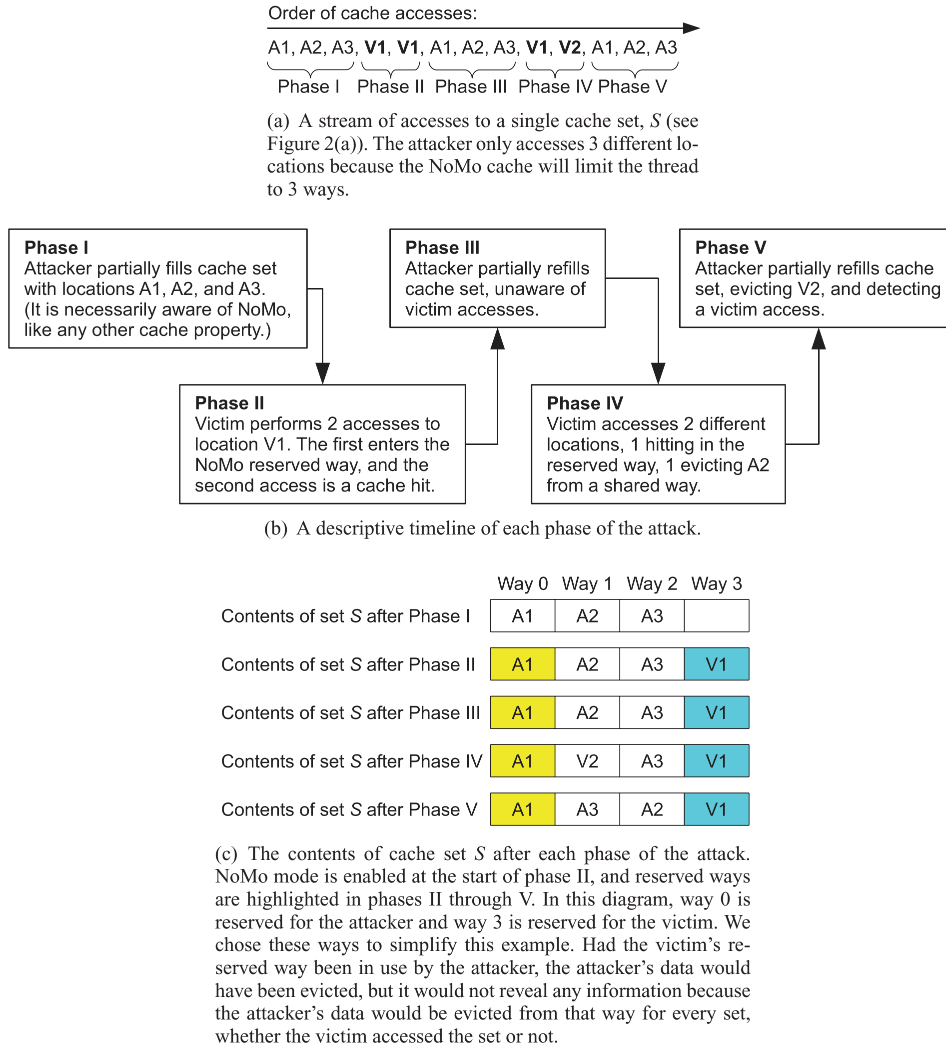
(c) The contents of cache set *S* after each phase of the attack.

Fig. 2. Example of side channel attack on traditional 4-way cache. LRU replacement is used.

8-way cache, and we show that NoMo-2 or higher is preferred. Again, the replacement policy assumed is the LRU, which explains the movement of some of the addresses across the cache blocks.

4.5. NoMo Scalability

When the NoMo design is extended to more than 2 threads, we can protect against a possible attacker that uses several colluding processes, where neither one monopolizes a set by itself. In this case, the NoMo invariant needs to be strengthened: *a process may use up to* $N-(A-1)Y$ *lines in a set*, where the new variable A is the number of threads actively sharing the cache. In our 2-threaded model, there is a binary NoMo mode, but with more threads, the threshold for set monopolization would vary with the number of active threads. Again, as mentioned in the threat model section above, we are not concerned with the situation where the number of threads exceeds the number of cache ways, NoMo design in its basic form is not applicable to such scenarios and additional considerations need to be taken into account. The exploration of these opportunities is part of our future work.



(a) A stream of accesses to a single cache set, *S* (see Figure 2(a)). The attacker only accesses 3 different locations because the NoMo cache will limit the thread to 3 ways.

(b) A descriptive timeline of each phase of the attack.

(c) The contents of cache set *S* after each phase of the attack. NoMo mode is enabled at the start of phase II, and reserved ways are highlighted in phases II through V. In this diagram, way 0 is reserved for the attacker and way 3 is reserved for the victim. We chose these ways to simplify this example. Had the victim's reserved way been in use by the attacker, the attacker's data would have been evicted, but it would not reveal any information because the attacker's data would be evicted from that way for every set, whether the victim accessed the set or not.

Fig. 3. Example of side channel attack on 4-way NoMo-1 Cache. LRU replacement is used.

4.6. Operating System and Instruction Set Architecture Support

While NoMo caches do not require any operating system and/or ISA support, they can further benefit if such support can be provided. For example, a system call and the corresponding ISA support could be added to dynamically adjust the NoMo degree. While allowing a process to decrease the NoMo degree opens the cache to attack, a process could request that a higher NoMo degree is used when it is running. The operating system could also adjust the timeout for exiting the NoMo mode, or provide hints about process activity. Finally, an application can request the use of NoMo caches through a system call. Again, we emphasize that while all this support can benefit the security-performance trade-offs of NoMo, it is not required. In fact, the evaluations in this paper assume no such support and still demonstrate very attractive trade-offs.

4.7. NoMo and Multithreaded Workloads

When multiple threads belonging to the same process are active together, partitioning schemes can lead to cache coherence issues as different copies of the same data exist in different partitions. However, NoMo is simply a reservation mechanism that affects the replacement policy. Consider two threads A and B belonging to the same process. When thread A accesses a memory location that was previously accessed by B, it will find it in the cache, even if it is located in the cache way reserved for B by NoMo: NoMo does not interfere with normal cache operation. In this way, NoMo is critically different from cache partitioning schemes [Suh et al. 2001; Qureshi and Patt 2006], which must either do the partitioning at the process level, or solve the resulting cache coherence issues.

5. EVALUATION METHODOLOGY

We used a Pin [Luk et al. 2005] based trace-driven x86 simulator for our performance studies. Our baseline system is a 2-way SMT processor with an 8-way issue width, a 128-entry reorder buffer and a three-level cache hierarchy. The L1 instruction and data caches are 8-way 32KB each while the L2 cache is unified 8-way 256KB. The last-level cache (L3) is a unified 16-way 2MB cache. All caches in the hierarchy use a 64B line size. This memory hierarchy is reminiscent of modern high performance processors, such as Intel's Core i7 [Nehalem 2009]. For replacement decisions, all caches use the LRU replacement policy. The load-to-use latencies for the L1, L2, and L3 caches are 1, 10, and 24 cycles respectively. We model a 250 cycle penalty to main memory and support a maximum of 32 outstanding misses to memory.

For our studies we use 15 representative SPEC CPU2006 [Spradling 2007] benchmarks compiled using the icc compiler with full optimization flags. The following benchmarks were used: astar, bzip2, calculix, dealII, gobmk, h264ref, hmmer, libquantum, mcf, perlbench, povray, sjeng, sphinx, wrf, xalancbmk. Traces of representative regions for these benchmarks were collected using PinPoints tool [Pinpoints 2009]. We ran all possible two-threaded combinations of the 15 SPEC benchmarks, i.e. 105 workloads. We simulated 250 million instructions for each benchmark. Simulations continue to execute until all benchmarks in the workload mix execute at least 250 million instructions. If a faster thread finishes its 250M instructions, it continues to execute to compete for cache resources. However, we only collect statistics for the first 250 million instructions of each application.

We report performance results using two widely-used metrics for multithreaded workloads. The first metric is the cumulative IPC throughput of two co-executing benchmarks, and the second metric is fairness [Luo and Franklin 2001].

We evaluated the security properties of NoMo using two popular cryptographic algorithms, AES [Daemen and Rijmen 2002] and Blowfish [blowfish 2009], obtained from the MiBench security benchmark suite [mibench 2009].

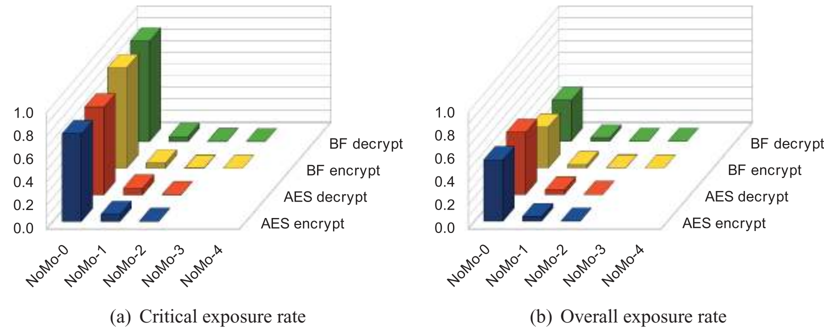
To evaluate the security of NoMo cache, we simulated the execution of an idealized attacker (as a separate thread) alongside an encryption or decryption process. We used AES and Blowfish encryption and decryption on 3 million blocks of truly random input [random 2009] and simulated them to completion. The large number of input blocks was used to detect the worst-case scenarios with our proposed designs.

6. RESULTS AND DISCUSSIONS

In this section we evaluate security and performance characteristics of NoMo cache.

6.1. Leakage Evaluation

We start by describing memory accesses performed by AES and Blowfish programs, and introducing terminology. Typical implementations use extensive lookups of



(a) Critical exposure rate (b) Overall exposure rate

Fig. 4. Aggregate exposure rates.

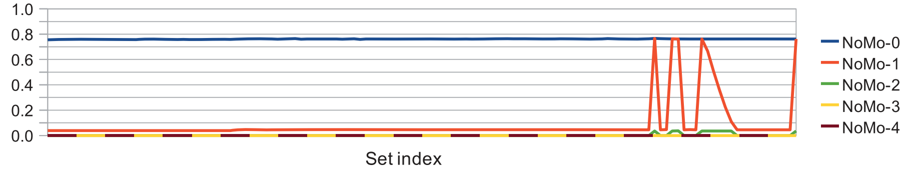
precomputed data tables. For AES, the tables used in rounds 1 through 9 of encryption or decryption occupy 4KB of contiguous memory space. Another set of tables, also 4KB, are used during the 10th and final round. Blowfish uses 4 1KB tables and an 18-entry (72 byte) array during all 16 rounds of encryption or decryption.

The indices used to access these tables are derived from the secret key. A cache-based side channel attacker can observe accesses to these tables, and can reconstruct the key from the leaked portions of memory addresses. Because these table accesses can be used by side channel attacks, we call them *critical accesses*, and the tables themselves, *critical data*. When an access is observed by an attacker, it is an *exposure*, and a critical access that an attacker observes is a *critical exposure*.

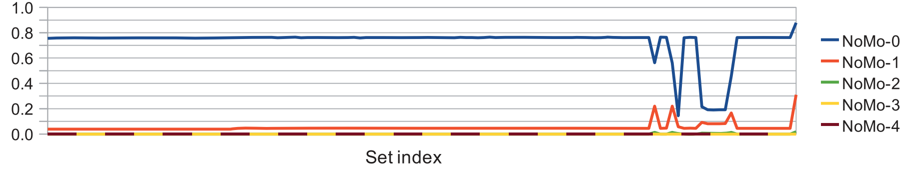
As part of our idealized attack scenario, we present cache access statistics from the start of encryption or decryption of a 16-byte data block to the end of the block operation, so noise from peripheral work is not captured. Our input data consisted of 3 million randomly generated blocks. We perform our experiments on an 8-way cache, varying the degree of NoMo between 0 (normal, fully-shared cache) and 4 (static non-overlapping partitioning).

Figure 4 shows the rate of exposure under all NoMo modes, averaged across all blocks. The critical exposure rate is the rate of critical exposures out of all critical accesses, and the overall exposure rate is the rate of all exposures out of all cache accesses. The critical exposure rate is higher than the overall rate, meaning that critical data is especially likely to be exposed. The baseline (NoMo-0) critical exposure rate for AES encryption is 75.8%. NoMo-1 reduces this rate to 6.1%, NoMo-2 to 0.2%, and NoMo-3 prevents all exposure. (NoMo-4, which fully partitions the cache, always guarantees 0 exposure.) Blowfish encryption shows a 87.0% baseline critical exposure rate, dropping to 4.5% for NoMo-1 and 0.3% for NoMo-2. Blowfish also has negligible non-zero exposure with NoMo-3. Decryption results are omitted, as they are almost identical to encryption.

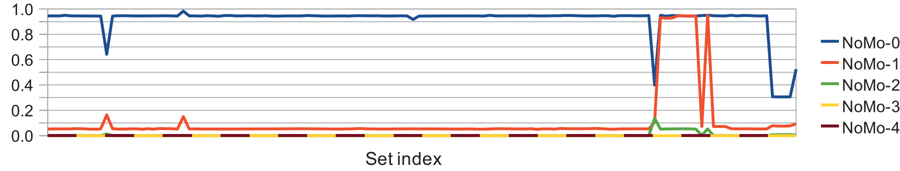
Although some critical accesses can be exposed with NoMo, exposures tend to occur in the same cache sets, and accesses to other sets are entirely filtered. NoMo-1 has critical exposures in all sets (this does not mean that the same number of exposures occur as without NoMo, just that all sets have *some* critical exposures). NoMo-2 filters most sets, and NoMo-3 filters all sets for AES, and all but one set for Blowfish. Figure 5 presents a spatial analysis of exposure rates. The graphs show exposure rates calculated only for accesses to an individual set, from set index 0 to 63. The general trend is of high exposure rates in all NoMo-0 cases, though there are dips for some sets. Exposure rate with NoMo-1 stays low, but above 0, with occasional spikes. NoMo-2 eliminates exposure in most sets, with small spikes in some sets. NoMo-3 and 4 rates overlap at 0 for all sets, though some there is minimal exposure in 1 set for Blowfish under NoMo-3.



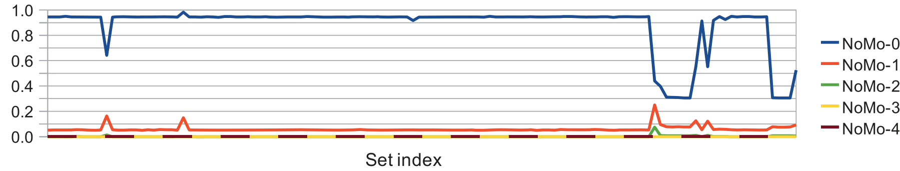
(a) Average exposure rate of critical data, by set, AES encryption



(b) Average exposure rate of all data, by set, AES encryption



(c) Average exposure rate of critical data, by set, Blowfish encryption



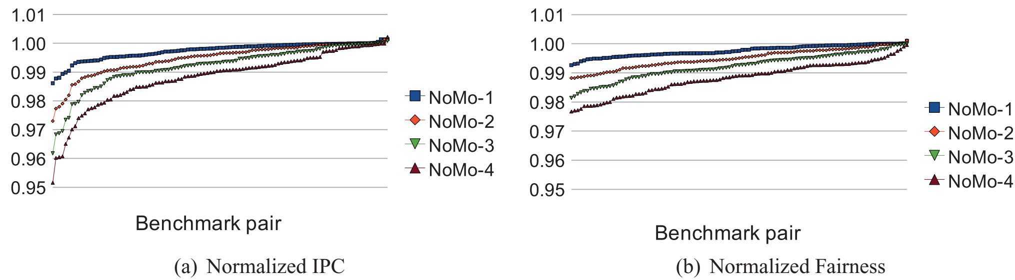
(d) Average exposure rate of all data, by set, Blowfish encryption

Fig. 5. Exposure rate in each of 64 cache sets, for all NoMo degrees.

Across all data blocks, our NoMo-2 cache has non-zero exposure rates in 10 of 64 sets for AES encryption, 14 for AES decryption, and 22 for Blowfish encryption and decryption. With NoMo-3, AES exposes nothing, and Blowfish occasionally exposes critical accesses, only in 1 set.

In addition to average exposure rates, we look at the maximum number of exposures possible in a single block operation, to see if some pathological inputs cause high exposure. For AES encryption on a baseline cache, the worst-block exposure rate is 68.9%. The worst block on a NoMo-1 cache had 10.9%, 1.6% with NoMo-2, and no exposure with NoMo-3 or 4. Blowfish encryption worst-block critical exposure rates, for NoMo-0 through 4, are 45.1%, 9.4%, 1.8%, 0.2%, and 0. Again, similar decryption numbers are omitted. Such pathological cases are a small fraction of all randomly-generated blocks that we considered in this study.

Since NoMo-2 and NoMo-3 designs filter all accesses to the majority of sets, information about accesses to most critical data is never exposed through the side channel. Additionally, all NoMo configurations hide the majority of critical accesses. For these reasons, NoMo-2 and NoMo-3 designs would require high brute-force cryptanalysis effort after a side channel attack.



(a) Normalized IPC (b) Normalized Fairness

Fig. 6. NoMo performance for 105 SPEC benchmark pairs.

6.2. Performance Evaluation

Figure 6 shows instruction throughput for 105 2-threaded workloads, each consisting of 2 benchmarks from the SPEC CPU2006 suite. The graphs show performance of NoMo-1 through 4, normalized to the baseline case (that is, NoMo-0 performance is 1). Each point on the graph represents one of the workloads, and they are sorted by increasing performance. We show performance as a simple IPC figure (committed instructions per cycle) and as fair throughput ("fairness"), which is the harmonic mean of weighted IPC.

As expected, increasing the degree of NoMo generally hurts performance. We demonstrate a continuum of performance options up to static partitioning that depend on the security requirement. On average, NoMo-1 decreases performance by 0.2%, NoMo-2 by 0.5%, NoMo-3 by 0.8%, and static partitioning by 1.2%. We see similar averages for the fairness metric, but lower worst-case degradation. By reserving part of the cache for each thread, NoMo prevents starvation, so we sometimes see better fairness performance than absolute performance. For a few workloads, NoMo performance reaches and slightly exceeds baseline performance. In general, performance and fairness losses are within 1.2% for all configurations. The highest performance degradation for an individual benchmark pair was observed with static partitioning (5%).

7. RELATED WORK

Both software and hardware solutions to address cache-based side channel attacks have been previously proposed.

7.1. Software Approaches

On the software side, the main idea is to rewrite the code of the encryption algorithms such that known side channel attacks are not successful. Examples of such techniques include avoiding the use of table lookups in AES implementations, preloading the AES tables into the cache before the algorithm starts, or changing the table access patterns [Osvik et al. 2005; Brickell et al. 2006; Tromer et al. 2009; Side 2009].

A limitation of software solutions is that they are tied up to a specific algorithm and attack, do not provide protection in all cases, are subject to programming errors, and often result in significant performance degradation [Wang and Lee 2007]. Another recent approach is to dedicate special functional units and ISA instructions to support a particular cryptographic algorithm. An example of this approach is the Intel AES instruction [Gueron 2008]. Another example is the support for general-purpose parallel table lookup instructions and permutation instructions that also speed up AES and other algorithms [Lee and Chen 2010].

7.2. Hardware Approaches

In response to the limitations of software solutions, several hardware supported schemes have been recently introduced. A partitioned cache was proposed [Page 2005],

along with ISA changes to make the cache a visible part of the architecture. This scheme requires changes to both the ISA and the cache hardware design and can lead to significant performance degradation.

Several alternative cache designs for thwarting cache-based attacks have been proposed by Wang and Lee [2007, 2008]. Partition-Locked Cache (PL cache) design [Wang and Lee 2007] uses cache line locking to prevent evictions of cache lines containing critical data, thus closing the side channel. The PL cache can lead to some cache underutilization, as the locked lines cannot be used by other processes, even after they are no longer needed by the process that owns them. In addition, the PLcache requires system support to control which cache lines should be locked.

Another effective side-channel protection technique proposed in Wang and Lee [2007] is the Random Permutation Cache (RPcache) that implements a randomization based approach to attack mitigation. A key operation performed by the RPcache is the permutation of memory addresses to cache sets. When the two cache sets are swapped (to realize permutation), the contents of their valid lines are invalidated, resulting in additional cache misses. To limit this effect, the OS support mechanism (similar to that used in the PLcache design) is needed. Finally, Wang and Lee [2008] proposed a novel cache architecture (called NewCache) with security-aware replacement algorithm. It uses a direct-mapped cache as an underlying substrate, but augments it with dynamic memory-to-cache re-mapping and longer cache index. While achieving security comparable to RPcache in Wang and Lee [2007], Newcache further improves cache performance for access time and cache misses. In contrast to the earlier schemes and similar to NoMo, NewCache does not require any support from the system software and/or the ISA. In addition, NewCache represents a more general solution to cache side-channel attacks than NoMo, because it applies to direct-mapped caches as well as set-associative caches, protects against the attacks on single-threaded cores and does not depend on the attack that monopolizes the cache. However, NoMo is a simpler design to implement for the typical Intel's microprocessor caches.

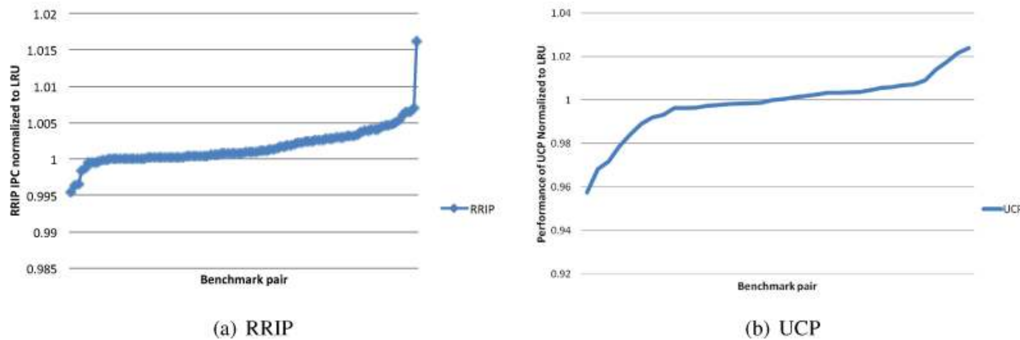
The technique proposed by Keramidas et al. [2008] introduces randomly selected cache decays to create caches with non-deterministic behavior and thus non-deterministic leakage through the side channel for the same input data. However, the use of small decay intervals needed to impact the aggressive attacks that we consider in this paper would lead to performance degradations.

In summary, prior efforts on L1 cache security rely on schemes that explicitly make an attacker incapable of identifying *any* critical cache access—this requirement was the underlying foundation of these designs. Consequently, the proposed schemes require substantial changes to the cache design. As a result, the integration of these designs into existing highly optimized cache data-paths can be challenging. Additionally, most of these techniques require OS, PL, compiler and/or ISA support to determine which memory lines need to be locked or permuted, to limit the impact of the proposed modifications only to the critical lines.

NoMo differs from prior art targeting cache security in that it is a low-overhead solution that can be readily adopted into the existing cache designs of modern high performance processors. We minimize design complexity by not explicitly identifying and hiding all critical cache accesses from a potential attacker. Instead, we use cache management mechanisms to probabilistically limit the amount of critical cache accesses that are exposed through the cache side channel.

7.3. Comparison with Prior Cache Partitioning Works

NoMo is a cache partitioning scheme targeting security. However, a number of previous efforts developed cache partitioning techniques targeted for performance [Suh et al. 2001; Qureshi and Patt 2006; Xie and Loh 2009; Jaleel et al. 2010]. In this subsection,



(a) RRIP (b) UCP

Fig. 7. IPC of recent cache partitioning schemes applied to L1 cache normalized to LRU. Each point on these graphs represents one benchmark pair.

we place NoMo design in the context of these prior cache partitioning efforts, notwithstanding the fact that those were designed for improving the performance of last-level caches.

With respect to their implications on security, cache partitioning schemes can be categorized into two groups. Techniques in the first group provide isolated cache partitions for multiple applications, for example by giving each application a predetermined number of cache ways [Qureshi and Patt 2006]. These solutions provide security in the steady state, after the partition boundaries are established, but do not address the side channel leakage occurring during the training period (when the proper partition sizes are determined) and also during the reconfiguration periods. An advanced attacker can potentially exploit these weaknesses. Prior work [Kong et al. 2009] demonstrated that if the initial phase is not protected, then enough side-channel leakage can occur to recover the secret key.

Techniques in the second group instead rely on cache pseudo-partitioning [Xie and Loh 2009; Jaleel et al. 2010]. Pseudo-partitioning provides flexible partition boundaries between the applications, sometimes allowing one application to replace the data placed in the cache by another application. While these approaches may have performance benefits due to soft-sharing of the cache, they are fundamentally vulnerable to side-channel attacks. Specifically, the attacker, exploiting knowledge of the cache pseudo-partitioning algorithm, can generate access patterns that will cause it to replace the victim's entries with high probability, which enables them to form the side-channel.

An additional point that we would like to emphasize is that most of these previous partitioning works were proposed for the last-level caches, where they were shown to provide a clear performance benefits over LRU replacement. However, these schemes do not outperform LRU replacement at the level of L1 cache. This is because the LRU captures well the high locality of references exhibited at the L1 level. For example, we studied the performance of a representative of the two partitioning groups (UCP [Qureshi and Patt 2006], representing isolated partitioning, and RRIP [Jaleel et al. 2010] representing pseudo partitioning) when used at the level of L1 cache. Figure 7 shows their performance on a number of two-benchmark mixes taken from the SPEC 2006 benchmark suite. UCP generally performed slightly worse than LRU, while RRIP provided marginal performance improvement over LRU. For many of the benchmark pairs that we considered, RRIP performed almost identical to LRU.

The primary reason for the observed behavior is that these partitioning schemes rely on the filtering of spatial/temporal locality from smaller caches to make intelligent replacement/partitioning decisions. The ample amount of spatial and temporal locality at the L1 cache causes intelligent replacement policies implemented at the L1 cache to

behave very similar to LRU. Therefore, since the performance benefits are not present, there is no motivation to apply performance-driven partitioning schemes designed for the last-level cache to the L1 cache. Applying them will only increase the complexity and adversely impact the L1 cache access time. The complexity of NoMo is indeed significanty lower that that of the L2/L3 partitioning designs. NoMo only requires a simple tweak to the LRU replacement policy to avoid replacing reserved lines, maintains way reservation vectors (1 bit per way per thread), and needs a simple logic to gang-reset the valid bits of the reserved ways. Just for comparison, the implementation of utility-based partitioning [Qureshi and Patt 2006] requires utility monitoring circuitry (8-bytes per line overhead for two threads sharing the cache), the partitioning algorithm, and the modification to the LRU policy to take partitioning into account. Other schemes used at the L2/L3 level also feature more complex logic and algorithms than NoMo. Again, this higher complexity is more affordable at the lower cache levels.

In summary, if the goal is to enhance the security of the L1 caches through partitioning, it is important to consider security directly, at the same time incurring as little redesign of the existing caches as possible. This is the approach taken by the NoMo proposal.

7.4. Implications on Key Reconstruction

Despite the presence of a number of a solutions to side-channel problems that do not perfectly close the channel [Goubin and Patarin 1999; May et al. 2001], the security properties of side-channels and the effectiveness of such imperfect solutions were open questions. Micali and Reyzin were the first to present a theoretical analysis of side-channel attacks [Micali and Reyzin 2004]. Using general assumptions, this model defines the notion of an abstract computer and a leakage function that together can capture almost all instances of side channels. However, the overly general assumptions make it difficult to apply this analysis to particular algorithms (e.g., DES or AES) or for specific side-channels.

Standaert et al. [2006] started from the Micali and Reyzin model and specialized it for more practical situations. Specifically, they restricted some of the assumptions to a range that corresponds to relevant adversary and leakage models. Moreover, they show how to map the abstract computational model to physical instances such as circuits and operations. Although this model brings the original model by Micali and Reyzin [2004] closer to practice, it models the leakage and adversary abstractly using information theoretic principles. Kopf and Badin [2007] investigate a similar information theoretic model.

8. CONCLUDING REMARKS

We proposed non-monopolizable caches, a family of flexibly partitioned cache designs which provides protection against cache-based side-channel attacks without significant cache redesign, without OS, programming language, compiler or ISA support, and with minimal performance impact. The NoMo variations considered in this paper provide a range of solutions with different performance to information leakage tradeoffs. NoMo-4 (static partitioning) design eliminates the side channel for all applications at the average performance cost of only 1.2%. The other configurations provide more flexible partitioning of the cache, where some sets remain in contention between the two threads. However, leakage is significantly reduced. For example, NoMo-2 leaks only 0.6% of critical accesses for AES, and 1.6% of critical accesses for Blowfish at an expense of 0.5% to the IPC throughput and 0.5% to fairness. Although the leakage of critical data is non-zero in this case, it is very small.

In summary, we show that simply partitioning the L1 cache among competing threads provides sufficient isolation and security at a minimal performance loss for

the access-driven cache side channel attacks considered in this paper. NoMo ideas may be easier to apply to existing microprocessor caches than the changes suggested by prior hardware designs.

ACKNOWLEDGMENTS

We thank all the anonymous reviewers for their useful insights related to this work. We also thank Mehmet Kayaalp for his help in improving the paper.

REFERENCES

ACIICMEZ, O. AND KOH, C. 2006. Trace-driven cache attacks on aes. Cryptology ePrint Archive rep. 2006/138.

ARM. 2010–2011. Cortex-r5 and cortex-r5f: A technical reference manual, revision r1p1. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0460c/DDI0460C_cortexr5_trm.pdf (accessed 7/11).

BACKES, M., DURMUTH, M., GERLING, S., PINKAL, M., AND SPORLEDER, C. 2010. Acoustic side-channel attacks on printers. In *Proceedings of the USENIX Security Symposium*.

BANGERTER, E., GULLASCH, D., AND KRENN, S. 2011. Cache games - bringing access-based cache attacks on aes to practice. In *Proceedings of IEEE Symposium on Security and Privacy*.

BERNSTEIN, D. 2005. Cache-timing attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

BIHAM, E. AND SHAMIR, A. 1991. Packaging of multi-core microprocessors: Tradeoffs and potential solutions. *J. Cryptology 4,* 1, 3–72.

BLOWFISH. 2009. The blowfish encryption algorithm. http://www.schneier.com/blowfish.html.

BONNEAU, J. AND MIRONOV, I. 2006. Cache-collision timing attacks against aes. In *Proceedings of the CHES Workshop*.

BRICKELL, E., GRAUNKE, G., NEVE, M., AND SEIFERT, J. 2006. Software mitigation to hedge aes against cache-based software side channel vulnerabilities. IACR ePrint Archive, rep. 2006/052.

CANTEAUT, A., LAURADOUX, C., AND SEZNEC, A. 2006. Understanding cache attacks. INRIA Tech. rep. ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-5881.pdf.

DAEMEN, J. AND RIJMEN, V. 2002. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer.

GOUBIN, L. AND PATARIN, J. 1999. DES and differential power analysis. In *Proceedings of the CHES*.

GUERON, S. 2008. Advanced encryption standard (AES) instruction set. White paper, Intel.

JALEEL, A., THEOBALD, K., STEELY, S., AND EMER, J. 2010. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

KELSEY, J., SHNEIER, B., WAGNER, D., AND HALL, C. 1998. Side channel cryptanalysis of product ciphers. In *Proceedings of the 5th European Symposium on Research in Computer Security*. 97–110.

KERAMIDAS, G., ANTONOPOULOS, A., SERPANOS, D., AND KAXIRAS, S. 2008. Non-deterministic caches: A simple and effective defense against side channel attacks. *Design Automation Embedd. Syst*.

KONG, J., ACLICMEZ, O., SEIFERT, J., AND ZHOU, H. 2009. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.

KOPF, B. AND BASIN, D. 2007. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*. 286–296.

LEE, R. AND CHEN, Y. 2010. A processor accelerator for aes. In *Proceedings of the Symposium on Application Specific Processors (SASP)*.

LUK, C., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V., AND HAZELWOOD, K. 2005. PIN: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

LUO, K. AND FRANKLIN, M. 2001. Balancing throughput and fairness in smt processors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*.

MATSUI, M. 1994. Linear cryptanalysis method for DES cipher. In *Proceedings of the Advances in Cryptology*, 386–397.

MAY, D., MULLER, H., AND SMART, N. 2001. Randomized register renaming to foil DPA. In *Proceedings of CHES*.

MIBENCH. 2009. The MiBench benchmark suite. http://www.eecs.umich.edu/mibench/.

MICALI, S. AND REYZIN, L. 2004. Physically observable cryptography. In *Proceedings of the Theory of Cryptography Conference*

NEHALEM. 2009. First the tick, now the tock: Intel microarchitecture (nehalem). http://www.intel.com/technology/architecture-silicon/next-gen/319724.pdf.

OSVIK, D., SHAMIR, A., AND TROMER, E. 2005. Cache attacks and countermeasures: the case of aes. Cryptology ePrint Archive, rep. 2005/271.

PAGE, D. 2005. Partitioned cache architecture as a side-channel defense mechanism. Cryptology ePrint Archive.

PERCIVAL, C. 2005. Cache missing for fun and profit. http://www.daemonology.net/papers/htt.pdf.

PINPOINTS. 2009. Pinpoints home page. http://www.cs.virginia.edu/wiki/pin/index.php/PinPoints.

QURESHI, M. AND PATT, Y. 2006. Utility-based partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the International Symposium on Microarchitecture (MICRO-39)*. 423–432.

RANDOM. 2009. Random.org. http://www.random.org/.

SIDE. 2009. Side channel attacks database. http://www.sidechannelattacks.com.

SPRADLING, C. D. 2007. Spec cpu2006 benchmark tools. *SIGARCH Comput. Archit. News 35,* 1, 130–134.

STANDAERT, F.-X., PEETERS, E., ARCHAMBEAU, C., AND QUISQUATER, J.-J. 2006. Towards security limits in side-channel attacks. In *Proceedings of the CHES Workshop*.

SUH, E., RUDOLPH, L., AND DEVADAS, S. 2001. Dynamic cache partitioning for simultaneous multithreading systems. In *Proceedings of the International Conference on Parallel and Distributed Computing and Systems (PDCS'01)*.

TROMER, E., SHAMIR, A., AND OSVIK, D. 2009. Efficient cache attacks on aes, and countermeasures. *J. Cryptology*.

TSUNOO, Y., SAITO, T., SUZAKI, T., SHIGERI, M., AND MIYAUCHI, H. 2003. Crypronalysis of des implemented on computers with cache. In *Proceedings of the Cryptographic Hardware and Embedded Systems (CHES) Workshop*. 62–76.

TSUNOO, Y., TSUJIHARA, E., MINEMATSU, K., AND MIYAUCHI, H. 2002. Crypronalysis of block ciphers implemented on computers with cache. In *Proceedings of the ICITA Conference*.

WANG, Z. AND LEE, R. 2007. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

WANG, Z. AND LEE, R. 2008. A novel cache architecture with enhanced performance and security. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

XIE, Y. AND LOH, G. 2009. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

ZHAO, X. AND WANG, T. 2010. Improved cache trace attack on AES and CLEFIA by considering cache miss and s-box misalignment. Cryptology ePrint Archive, rep. 2010/056.

ZHOU, S. 2010. An efficient simulation algorithm for cache of random replacement policy. Lecture Notes in Computer Science vol. 6289, 144–154.

Received July 2011; revised October 2011; accepted November 2011