Limiting Cache-based Side-Channel in Multi-tenant Cloud using Dynamic Page Coloring

Jicheng Shi, Xiang Song, Haibo Chen, Binyu Zang
Parallel Processing Institute, Fudan University
{jcshi, xiangsong, hbchen, byzang}@fudan.edu.cn

Abstract

Multi-tenant cloud, which features utility-like computing resources to tenants in a "pay-as-you-go" style, has been commercially popular for years. As one of the sole purposes of such a cloud is maximizing resource usages to increase its revenue, it usually uses virtualization to consolidate VMs from different and even mutually-malicious tenants atop a powerful physical machine. This, however, also enables a malicious tenant to steal security-critical information such as crypto keys from victims, due to the shared physical resources such as caches.

In this paper, we show that stealing crypto keys in a virtualized cloud may be a real threat by evaluating a cache-based side-channel attack against an encryption process. To mitigate such attacks while not notably degrading performance, we propose an approach that leverages dynamic cache coloring: when an application is doing security-sensitive operations, the VMM is notified to swap the associated data to a safe and isolated cache line. This approach may eliminate cache-based side-channel for security-critical operations, yet ensure efficient resource sharing during normal operations. We demonstrate the applicability by illustrating a preliminary implementation based on Xen and its performance overhead.

1. Introduction

Multi-tenant cloud, which usually leases computing resources to tenants in the form of virtual machines (VMs), have been adopted in various usage scenarios such as application hosting, content delivering, e-commerce and web hosting [2]. The approach of consolidating resources using virtualization allows the cloud infrastructure providers to achieve optimal resource utilization while maintaining adequate isolation.

However, providing *virtual isolation* (i.e., VM) other than *physical isolation* may also have some security implications. For example, co-locating VMs on the same platform may lead to implicit resource sharing (e.g., cache) among co-located VMs, which introduces opportunities of security interference. Previous researchers have demonstrated the applicability of using various side-channel attacks to extract information such as physical location and workload information [13].

Side-channel attack, which leverages low-bandwidth message channels (e.g., timing, power, cache misses) in a system to derive or leak security-sensitive information, has been proven to be realistic threats to modern computer systems. Among them, cache-based side-channel attacks have been shown practical to steal cryptographic information within a single operating system [4], [10], [12]. The main idea is that cryptographic algorithms usually have data-dependent memory access patterns, which can be revealed by observing and analyzing the associated cache hit/miss statistics. Cache-based attacks then can rely on certain statistics during the encryption or decryption operations to extract the cryptographic key.

In this paper, we make the first illustration of the applicability of mounting cache-based side-channel attacks among VMs in multi-tenant cloud, by building a simple example of cross-VM side-channel attacks through revealing the cache hit/miss statistics [10]. The attack is done on an Intel i7 machine with hyper-threading technology running the Xen VMM [3], where the victim guest VM shares the same L1 cache with the attacking VM. Our experiment shows that the attacking VM can still extract the cryptography key information even in the presence of much more interference than in a single OS.

One intuitive defense against cache-based side-channel attacks across VMs is to provide strong cache isolation among VMs such as applying static page coloring in virtual platforms [7]. However, this approach will proportionally decrease the available cache sets for use, thus may significantly degrade the performance for not only the protected VM, but also other VMs. Further, typical processor cores usually have limited number of cache sets, which could limit the number of runnable VMs within a shared cache when applying static page coloring.

To enforce cache isolation while providing good performance, we propose a non-intrusive, low-overhead dynamic page coloring mechanism, named Chameleon, which provides strict cache isolation only during security-critical operations. A specific color (named *secure color*) is assigned to the secure process so that strict cache isolation can be achieved through dynamic page coloring. We provide a specific interface for applications to notify the hypervisor the entering of a security-critical section. During the security-critical section, the *secure color* is only available for security-critical operations and not usable by any other co-located VMs on the same hardware platform.

We have implemented Chameleon based on Xen [3] with 750 lines of code changes. The prototype only requires several lines of code changes to applications and no change to guest OS core kernel to protect VMs from cache-based side-channel attacks. Our preliminary performance evaluation using a key-encryption process and Apache SSL mode shows that Chameleon incurs negligible overhead when isolating only security-critical operations, and still acceptable performance overhead when isolating the entire application.

The rest of this paper is organized as follows. Section 2 describes the threat model and illustrates the applicability of cache-based attacks using a simple experiment on a virtualized platform. The overall idea of Chameleon is described in section 3, followed by the design and implementation in section 4. The preliminary evaluation result is shown in section 5. We then briefly discuss related literatures (section 6) and conclude the paper in section 7.

2. A Case on Cross-VMs Side Channel Attack

This section first illustrates the cache-related threat in multi-tenant cloud and describes the threat model. Then, it shows that the threat may be realistic by describing an attack against an AES encryption application in a virtualized environment.

2.1. Shared Cache and Threat Model

The speed gap between processors and main memory makes cache a critical component for performance. To hide memory access latency and increase parallelism, many commercial processors such as POWER, UltraSPARC and Xeon usually support simultaneously multi-threading technology, which allows multiple hardware threads simultaneously executing on the same CPU core. Hence, except the typical sharing of the last-level cache among multiple cores on a chip, multiple hardware threads also shares the private (e.g., L1 and/or L2) cache of a core in a multicore processor. While efficient, the sharing of caches among cores and threads also introduces the vulnerability of cache-based side-channel.

Here, we assume that the cloud provider and the underlying infrastructure are trustworthy and will not explicitly leak data to an adversary. Hence, the only way an adversary can steal information is from implicitly shared resources. Given other side-channels [13] in a multi-tenant cloud, we assume that an adversary may leverage such information to co-locate an evil VM whose sole purpose is stealing information from other victim VMs.

2.2. AES First-Round Attack and Analysis

In AES, the cryptographic process needs many computation steps. For the sake of performance, most implementations of AES use look-up tables and one cryptographic operation is divided into ten or more rounds. The table lookup index in the first round is obtained from the exclusive OR (XOR) result of the plain text and the AES key. Hence, if we know the location of the table (i.e., index) that the AES operation accesses and the plain text, we can guess some bits of the AES key. If a victim VM is running some encryption services (e.g., SSL), an attacking VM can supply some plain text to the victim VM through networking and the monitoring the cache access statistics to guess the index to the table. It should be noted that the lookup table is publically available as the attacking VM can easily guess which AES implementation the victim VM uses.

We conducted such a first-found AES attack [10] that leverages the first-round table lookup to guess AES key on an Intel i7 machine (with 2 hardware threads) on the Xen VMM. The test coexists two hardware-assisted VMs on two different physical threads on the same core , thus sharing the L1 cache. The attack procedure can be divided into four steps:

1) First, the attacking VM gets the starting address of an AES encryption lookup table. This is usually not difficult on a VM without address-space randomization if the attacking VM has already known which software the victim VM is running with. The starting address can be used to determine the cache sets for the lookup table. As the cache line size is 64 bytes and each element in the lookup table is with 4 bytes, each cache line contains 16 elements. Typically, the entire lookup tables in AES occupy 4 KBytes memory.
2) Second, the attacking VM allocates a buffer whose start address also fits within the same cache set with the lookup table. Then, the attacking VM profiles the memory access time of the buffer in normal case when the victim is not using that cache set, by reading the buffer multiple times and using the average access time from a set of minimal access time for reference.
3) Third, the attacking VM sends a random 16-byte plain text to the victim VM for encryption. Once the victim VM completes the encryption operation, the attacking VM accesses its own buffer again to profile access time. As shown in figure 1, if the encryption procedure in the victim VM accesses the lookup table, the cached content will be flushed out and the attacking VM will take longer time to access its buffer because of the cache miss. To get a convincing result, we repeat step 2 and 3 hundreds of times and record the differences between these two steps.
4) Finally, we analyze the differences between these two observed cache-access time and the plain text. If there is a cache miss for a cache line, we can then infer the index. For instance, if the key byte is 0xa0 and the plain text byte is 0xa1, then the attacking VM will always touch the cache set indexed by $0xa0 \oplus 0xa1 = 0x0e$. This encryption procedure will access the table's 15th (0x0e) element, which just locates in first cache set for the lookup tables. As the encryption procedure accesses the first cache set and any key byte such as 0xaY (Y is any hexadecimal number) can cause table lookup in the first cache set, we can infer that the key byte must be in the form of 0xaY. Hence, the attacking VM successfully steals 4 bits from the victim VM.

Figure 2 shows the analysis result of our attacking to key byte 0xa0. We analyze the cache-missing rate in the cache set for the lookup table classified by corresponding plain text. The x-axis represents the plain text byte. *Ranking First* means the probability of highest cache missing rate, while *Ranking Second* means the probability of the second highest one. As shown in Figure2, when the plain text byte starts with **a**, the cache miss rate is the highest. In practical attack situation, we can consider to use bytes starting with **a** as the candidate key byte.

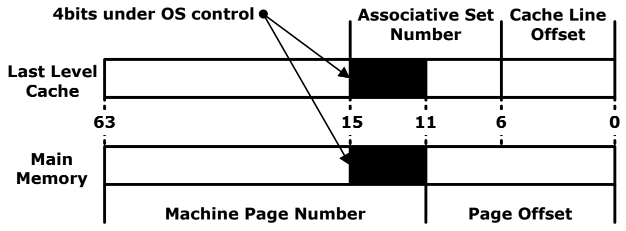

Fig. 3. Address mappings for cache and memory pages: the overlapped bits are controlled by the page coloring system.

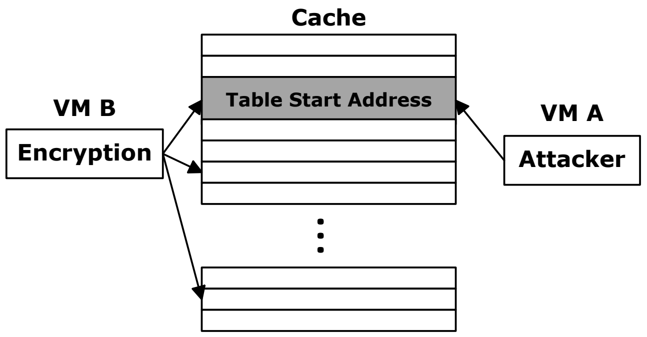

Fig. 1. Lookup table and how the attacking VM tries to access the lookup table and causes cache misses.

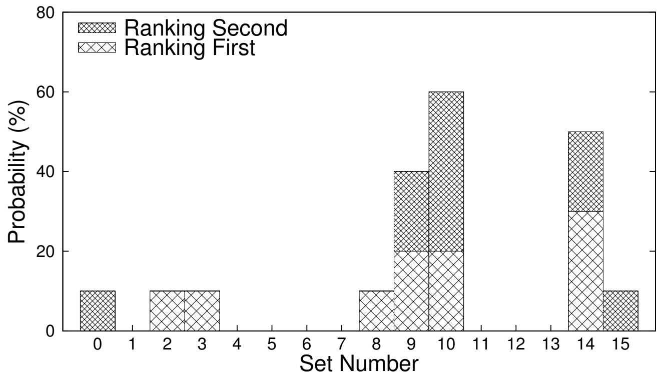

Fig. 2. Candidate key probability of a sample run.

This side-channel attack is affected by network transferring. Cache pollution during network communication may lead to inaccurate results. Thus, the candidate key may not be the one causing the highest cache miss rate. For example, key byte 0xey and 0x9y may be the right one in our previous test. This usually requires hundreds of tests to increase accuracy. Anyway, such an attack can shorten the key searching space and could be powerful when combined with other social-engineering techniques.

3. A Case for Dynamic Page Coloring

In this section, we first briefly review the technique of page coloring, and then describe the dynamic page coloring approach that balances the requirements for security and performance.

3.1. Background on Page Coloring

Page coloring is a software-based technique that directs how memory pages are mapped to cache lines. It is previously introduced by Taylor et al. [16] as an OS mechanism to stabilize performance in a virtually-tagged cache with a physical cache index for MIPS. Recently, it is usually employed in operating systems to improve the fairness and utilization of cache in multicore [8], [14], [15], [18]. There are two categories of page coloring techniques: static page coloring [8] and dynamic page coloring [8], [14], [18].

Essentially, page coloring systems controls the memory management module to ensure that a group of pages with the same color will be mapped to a fixed set of cache lines. Figure 3 illustrates how memory pages are mapped to the cache lines. There are several overlapped bits between the cache associative set number and the machine page number, which are directly under the control of the page coloring system. These bits can be used to group memory pages into different colors. In the example of Figure 3, there are 4 overlapped bits, which indicates that the cache can be partitioned into up to 16 colors.

3.2. Limiting Cache Side-Channel with Dynamic Page Coloring

An intuitive approach to defending against cache-based attacks is through static page coloring among VMs [17], which however can proportionally decrease the number of cache sets a VM can use. However, many applications' performance is very sensitive to the cache size and the sensitivity is exacerbated by the memory wall. Further, depending on the set associativity and the total size of a cache, there are usually a very limited number of colors that a page coloring system can provide, which may place a restriction on the number of virtual machines running on a cache.

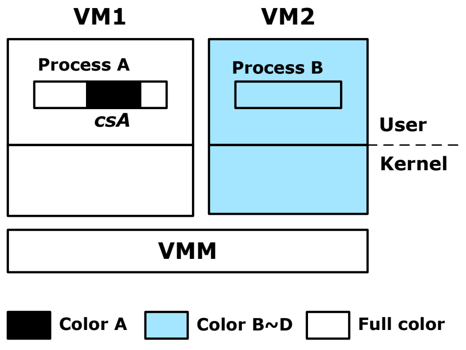

Fig. 4. Dynamic page coloring in Chameleon. Color A is a secure color and is dedicated to security-sensitive applications.

To address the issues in performance slowdown and restrictions on VMs, we propose a dynamic page coloring scheme that balances the requirements for security and performance. Specifically, our system, called Chameleon, dedicates a set of specific *secure colors* in each shared cache and allows other colors to be flexibly used by other VMs, as shown in Figure **??**. The secure colors can also be shared by other VMs when there are no security-sensitive operations running. When a VM indicates that a security-sensitive application is about to execute, Chameleon will do a page recoloring by swapping all pages using the secure colors with pages of other colors. The recoloring is done by manipulating the address translation from the guest physical address to host physical address, which is thus transparent to guest VMs. Chameleon also manages a pool of pages that corresponds to the secure colors. Program data (e.g., AES key tables) that demands prevention of side-channel attacks will be allocated from this secure pool.

To provide different levels of defense against side-channel attacks, Chameleon supports two protection modes:

- *Selective Mode*: In this mode, only the critical memory section specified by the application is isolated using *secure colors*. In this mode, we suppose that an application knows the location and size of the critical memory. Although this mode lacks strong isolation of entire application data, it can provide sufficient protection for security-sensitive data and can benefit from good performance.
- *Full Mode*: In this mode, any pages used by an application are isolated using *secure colors*.

Figure 4 also shows how Chameleon protects a security-sensitive application on a *secure VM*. VM1 is the secure VM and Process A is the security sensitive application. In selective mode, when Process A needs to protect its security-critical memory, all pages belonging to the memory (*csA*) must use the secure color (*ColorA*). Meanwhile, the potential evil VM, namely VM2, which shares the same cache with VM1, is guaranteed not using the pages with the secure color. Process A now can execute safely without leaking the footprint of its secure data accesses through the shared cache. In the full mode, all pages of the application process must be limited into *ColorA*, which demands a recoloring of pages currently using that color. This requires some page table walking and page copying.

In Chameleon, the performance is affected the frequency of recoloring and the available cache sets to use. Usually, if a VM is involved in frequent security-sensitive operations (e.g., an encryption server), it may be more beneficial to dedicate the secure colors to the application, to avoid frequent recoloring. For VMs that are with infrequent security operations, dynamical page coloring will be a good choice to balance available cache sets to use and security. Further, in many applications, it is common that only a small portion of data requires guards against information leakage. In such a case, the selective mode will provide good performance to an application. In full mode, as the entire application can only use the secure color, its performance will likely be degraded.

4. Design and Implementation

We have implemented Chameleon based on Xen running hardware-assisted virtual machines (HVM) with shadow page management. In the following, we describe the implementation issues of Chameleon.

4.1. Boot Preparation

During system booting, Chameleon reserves a small set of memory for page coloring. Chameleon partitions such memory into different colors according to the underlying cache infrastructure. These pages are organized into different pools according to their colors. For example, in our testing environment, Chameleon partitions the reserved memory into 32 different colors. Chameleon organizes the memory into 8 color groups, each of which contains four color pages. The colors in the first group are defined as security colors.

4.2. Interaction with Security-Sensitive Applications

Chameleon is a passive module inside the Xen VMM. When a security-sensitive application needs to execute security-critical operations, it can then use a *vmmcall* to notify Chameleon to turn on the protection mode. Specifically, Chameleon uses the VMMCALL and VMCALL instruction on SVM [1] and VMX [9] accordingly. The request contains additional arguments that specify the address range to protect and the protect mode. These arguments are transferred through registers when calling VMMCALL or VMCALL.

4.3. Preparing Protection

When Chameleon receives a protection request from the application, it takes the action according to the protection mode. If the *selective mode* is requested, Chameleon scans the shadow page table for the specified memory area and marks the pages as non-present with a *magic* bit set in its corresponding page type info. The following accesses to these pages will cause a page fault, and Chameleon will check the magic bit to see if the page has been nullified by itself. In *full mode*, Chameleon has to walk the application's shadow page table and clear pages with insecure color belonging to the application. Besides of dealing with the shadow page table of a protected application, Chameleon also scans the shadow page tables for other related potential evil domains in both modes. Chameleon clears all the pages with the secure color in these potential evil domains.

4.4. Intercept Memory Accesses

Accessing pages marked as non-present or allocating a new page will cause a shadow page fault, which is handled by the shadow page fault handler in VMM. If the page fault is caused by the nullification by Chameleon or by new page allocation, Chameleon will replace the old page with a page with the right color.

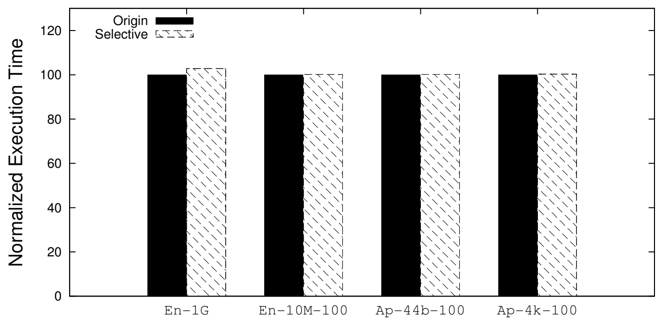

Fig. 6. Performance overhead for selective mode.

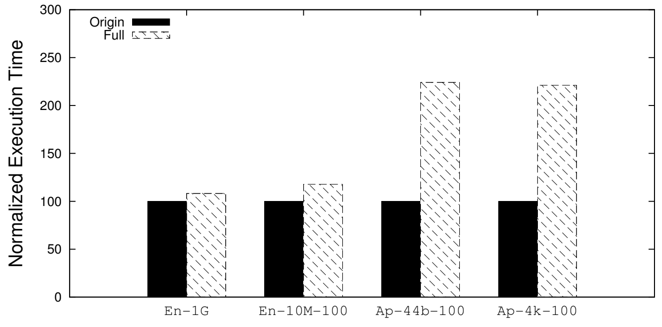

Fig. 7. Performance overhead for full mode.

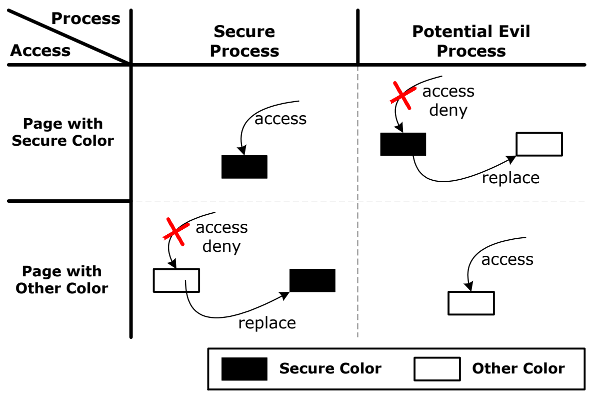

Fig. 5. Monitoring memory access and swapping pages in Chameleon.

4.5. Turn off Protection

When the application needs to turn off the protection, it will send another *vmmcall* to Chameleon. Chameleon scans the old page pool to return the old pages back to shadow page tables and changes the guest-physical to host physical mapping table. Chameleon reclaims the pages borrowed by domains back. Chameleon rewalks the shadow page tables and locates the page entries with the magic bit set. These entries will be marked as present again.

4.6. Optimization for Full Mode

The full mode needs to walk the entire shadow page tables when turning on and off the protection. Returning pages when turning off protection also introduces performance overhead. To reduce such overhead, Chameleon records the number of pages taken from reserved page pools. When turning off *full mode* protection, Chameleon will check the recorded pages. If the value exceeds some threshold, Chameleon will return the pages to reserved page pools. Otherwise, Chameleon will ignore the turn off command. This optimization works well in situations where an application touches a small number of pages. However, if application touches a lot of pages during the protecting period and the memory for the virtualized system is scare, this optimization will have limited benefit.

4.7. Implementation Status

We initially implemented Chameleon on Xen 3.3.0 first and ported it to Xen 4.0.0 later. Because the shadow page fault handler in Xen 4.0.0 is almost the same as that in Xen 3.3.0. We change very little code during the porting. In total, the implementation adds 750 lines of code to Xen and requires no change to the guest OS kernel.

5. Preliminary Results

This section evaluates the performance overhead of the Chameleon system. Our evaluations were conducted on a machine with a quad-core Intel processor (i7 930) with 2 hardware threads per-core and 2 GBytes memory. We created two VMs, one is the victim VM (VM1) and the other is the attacking VM (VM2).

We tested two modes of Chameleon, *selective mode* and *full mode*, with five benchmarks. Two encryption benchmarks are involved with encryption of 100 10M files (En-10M-100) and 1G files once (En-1G). The other three benchmarks are Apache HTTPD SSL mode testing. We tested the time of transferring three files with different sizes. Their sizes are 44bytes (Ap-44b-100), 4K (Ap-4k-100) and 1M (Ap-1M-100) accordingly. The y-axis represents consumed time normalized to the unchanged Xen hypervisor.

Figure 6 shows the performance overhead under the selective mode, where only the encryption module is protected. All benchmarks have overhead less than 3%, which is caused from the reduced cache size and the associated page replacements.

Figure 7 shows the test results under the full mode. Benchmarks *En-1G* and *En-10M-100* have less than 12% overhead. Our optimization has little effect in this situation, because one process only needs to turn on the *full mode* only once. For the benchmarks *Ap-44b-100*, *Ap-4k-100* and *Ap-1M-100*, as the application needs to turn on the protection several times for each network request. Because Apache touches many pages for each response, Chameleon has to walking page table and return pages to the *secure page pool* during these 100 requests. The optimization only reduces the times of page table walking and returning page. In total, the cost of whole protection is 2.24X and 2.21X accordingly. Anyway, users can switch to use selective mode if their applications are more sensitive to performance. Actually, for Apache with SSL mode, as the memory needs to protect can be easily determined, they can easily be protected using the selective mode.

6. Related Work

A lot of work has been done on cache-based side channel attacks on stealing cryptographic information [4], [10], [12]. Ristenpart et al. [13] demonstrate that it is possible to introduce side-channel attacks into commodity cloud computing environments in which VMs belonging to different users can co-locate on the platform with implicit resource sharing. In this paper, we proposed a dynamic page coloring mechanism to protect secure process dynamically. As the secure color is exclusively owned by the secure process against other co-located VMs, it is impossible for the VMs to steal information of the secure process by observing and analyzing cache behavior.

There are several defenses against cache-based side-channel attacks. These methods may also be applied to defend against cross-VM attacks. One approach is to rewrite the software in a way that known attacks cannot succeed [5]. There are also proposals that refine the processor architecture to minimize cache-based information leakage [17], which, however, requires non-standard hardware. Oswald et al. [11] presented a method to disable cache sharing. Recently, Intel has proposed new AES instructions that aiming at mitigating cache-based side-channel [6]. However, this approach is only available in newer processors and is limited to side-channels for AES operations. In contrast, Chameleon is a more general and portable approach to mitigating cache-based side-channels.

7. Conclusion and Future Work

In this paper, we have demonstrated the applicability of cache-based side-channel in a virtualized environments using a simple example. We also presented a countermeasure to certain attacks in cloud platforms. Our approach used dynamic page coloring to partition caches dynamically among security-critical applications of cloud tenants to exclude any possible cross-VMs cache interference during security-critical operations. A preliminary implementation demonstrated that the cost of protection is not significant. In our future work, we plan to investigate the tradeoff between the number of dedicated cache colors and the incurred overhead. Further, we plan to validate the approach in other virtualization techniques such as extended page table or nested page table, which might have even less performance overhead.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work was funded by China National Natural Science Foundation under grant numbered 90818015 and 61003002, a grant from the Science and Technology Commission of Shanghai Municipality numbered 10511500100, a research grant from Intel, Fundamental Research Funds for the Central Universities in China , Shanghai Leading Academic Discipline Project (Project Number: B114) and Wang-Dao Undergraduate Research Endowment of Fudan University.

References

[1] AMD virtualization technology. http://www.amd.com/virtualization.
[2] Amazon Inc. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/, 2011.
[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, page 177. ACM, 2003.
[4] D. Bernstein. Cache-timing attacks on AES. 2005.
[5] E. Brickell, G. Graunke, M. Neve, and J. Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR ePrint Archive, Report*, 52, 2006.
[6] S. Gueron and M. E. Kounavis. New processor instructions for accelerating encryption and authentication algorithms. *Intel Technology Journal*, 2009.
[7] X. Jin, H. Chen, X. Wang, Z. Wang, X. Wen, Y. Luo, and X. Li. A Simple Cache Partitioning Approach in a Virtualized Environment. In *Proc. IPDPS*, 2009.
[8] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *the 14th Symposium on High-Performance Computer Architecture (HPCA)*, pages 367–378. ACM, 2008.
[9] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–177, 2006.
[10] D. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *RSA Conference Cryptographers Track(CT-RSA)*. Springer, 2006.
[11] Oswald E, Mangard S, Pramstaller N, Rijmen V. A side-channel analysis resistant description of the aes s-box. In *Lecture notes in computer science: FSE*, 2005.
[12] C. Percival. Cache missing for fun and profit. *BSDCan 2005*, 2005.
[13] Ristenpart, Thomas and Tromer, Eran and Shacham, Hovav and Savage, Stefan. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proc. CCS*, 2009.
[14] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *the 41st IEEE/ACM International Symposium on Microarchitecture, 2008. MICRO-41.*, pages 258–269. IEEE/ACM, 2008.
[15] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared L2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems and Computer Architecture*. Citeseer, 2007.
[16] G. Taylor, P. Davies, and M. Farmwald. The TLB slice low-cost high-speed address translation mechanism. *ACM SIGARCH Computer Architecture News*, 18(3a):363, 1990.
[17] Z. Wang and R. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 83–93, 2008.
[18] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102. ACM, 2009.