Copyright ⓒ 2018 KSII

CacheSCDefender: VMM-based Comprehensive Framework against Cache-based Side-channel Attacks

Chao Yang*, Yunfei Guo, Hongchao Hu, Wenyan Liu
National Digital Switching System Engineering & Technological Research Center
Zhengzhou, 450000, China
[1989600235@qq.com]
*Corresponding author: Chao Yang

Received July 6, 2017; revised April 22, 2018; revised June 28, 2018; accepted July 15, 2018; published December 31, 2018

Abstract

Cache-based side-channel attacks have achieved more attention along with the development of cloud computing technologies. However, current host-based mitigation methods either provide bad compatibility with current cloud infrastructure, or turn out too application-specific. Besides, they are defending blindly without any knowledge of on-going attacks. In this work, we present CacheSCDefender, a framework that provides a (Virtual Machine Monitor) VMM-based comprehensive defense framework against all levels of cache attacks. In designing CacheSCDefender, we make three key contributions: (1) an attack-aware framework combining our novel dynamic remapping and traditional cache cleansing, which provides a comprehensive defense against all three cases of cache attacks that we identify in this paper; (2) a new defense method called dynamic remapping which is a developed version of random permutation and is able to deal with two cases of cache attacks; (3) formalization and quantification of security improvement and performance overhead of our defense, which can be applicable to other defense methods. We show that CacheSCDefender is practical for deployment in normal virtualized environment, while providing favorable security guarantee for virtual machines.

Keywords: Cloud computing, cache-based side-channel attacks, dynamic remapping, cache cleansing, comprehensive defense

1. Introduction

Popularity and maturity of cloud computing technology brings us with a new form of business, such as Amazon's Elastic Compute Cloud (EC2) [1], Microsoft's Azure Service Platform [2], and Rackspace's Mosso [3], which provides third-party resources for a wide variety of traditional businesses. In order to maximize utilization of those resources, infrastructure in cloud is always serving more than one clients, which leads to popular character called multitenancy. While multitenancy realizes improvement of resource utilization, it creates a range of new security concerns, of which cache-based side-channel attack [4, 5, 6, 7, 8] is one of the most dangerous and developed. This kind of attack occurs due to the fact that different virtual machines for clients share different levels of CPU caches, so malicious Virtual Machine (VM) achieving co-residency with victim VM gets chance to infer sensitive information from it through manipulating the shared cache.

Accordingly, there are a large number of solutions proposed for defending against cache-based side-channel attacks. Most of them can be divided into two types: one requires modifying the hardware [9][10], which imposes bad compatibility with current platform; the other needs to alter systems [11][12] or vulnerable applications [13, 14, 15], which is too specific and cannot be applied widely. There are also some defense methods proposed for VMM layer [16, 17, 18, 19, 20], which might influence the operation of normal applications [16], or would only defend against a portion of cache attacks [17], or greatly reduce benefits of cache sharing [18][19], or would bring too much overhead [20]. Besides, all current works defend blindly, thus costing more resources than needed.

In this paper, we propose a VMM-based comprehensive defense method in order to tackle the above shortages of current works. Our method is designed to achieve the following goals:

- **Comprehensive:** Our method provides a comprehensive defense against attacks on all levels of caches;
- **Compatible and general:** We enforce our defense in VMM layer, but not in any hardware, OS or application;
- **Attack-aware:** Our defense is aware of on-going cache attacks, and scheduled according to current circumstance.

In order to achieve the above goals, we first analyze and divide cache attacks into three cases based on level of targeted cache. Upon analysis of each case, we propose dynamic remapping, a novel version of traditional random permutation, along with cache cleansing in VMM layer to realize a comprehensive defense for all cache attacks. In addition, our defense is made attack-aware with detection of on-going cache attacks. The main contributions of this paper are as follows:

- We propose an attack-aware VMM-based comprehensive defense framework with combination of dynamic remapping and cache cleansing for all three cases of cache attacks which we divide in this paper;
- We design a new random permutation method called dynamic remapping which continues changing mapping relationship from virtual memory to cache to defend against two cases of cache attacks;
- We formalize and quantify each of our defense operations, which is further used to facilitate more precise and efficient scheduling of our defense.

- We compare our method with one of the current defense methods, and the result indicates that our method is far more efficient.

The rest of this paper is organized as follows: In section 2, we provide an overview of related works. In section 3, we briefly introduce the adversary model our method is defending against. In section 4, we describe our VMM-based comprehensive defense framework, followed by detailed introduction of dynamic remapping in section 5. In section 6, we give some formalized security analysis for each of our defense operations. In section 7, we further give some quantitative evaluation of our defense. Finally, section 8 concludes this paper.

2. Related Work

2.1 Cache

Due to big gap of access speed between main memory and fast processors, caches, which are smaller but faster memories, are designed to reduce the effective memory access time as seen by a processor. Modern processors feature a hierarchy of caches. “Higher-level” caches, which are closer to the processor cores, are smaller but faster than “lower-level” caches, which are closer to main memory. Level-1 (L1) caches, which are typically private high-level caches to the processor cores, are usually divided into two types, one of which is data cache while the other is instruction cache. Size of a typical L1 cache is 32 KB with a 4-cycle access time, as in Intel Core and Xeon families. The last level cache (LLC) is shared among all cores of a CPU and is a unified cache storing both data and instructions. Size of LLC measures at level of megabytes, and access latency is typically of the order of 40 cycles. Typical modern x86 processors also support core-private, unified level-2 (L2) caches of intermediate size and latency. Any memory access first accesses the L1 cache, and on a miss, the request is sent down the hierarchy until it hits in a cache or accesses main memory.

To exploit spatial locality, caches are organized in fixed-size lines, which are the units of allocation and transfer down the cache hierarchy. Size B of a typical line is 64 bytes, and the lowest-order $\log_2 B$ bits of the address, which is called line offset, are used to locate data inside the cache line. Modern caches are usually set-associative, which means that they are organized as S sets with W lines in each set. Such caches are called W-way set-associative caches. With such cache architecture, memory addresses are used to index these caches. When the cache is accessed, the set index field of the address, i.e. bits within $[\log_2 B, \log_2 B + \log_2 S - 1]$, is used to locate a cache set. The remaining high-order bits are used as a tag for each cache line. After locating specific cache set, the tag field of the address is compared with the tag of the W lines in that set to decide if it is matched.

2.2 Cache-based Side-channel Attacks

Cache-based side-channel attacks are originated in 1992 by Hu [21] when cache is considered to construct covert channel, and it is not until in 2004 that first practical cache attack [5] is carried out. Currently there are two basic forms of cache-based side-channel attacks. One of them is called Evict+Time [4], where the attacker first preempts CPU from the victim and primes target caches with his own data. Then he gives up the CPU, and measures the time used for sensitive operation of the victim. Finally, the attacker can get knowledge of whether specific cache sets are used by the sensitive operation through comparing duration time of sensitive operation before cache priming and after cache priming.

While Evict+Prime achieves poor results as the attacker should know exactly the start and the end of sensitive operation, another attack, called Prime+Probe [4], relaxes this restriction. To carry out Prime+Probe attack, the attacker should first prime target cache sets, then give up the CPU, which is followed by probing the same cache sets to determine whether certain data has been evicted from the cache by recording and comparing access time. Since only access time of the attacker himself needs to be measured duration priming and probing stages, it receives much less influence from the environment, and can therefore be applied in various situations [6, 7, 8, 30, 31]. In 2014, Prime+Probe was developed into a more powerful attack as Flush+Reload [7], which requires memory deduplication to infer the cache hit and cache miss. Basically, the natures of the latter two attacks are the same.

From then on, cache attacks have been further improved and extended. To bypass defenses capitalizing on the reliance of LLC attacks on timers, PRIME+ABORT [32], which utilizes the Intel TSX hardware widely available in processors, is immune to most defenses with better accuracy and efficiency. Besides, Cesar et al. proposes to take advantage of variable-time callers in the program to circumvent defenses based on constant-time callees [33]. Furthermore, cache attacks have been implemented on ARM-based mobile devices, such as AutoLock [34]. All these cases indicates the trend of wide application of cache attacks, which implies that corresponding defenses brooks no delay.

2.3 Defense against Cache-based Side Channels

Current defenses against cache-based side-channel attacks can be classified as follows:

- **Host-based defense**

Cache isolation might be the most intuitive method because the essence of cache attacks is the sharing of caches. Multiple methods are proposed, including locking cache lines for different VMs [12], partitioning caches [10] and separating memory pages according to the mapping relationship with caches [19]. The disadvantage is that isolation violates the principle of cloud computing and reduce its benefits.

Varadarajan et al. [17] proposes to control the least time duration that a VM must occupy a CPU core, thus cache behaviors will be hindered. Similarly, KeyDrown [35] defend against keystroke timing attacks by injecting a lot of fake keystrokes into the kernel, creating an uniform distribution of keystroke interrupt. However, it achieves limited defense effect due to the popularity of Simultaneous Multithreading (SMT, also called Hyper-threading) of CPU core and multi-core architecture. Besides, long waiting time for a VM's real-time jobs would affect interactive operations.

Zhang et al. proposed in [11] to add noise into cache behavior in order to mix different memory access operations. However, just adding noise is not enough since large amount of samples are possible to deal with those noises.

In [16], Vattikonda et al. introduced a method to use coarse-grained timers instead of fine-grained ones, thus reduce difference between access time of cache and memory. A similar approach is Cloak [36] which utilizes hardware transactional memory to make cache misses on sensitive code and data invisible to attackers. These methods might be impractical, because it would possibly affect the normal operation of applications, or even systems.

In [5], Aviram et al. proposed to modify the applications to make execution time as a constant. Besides, [15] proposed to dynamically change execution time through dynamic software diversity. These methods are limited because a lot of commercial software is not open-source. In addition, constant execution time cannot be really achieved due to noise from complex operations of application's execution environment.

Among host-based defense random permutation and cache cleansing might be most practical. There are two types of random permutation. As is proposed in [22][23], one way to realize random permutation is to randomly change virtual memory where sensitive data resides. Such a method is good to protect specific application, but would not be widely applied since each target application needs to be modified. Another type of random permutation is randomizing the mapping from machine address to cache sets [9,10,22], thus making information inferred from cache operations cannot reveal the memory access pattern of the victim. However, all current works of this type requires new design of hardware, leading to bad compatibility with current cloud platforms. To sum up, current works of random permutation are facing the following challenges: (1) They are not aiming at the root cause of cache attacks, which is cache sharing implemented in VMM layer; (2) They are either incompatible with current cloud platforms or too specific to certain application.

There are also typically two methods of cache cleansing [11][17]: using cache cleansing instructions and issuing memory visit. The latter method is implemented by randomly selecting memory addresses to issue memory access, changing the content of their corresponding caches. Compared with instruction-based method, it has an advantage that it can be used in any situation while executing cache cleansing instructions requires high privilege that might be forbidden.

- **Migration-based defense**

It is a good choice since it eliminates the underlying reason of cache attacks. However, it is still under development because of some migration problems [24], such as migration of network configuration and inevitable down time for service. Moreover, it defends blindly as all current works, thus wasting many unnecessary resources.

In light of the above analysis, we propose an attack-aware comprehensive defense composed of developed random permutation and cache cleansing. A new kind of random permutation which is called dynamic remapping is designed in a way that periodically changes mapping relationship from virtual memory to machine memory in VMM layer. Therefore, it can deal well with the above two problems of current random permutation. Besides, we utilize memory access as the complementary method to dynamic remapping in this paper, and adjust it to apply for our defense. At last, our defense operations is dynamically scheduled based on detection of on-going cache attacks.

2.4 Motivating Researches

Further improvements of our work are motivated by some other researches that focus on attack detection and mitigation against traditional cyber threats. For example, Weizhi Meng et al. propose the applicability of blockchain to intrusion detection [37] since blockchain can protect the integrity of data storage and ensure process transparency. We can use the blockchain to enhance the detection capability of HexPADS, which is the main part of Attack Detection Module of our proposed framework. To avoid the disadvantages of publickey certificates used in the blockchain, Qun Lin et al. construct a new ID-based linear homomorphic signature scheme [38], which is proved to be secure against existential forgery on adaptively chosen message and ID attack under the random oracle model. In addition, the attack detection can be further improved with deep learning [39], which has been proved to be effective in other fields such as image processing. At last, since the target of our approach is to protect the secret information, our work can be incorporated with modern cryptographic solutions [40] which are utilized in the field of cryptology and cyber threat prevention.

3. Attack Model and Formalization

In this section, we describe a general model of cache-based side-channel attacks in public clouds that (a) can capture most popular and powerful cache attack prototype as Prime+Probe and is easy to apply for other attacks; (b) is independent of specific cache that is under attack.

3.1 Adversary Goals and Capabilities

We assume that each VM in cloud has some private information (location of critical code or encryption table, etc.) in its memory and relative position of that information inside a memory page is fixed and known. The goal of the attacker is to first get the location of visited cache during sensitive operation which reflects memory pages it uses since different sets of pages cover different ranges of the cache. After that, with offset of sensitive data inside one (or many) of those pages, the attacker can further infer the position of all private information in cache, which is finally used to deduce critical information (such as encryption key) in target VM. Then we will give some capabilities of the attacker.

- **Co-residency and identification:** The malicious VM is able to achieve co-residency with target VM on the same physical platform and verify its identity [6];
- **Attacking technology:** We assume Prime+Probe attack described in [8], which is very powerful and has the widest scope of application than any other types of attacks. Flush+Reload may be more powerful but it is limited in the condition of memory sharing between the attacker and the victim. It should be pointed out that our method can defeat many other types of attacks like Evict+Time and its variations, which we will defend against in similar ways;
- **Memory Manipulation:** The attacker is able to manipulate memory of his/her own VM at will so that he can issue memory operations to access specific cache sets. This is a very aggressive assumption which is based on conditions like use of large pages since we are now dealing with the most powerful attacker;
- **Knowledge of data location:** Previous detection launched by the attacker is able to derive the location of sensitive data in cache, thus he/she can get a set of memory pages that map to this cache location which includes the one or those that contain sensitive information. If data location is changed (by our defense), the attacker only needs to search other possible positions that related page would map to instead of the whole cache.

After finding BMU, the SOM codebook vectors are updated, such that the BMU is moved closer to the input vector. The topological neighbors of BMU are also treated this way. This procedure moves BMU and its topological neighbors towards the sample vectors. The update rule for the *i*th codebook vector is:

3.2 Level-dependent Cache Data Location Model

In early work, researches on cache-based side-channel attacks assume that location of sensitive data is already known to the attacker which is definitely unreasonable except in the case of Flush+Reload attack. The only practical way proposed in recent work [8] is to use cache inference operations to deduce the location. In this paper, we assume that the attacker uses Prime+Probe operations to locate the position of target data in cache, and each single operation can check whether target data resides in a range of cache positions that mapped by a certain set of memory pages.

For locating sensitive information in cache, we identify level of targeted cache as the key factor. Typically, modern processors feature a hierarchy of three-level caches. They are

level-1 (L1) cache, level-2 (L2) cache and last level cache (LLC). Different levels of caches have different indexing mechanisms, which leaves the attacker with different sizes of search spaces to locate cache position of sensitive information. Obviously, there are three cases with respect to different levels of caches

- **Search space of L1 cache**

L1caches are usually small enough to be "virtually" indexed, which means that only bits in page offset field are enough to index all cache sets. So all pages will map into the same cache position, and just single data location operation is enough to ensure the existence of sensitive data in cache.

- **Search space of L2 cache**

Indexing L2 cache is very much different since sets of L2 cache is typically more than bits of page offset can index. So a page can map to the space indexed by the bits belonging to cache indexing bits but outside the range of page offset bits, which is also the search space for the attacker

- **Search space of LLC**

Cache set in the LLC is uniquely identified by the slice id and set index. Due to the sliced character of LLC, size of the search space might be different from that of unsliced LLC because it would depend on the number of slices in addition to the bits between the bits belonging to cache indexing bits but outside the range of page offset bits.

3.3 Formalization of the Model

We start by presenting some preliminaries as shown in **Table 1** below.

Table 1. Notations for formalizing data location model

Notation	Description
N_{L1}, N_{L2}, N_{L3}	Number of cache sets in L1 and L2 cache per core, as well as L3 cache of all cores. There are two types of L1 cache as data cache and instruction cache, and is the sum of them.
N_{PAGE}	Number of bits used to index page offset.
N_{LINE}	Number of bits used to index offset in a cache line
N_{SLICE}	Number of slices in LLC which equals the number of CPU cores.
W_{L1}, W_{L2}, W_{L3}	Associativity of three levels of caches.
T_{L1}, T_{L2}, T_{L3}	Time duration of single data location operation for three levels of caches.
S_{L1}, S_{L2}, S_{L3}	Size of search space for three levels of caches.
A_{L1}, A_{L2}, A_{L3}	Average time used to conduct one by one search for the whole space of three levels of caches.

With the above analysis, we can get expressions of the last six notations as in the following:

- **L1 cache-based data location**

Since L1 cache is virtually indexed by page offset, there is only one fixed position in cache for sensitive data. So its search space is limited to 1, and average time used for searching the whole space is to scan once. Formally,

$$S_{L1} = 1 \tag{1}$$

$$A_{L1} = T_{L1} \tag{2}$$

- **L2 cache-based data location**

In case of L2 cache, bits belonging to cache indexing bits but outside the range of page offset bits determine the search space, so the average time used for scanning the search space would be half the time used to conduct one by one search over the whole space. Formally,

$$S_{L2} = 2^{\log N_{L2} - (N_{PAGE} - N_{LINE})} \tag{3}$$

$$A_{L2} = T_{L2} \times 2^{\log N_{L2} - (N_{PAGE} - N_{LINE}) - 1} \tag{4}$$

- **LLC-based data location**

Calculation of search space and average time for scanning that space for LLC is similar to that for L2 cache except that the sliced character should be considered. Formally,

$$S_{L3} = N_{SLICE} \times 2^{\log \frac{N_{L3}}{N_{SLICE}} - (N_{PAGE} - N_{LINE})} = 2^{\log N_{L3} - (N_{PAGE} - N_{LINE})} \tag{5}$$

$$A_{L3} = T_{L3} \times 2^{\log N_{L3} - (N_{PAGE} - N_{LINE}) - 1} \tag{6}$$

However, we can find out that size of search space is not influenced by the introduction of sliced LLC, indicating that this design is completely not for security concern.

These above equations indicates a fact that changing the mapping relationship from memory address to L2 cache or LLC will introduce a large search space for the attacker to locate the sensitive data, which brings large entropy of security for cache users. This motivates our design of dynamic remapping. And for L1 cache, other mitigation methods might be used.

4. Frame Overview

In this section, we present CacheSCDefender, which combines dynamic remapping and cache cleansing to meet the following requirements:

- It is able to defend against the most powerful cache attacks for all levels of caches;
- It would not modify any hardware to provide good compatibility with current cloud platforms, and would not require much modifications for upper VM systems or applications in order to provide good compatibility with current cloud service;
- It is always aware of the on-going cache attacks and can defend accordingly;
- Its performance overhead is controllable so that it is practical for deployment in the cloud.

4.1 Overall Design

In this part, we provide a high-level overview of the CacheSCDefender framework which provides an attack-aware comprehensive solution for cache-based data location attacks we discussed in the previous section. **Fig. 1** shows the overall architecture of CacheSCDefender.

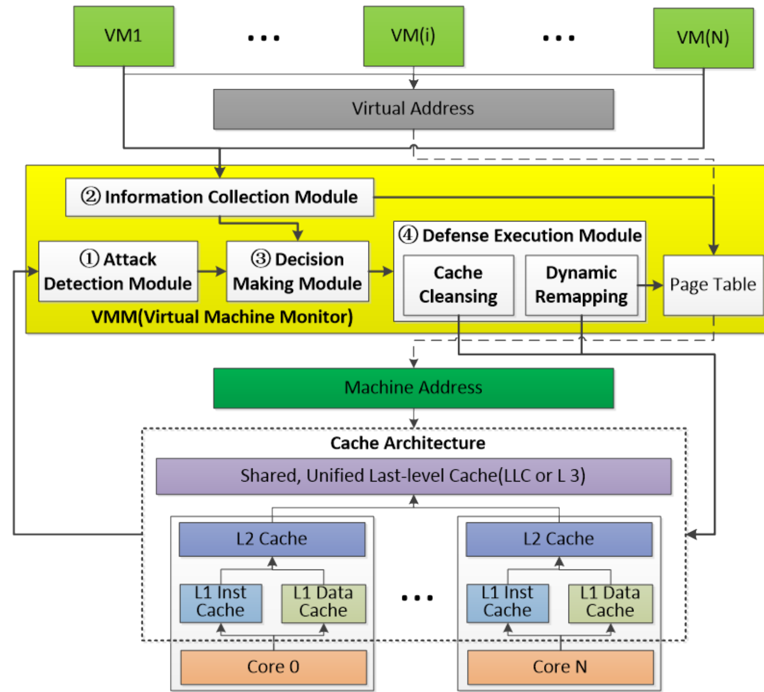


Fig. 1. Architecture overview of CacheSCDefender

Recall that we consider an adversary model that focuses on preparation stage of cache attacks which tries to locate sensitive data in cache. Our goal is to provide a mitigation mechanism against this threat model while meeting requirements stated at the beginning of this section.

Motivated by random permutation, we come up with an novel version called dynamic remapping designed in VMM layer which periodically changing mapping relationship from memory address to cache. After remapping, sensitive data resides at another random position of cache so that the attacker has to scan the whole search space to regain the location. One extreme solution might be using dynamic remapping for cache attacks against all levels of caches.

However, we know from Eq. (1) that search space is so much limited that dynamic remapping is definitely inappropriate. Thus we incorporate cache cleansing, a suboptimal method to be the supplement. It will clean cache content corresponding to our critical memory pages so as to confuse the attacker of data location. Besides, periodic cache cleansing can also slow down the attack process for stealing private information [11].

Despite all this, we will not substitute cache cleansing for dynamic remapping for the following two reasons: (a) Too frequent cleansing of cache would eliminate of performance benefits brought by sharing of physical cache; (b) Dynamic remapping is more efficient than cache cleansing, a fact that we will prove later in Section 6.

Therefore, the optimal solution should be to use dynamic remapping for cache attacks against L2 cache and LLC, and use cache cleansing for cache attacks against L1 cache.

4.2 System Modules

As is shown in **Fig. 1**, there are four modules designed in hypervisor layer to carry out our comprehensive defense. They are Attack Detection Module, Information Collection Module, Decision Making Module and Defense Execution Module.

- **Attack Detection Module**

This module is responsible for checking out whether cache-based side-channel attacks are happening, and which level of cache is under attack. Since our focus is not on attack detection, we will use the real-time detection system HexPADS mentioned in [25]. Their work is based on the observation that attacks will always change the execution behavior of a system, so HexPADS detects attacks through divergences from normal behavior using attack signatures. The system collects information from the operating system on runtime performance metrics with measurements from hardware performance counters for individual processes. Since cache behavior is a strong indicator of ongoing side-channel attacks, collecting performance metrics across all running processes allows the correlation and detection of these attacks almost without false positives or false negatives. The perceived overhead for HexPADS is negligible and makes up for less than 1% of CPU time on a single core on a modern system.

The output of this module is totally delivered to Decision Making Module.

- **Information Collection Module**

This module is responsible for collecting VM's information. The information includes which VMs are requiring our protection, and for those who need protection, we further collect information about which pages among them are applied as sensitive. Here we design a client API for VMs to apply for security-critical memory they want to protect, and our module is responsible for determining sensitive pages which stores these memory contents. In order to get security-critical memory for the given sensitive information of users, we design a small tool KeyDeLocater to analyze all security-critical codes in user program according to [26] which proposes to use data flow analysis to detect all necessary addresses. When the user gets all security-critical memory with KeyDeLocater, he should delivers the result to our client API, and this module would mark all pages containing these memory as sensitive pages. Of course, we limited the number of pages for each VM in case of DoS (Denial of Service) attack by malicious users who deliver specially constructed security-critical addresses in try to let the system mark as many sensitive pages as possible, thus causing unnecessary defense operations.

The output of this module is partially delivered to Decision Making Module, and also partially delivered to initiate or change content of the page table. We will use additional bits in page table entry to indicate whether a page is sensitive.

- **Decision Making Module**

In this module, decisions are made about how to carry out the defense method with information gathered from the first two modules. Its output will be delivered to Defense Execution Module to instruct practical defense operation.

Our decision making algorithm is an unconditional loop after initializing defense operations for all VMs requiring protection. And our initial protection is dynamic remapping as it is more effective.

After that, we enter a loop which is periodically checking the existence of cache-based side-channel attacks for each VM, which is realized by Attack Detection Module. When an attack is detected, we will reduce the interval for our defense operations. Moreover, we will decide which level of cache is under attack for VM_i, which is conducted by Information Collection Module. For case with L2 cache and LLC attack, we will use dynamic remapping even with presence of L1 cache attack because dynamic remapping covers action of cleaning cache; for case of only L1 cache attack, we carry out cache cleansing. If multiple consecutive detection operations fail to find the existence of cache attacks, we will increase the interval for defense operations in order to save resource used for protection. The threshold of this number is related to the number of VMs. For the change of interval duration, it is limited to the range of

$[MIN_{Interval}, MAX_{Interval}]$, and it increases and decreased by $\Delta_{Interval}$. In case that the attacker might be able to find out $\Delta_{Interval}$ and $Interval[i]$ which may be helpful to more successful attack, we introduce the randomness into change of $Interval[i]$, and the result could be any value between $Interval[i]$ and $Interval[i] \pm \Delta_{Interval}$.

- **Defense Execution Module**

The responsibility of this module is carrying out practical operations of our defense. It is composed of two parts: Dynamic Remapping and Cache Cleansing. Our Cache Cleansing is simple, as it just periodically flushes the caches by memory access which is flexible and does not require any privileged operations. As described in the decision making algorithm, we know that in cases of L1 cache attacks where caches are virtually indexed, cache cleansing is used in this case. It is simple since we just need to randomly visit those memory which maps to the same cache sets as SPs, as is shown in Fig. 2 below. All cache sets corresponding to those SPs should be covered.

In Fig. 2, the most important module is creating eviction sets which is used to clean cache sets related to sensitive pages. The algorithm is simple: for S ($S \geq W$, and W is associativity of corresponding cache) memory pages that map to the same cache sets as the sensitive page, first randomly select W pages to use their memory blocks corresponding to 1st cache set related to the sensitive page, thus creating eviction set 1. Then we randomly select W pages to create other eviction sets, until all sets have been created. After that, if we visit all these cache sets, cache content related to the sensitive page would be cleaned.

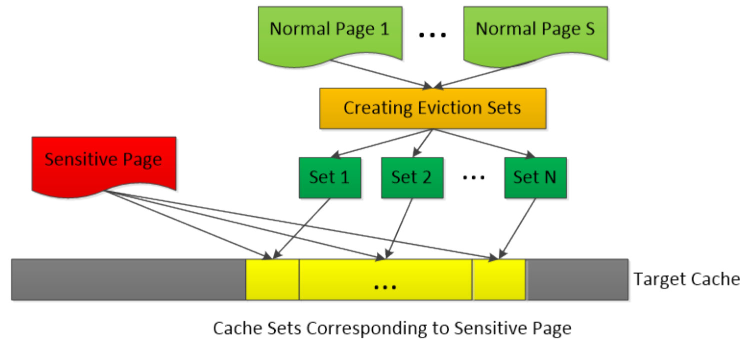


Fig. 2. Cache cleansing as defense for L1 cache attack

Instead, dynamic remapping turns out more complicated, as it should not only change the items of page tables, but also reconcile all memory related, including exchanging content of two memory pages, invalidating caches and TLBs. Nevertheless, dynamic remapping is a more effective timely operation, which we will detail in section 6.

5. VMM-based Dynamic Remapping

In this section, we will give detailed introduction of dynamic remapping, which is the most important component of our defense framework.

5.1 Basic Conception

Currently, there are mainly two mechanisms for mapping virtual address in VM to physical address in real machine. One is shadow paging based on software simulation, and the other is two dimensional page walk with hardware support. No matter which mechanism is used,

VMM could only affect the translation to machine address via page tables. We will utilize this feature to randomize the mapping relationship at the granularity of memory page. Our basic idea is shown in **Fig. 3**.

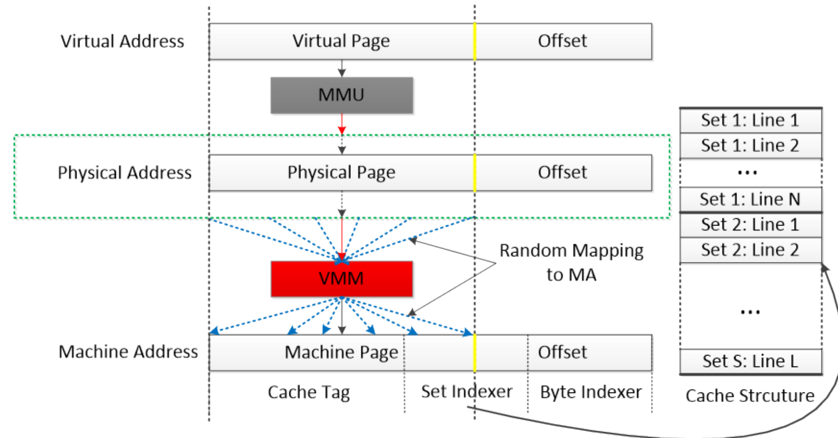


Fig. 3. Random permutation from machine memory to cache

In this paper, all memory pages (machine frames) of a VM are divided into two types: Sensitive Page (SP) which contains sensitive data and Ordinary Page (OP) which only contains ordinary data. We will enforce moving target defense on all SPs. For VMM layer, we can realize dynamic remapping with the following procedure:

a) Randomly choose a mapping record for a SP and a mapping record for an OP in the page table. The page table can be shadow table in shadow paging mechanism, or p2m (physical to machine) table in two-dimensional page walk mechanism. And as we are remapping at the granularity of page size, the records should be in L1 page tables which store the specific machine frame numbers;
b) Block visit from VM to these two records and flush TLB;
c) Exchange two selected items in the page table;
d) Block visit from VM to machine pages mapped from these two records in the page table;
e) Exchange the content of these two machine pages;
f) Visit these two machine pages so as to fill cache with their relative memory content;
g) Recover the visit from VM to the blocked page table entries and machine pages.

However, if large page is enabled in the system, and the sensitive information of the victim is located in the large page (like 2MB and 1GB), then dynamic remapping would not change the cache set range of the sensitive victim buffer, thus rendering the victim defenseless. Fortunately, for many applications, the sensitive data is usually in the small size (4KB) pages, and therefore dynamic remapping of those pages should be able to effectively prevent the identification of security-critical memory area.

5.2 Formalization

A defense method can be evaluated from two aspects: the overhead for carrying out the defense and the security improvement it brings. In our case, the overhead is mainly about time needed for defense operations, including time for dynamic remapping and cache cleansing. In regards to defend against information leakage, the security improvement can be described as additional cost for the attacker to get the same information due to the defense which can be widely applied. Then we have the following definitions:

Definition 1. *Effect of defense*, denoted as R_{EE} : minimal additional resource needed to get the same information as before due to the defense. In this paper, it refers to additional time used for locating sensitive data in cache.

Definition 2. *Effort of defense*, denoted as R_{EO} : minimal resource needed for carrying out certain defense operation or method. In this paper, it refers to total time used for the defense since other resources like memory are negligible.

Definition 3. *Efficiency of defense*, denoted as R_{EI} : ratio between the effect of defense and the effort of defense. This definition might be the most comprehensive indicator for evaluating the defense.

Table 2. Notations for formalized evaluation

Notation	Description
N_{SP} , N_{OP} , N_{VP}	Numbers of SPs, OPs and VPs of a VM.
T_{CC}	Time duration of cleaning corresponding cache with associativity of W_{L1} to single page.
T_{CR}	Time duration of changing single item in page table, as well as its corresponding records (TLB) in other places except in cache.
T_{CP}	Time duration of copying content of one page to address of another page.
T_R	Time duration of conducting memory read operation for size of a single cache line.
T_C	Time duration of conducting memory copy operation for size of a single cache line.
T_{RC}	Time duration of conducting cache read operation for size of a single cache line.
T_{SO}	Time duration of sensitive operation of the victim between priming stage and probing stage of a single Prime+Probe operation.

Besides, we define other factors involved in our defense in the following table. Here we take case of defense against LLC attack as an example, and defense against L2 cache attack can be viewed in similar way.

In this paper, effect of defense refers to additional time used for locating sensitive data in cache, which is product of the duration of single Prime+Probe operation and times of that operations needed. Here we use average time used to conduct one by one search for the whole space which is defined in section 2.3 to represent effect of a certain defense. For effort of defense, cache cleansing and dynamic remapping bring different expenses. As implied by its name, cache cleansing costs the effort of cleaning cache, which means visiting eviction sets of target pages. Compared with it, dynamic remapping is much more complicated since its cost is composed of the following operations

- Change items in page table, and flush TLB;
- Copy content of one page to address of another page;
- Clean corresponding cache to memory page by issuing memory visit to eviction sets.

Since there are two records to change, T_{CR} and T_{CC} will be doubled. For exchanging content of two pages, we use three-step exchange as ‘a→c; b→a; c→b’, so the exchange equals three times of copying content of a page to another. Therefore, T_{CP} is multiplied by 3.

$$R_{EO}^{L3} = 2T_{CR} + 3T_{CP} + \frac{2W_{L3} \times T_{CC}}{W_{L1}} \tag{7}$$

After dynamic remapping, the attacker has to scan the whole search space to regain the location of sensitive data in cache. For each SP, effect of single operation 2SP can be expressed as $T_{L3} \times 2^{\log N_{L3} - (N_{PAGE} - N_{LINE}) - 1}$ which is shown in Eq. (6). So the defense effect is:

$$R_{EE}^{L3} = 2^{\log N_{L3} - (N_{PAGE} - N_{LINE}) - 1} \times T_{L3} \tag{8}$$

With Eqs. (7) and (8), we can get the efficiency of 2SP operation:

$$R_{EI}^{L3} = \frac{2^{\log N_{L3} - (N_{PAGE} - N_{LINE}) - 1} \times T_{L3}}{2T_{CR} + 3T_{CP} + (2W_{L3} \times T_{CC}) / W_{L1}} \tag{9}$$

6. Security Analysis

In this section, we formally describe security improvement CacheSCDefender brings in each case of cache attacks, and then prove our announcement in section 4.1 from the aspect of security defense efficiency. At last, we describe how CacheSCDefender deals with smart adversaries in several cases.

6.1 Analysis of Security Improvement

Here we will give formalized evaluation of our defense in each case. In order to calculate the concrete value of effort and effect of our defense, we have to get values of following 6 critical variables: T_{L1}, T_{L2}, T_{L3}, T_{CC}, T_{CR} and T_{CP}.

As we assume perfect probing ability of the attacker, the attacker needs only to conduct a single Prime+Probe operation for scanning a single set, while each operation involves one round of W memory access and one round of W cache access, as well as one sensitive operation of the victim. Here we ignore waiting time of the attacker due to inaccurate judgement of the start and the end of sensitive operation. Then we have the following results:

$$T_{L1} = W_{L1} \times T_R + T_{SO} + W_{L1} \times T_{RC} \tag{10}$$

$$T_{L2} = W_{L2} \times T_R + T_{SO} + W_{L2} \times T_{RC} \tag{11}$$

$$T_{L3} = W_{L3} \times T_R + T_{SO} + W_{L3} \times T_{RC} \tag{12}$$

In our defense, cache cleansing is fulfilled by issuing memory access to eviction sets. In order to evict all cache lines of cache sets corresponding to a SP, at least one whole page of the eviction sets should be totally accessed. Then T_{CC} can be expressed as:

$$T_{CC} = W_{L1} \times 2^{N_{PAGE} - N_{LINE}} \times T_R \tag{13}$$

Eq. (13) means that total time needed for cleaning all cache sets of a SP is product of cache's associativity, number of sets and T_R. Besides, with T_C representing the time used for copying a memory block which maps to a single cache line, T_{CP} can then be expressed as:

$$T_{CP} = 2^{N_{PAGE} - N_{LINE}} \times T_C \tag{14}$$

With experience, we know that $T_R \approx T_C$, meaning that speed of memory reading and that of memory copying are very much close. For practical demonstration, we use AIDA64 [27] to test reading and copying speed of DDR3 RAM of five randomly selected PCs which are shown in the following table.

Table 3. Information of five selected PCs

Machine	CPU
ThinkCentre M8500t-N0000	QuadCore Intel Core i7-4790, 3.6GHz
H3060	Intel Core i3 6100, 3.7GHz
M4000e	Intel Core i5 6500, 3.2GHz
ThinkPad T440	Intel Core i7-4500 U, 1.80GHz
ThinkPad x201i	Inter Core i3 CPU M 380, 2.53GHz

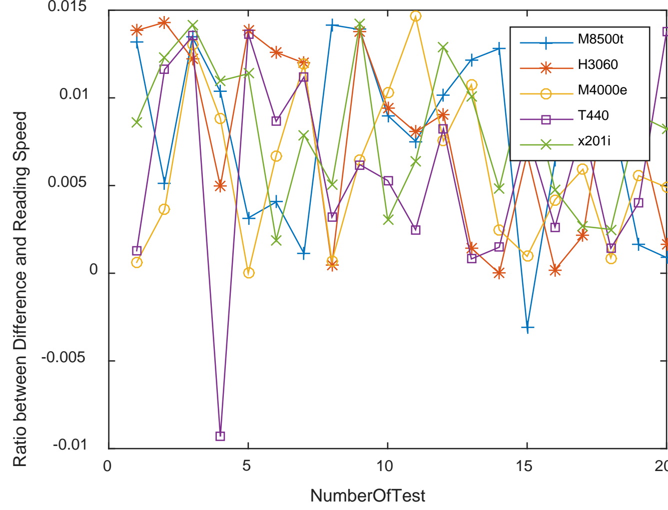


Fig. 4. Test of difference between reading and copying speed using AIDA 64

For each PC, we test the memory reading and copying speed for 100 times, and compute the ratio between the difference of these two speeds and the larger reading speed, which is shown above in **Fig. 4**. From **Fig. 4**, we can find that although the reading speed is slightly larger than the copying speed, they have tiny difference (all values are less than 1.5%) compared to their own value. Therefore, we can assume $T_R \approx T_C$ in this paper.

For T_{CR}, we can just ignore it because in the effort of dynamic remapping, number of memory operations in T_{CR} is far less than other operations, so we have $T_{CR} << T_{CC}$ and $T_{CR} << T_{CP}$.

At this time, we can get the formalized result of our defense for each case below.

- **Defense for L1 cache attack**

In case of L1 cache attack, our framework uses cache cleansing to hinder location of sensitive data, so the security improvement is single operation of scanning for T_{L1}. With Eq. (10) we can get:

$$R_{EE}^{L1} = W_{L1} \times T_R + T_{SO} + W_{L1} \times T_{RC} \tag{15}$$

Its defense overhead is spent on cleaning cache corresponding to certain sensitive memory page as T_{CC}. With Eq. (13) we can get its expression as:

$$R_{EO}^{L1} = W_{L1} \times 2^{N_{PAGE} - N_{LINE}} \times T_R \tag{16}$$

Then we can get its defense efficiency:

$$R_{EI}^{L1} = \frac{W_{L1} \times T_R + T_{SO} + W_{L1} \times T_{RC}}{W_{L1} \times 2^{N_{PAGE} - N_{LINE}} \times T_R} \tag{17}$$

- **Defense for L2 cache attack**

Following the same procedure as in section 5.2, we can get the defense effort and effect of the remapping operation. Following Eqs. (7) (8) and (9), we have:

$$R_{EO}^{L2} \approx 2^{N_{PAGE} - N_{LINE}} (3T_C + 2W_{L2} \times T_R) \tag{18}$$

$$R_{EE}^{L2} = 2^{\log N_{L2} - (N_{PAGE} - N_{LINE}) - 1} (W_{L2} \times T_R + T_{SO} + W_{L2} \times T_{RC}) \tag{19}$$

$$R_{EI}^{L2} \approx \frac{2^{\log N_{L2}-2(N_{PAGE}-N_{LINE})-1}(W_{L2} \times T_R + T_{SO} + W_{L2} \times T_{RC})}{3T_C + 2W_{L2} \times T_R} \tag{20}$$

- **Defense for LLC attack**

With Eqs. (7) (8) and (9), we can change transform the results of section 5.2 into final expressions:

$$R_{EO}^{L3} \approx 2^{N_{PAGE}-N_{LINE}}(3T_C + 2W_{L3} \times T_R) \tag{21}$$

$$R_{EE}^{L3} = 2^{\log N_{L3}-(N_{PAGE}-N_{LINE})-1}(W_{L3} \times T_R + T_{SO} + W_{L3} \times T_{RC}) \tag{22}$$

$$R_{EI}^{L3} \approx \frac{2^{\log N_{L3}-2(N_{PAGE}-N_{LINE})-1}(W_{L3} \times T_R + T_{SO} + W_{L3} \times T_{RC})}{3T_C + 2W_{L3} \times T_R} \tag{23}$$

6.2 Comparison of Dynamic Remapping with Cache Cleansing

With above formalized result in section 6.1, we are able to compare dynamic remapping with cache cleansing theoretically in cases that dynamic remapping and cache cleansing can both work. We still take LLC attack defense as an example, while L2 cache attack defense can be carried out in completely the same way. If we use cache cleansing in LLC attack defense, its defense efficiency turns out to be:

$$R_{EI}^{L3-CC} = \frac{W_{L3} \times T_R + T_{SO} + W_{L3} \times T_{RC}}{W_{L3} \times 2^{N_{PAGE}-N_{LINE}} \times T_R} \tag{24}$$

With Eq. (23) and (24), as well as $T_R \approx T_C$, we can get that:

$$\frac{R_{EI}^{L3-DR}}{R_{EI}^{L3-CC}} \approx \frac{2^{\log N_{L3}-(N_{PAGE}-N_{LINE})-1} \times W_{L3} \times T_R}{3T_C + 2W_{L3} \times T_R} \approx \frac{2^{\log N_{L3}-(N_{PAGE}-N_{LINE})-1}}{\frac{3}{W_{L3}} + 2} \tag{25}$$

In current hardware platforms, we always have $N_{PAGE} = 12$, $N_{L3} \geq 2^{18} / 2^6 = 2^{12}$, $W_{L3} \geq 4$ and $N_{LINE} = 6$. These conditions also satisfy the case of L2 cache, so the following derivation applies for that case. Then with Eq. (25) we can infer that:

$$\frac{R_{EI}^{L3-DR}}{R_{EI}^{L3-CC}} \geq \frac{2^{12-(12-6)-1}}{\frac{3}{4} + 2} >> 1 \tag{26}$$

In case of L2 cache attack defense, we will get the same result. Thus, we have formally demonstrated that dynamic remapping is more efficient than cache cleansing.

7. Evaluation

In this section, we will first give quantitative values about the effort and effect of each defense operation in CacheSCDefender with a comparison between them. Then we will determine the defense interval suitable for dynamic remapping and cache cleansing. At last, we will compare our method with Düppel [11], an existing defense method using periodic cache cleansing, which shows the improvement realized by CacheSCDefender.

7.1 Evaluation and Comparison of Three Defense Operations

For experimental setup, we use the above Lenovo ThinkCentre M8500t-N0000 desktop used in 6.1 to simulate some of the parameters in our evaluation. Here are some parameters about the experimental environment:

Table 4. Parameters of experimental environment

Item	Value
CPU	QuadCore Intel Core i7-4790, 3.6GHz
Number of cores	4
L1 cache	Size:32KB*2*4; Associativity:8
L2 cache	Size:256KB*4; Associativity:8
L3 cache (or LLC)	Size:8192KB; Associativity:16
Size of cache line	64B
Speed of reading memory	10025 MB/s
Speed of copying memory	9916 MB/s
Speed of reading cache	L1:180.61 GB/s; L2:95588 MB/s; L3:73173 MB/s

Based on the above values, we can calculate values of the following two variables:

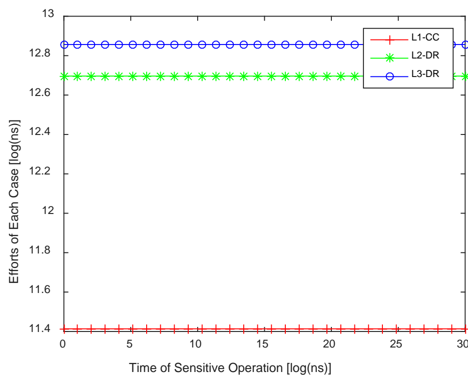
$$T_R^{line} = 64 / \left(10025 * 2^{20}\right) = 6.08829ns$$

$$T_C^{line} = 64 / \left(9916 * 2^{20}\right) = 6.15522ns$$

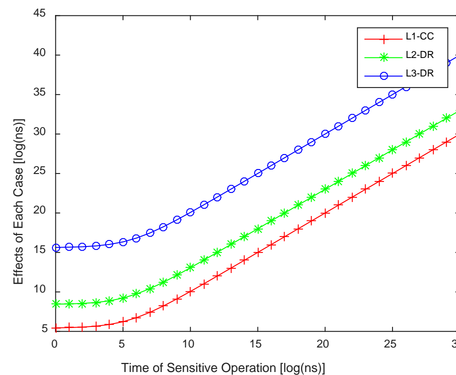
From **Table 4**, we can infer the fact that $T_{RC}^{line} << T_R^{line}$, and $T_{RC}^{line} << T_C^{line}$. This can be explained by the function of cache, which is used to balance the difference of speed between main memory and central processor. So we can further ignore T_{RC}^{line} in expressions of effort and effect of our defense operations in section 6.1.

For T_{SO}, it is quite different for different sensitive operations. So in this case, we can get the quantitative result of our defense in **Fig. 8** with variation of T_{SO}.

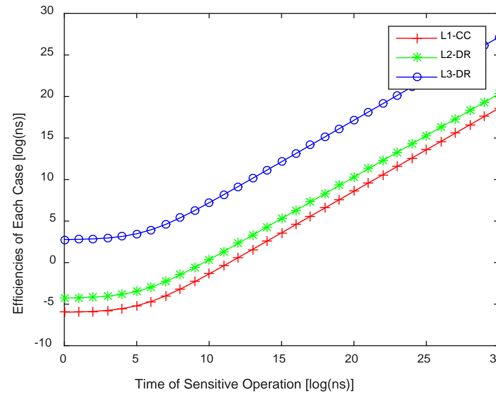
Fig. 8(a) shows the change of defense overhead in each case with the change of T_{SO}, **Fig. 8(b)** shows the change of security improvement, and **Fig. 8(c)** shows the change of defense efficiency. From **Fig. 8(a)**, we can know that efforts of each defense operation are constant with the change of T_{SO}. And L3-DR costs more than L2-DR, while all the two dynamic remapping operations cost more than L1-CC. This can be explained that dynamic remapping not only conducts memory reading to clean cache which takes the same time as cache cleansing, but also spends a lot of time copying memory page and other memory operations, which will consume more time. And since dynamic remapping for LLC reads and copies more memory than that for L2 cache, L3-DR would definitely more expensive than L2-DR. However, even in the most expensive case, single operation L3-DR will only take less than $2^{13} = 8192ns$, which would bring tiny influence to routine operation of VMM. It shows that our defense is practical for deployment.



(a) Changing curve of defense overhead



(b) Changing curve of security improvement



(c) Changing curve of defense efficiency

Fig. 8. Result of quantitative evaluation for CacheSCDefender. L1-CC is cache cleansing for L1 cache; L2-DR represents dynamic remapping for L2 cache; and L3-DR means dynamic remapping for LLC

In **Fig. 8(b)**, we can get that efforts are increasing as T_{SO} increases. This is due to the fact that each Prime+Probe operation needs to wait for end of sensitive operation, and when it takes longer time, the attacker has to spend more time completing single operation. We can find out that for single operation, L3-DR brings us with most security improvements, while L1-CC still brings least benefits. When $T_{SO} = 2^{30} ns \approx 1s$ which is one of common cases, enforcing L3-DR will make the attacker spend at least $2^{40} ns \approx 1000s$ to complete locating of only one piece of sensitive data, although enforcing L1-CC will cost the attacker at least $2^{30} ns \approx 1s$ to conduct the locating job. It indicates that dynamic remapping that we propose in this paper brings us with greater security improvement, compared with traditional cache cleansing method

When comparing all curves in **Fig. 8(c)**, we will find out that L3-DR is most efficient. On the contrary, efficiency of operation L1-CC is worst due to the fact it costs cleaning caches of a whole memory page but just takes the attacker a single Prime+Probe operation to confirm the location. Besides, **Fig. 8(c)** demonstrates quantitatively that dynamic remapping operations are more efficient than cache cleansing.

7.2 Defense Interval

After evaluation of single defense operation, it is time for working out the interval between two defense operations. Since cache cleansing does not change the position of security-critical memory, it only affect the size and accuracy of data that the attacker infer from side-channel attacks, which is another field of defense. So we just set its operation interval the same as that of dynamic remapping.

Our method of dynamic remapping is designed based on the security game between the attacker and the defender. For the attacker, we assume that he uses sequential scanning over all possible cache areas to locate positions of security-critical memory. Then we assume that each scanning of one certain position for security-critical memory of secret i costs time of Δt_i. Considering the multi-core architecture, we further assume the ability of scanning l_i cache positions at the same time during Δt_i, and there are totally L_i possible positions. Based on these assumptions, we can get that the total time for the attacker to scan the whole cache to

identify those for security-critical memory of secret i is $T_i = \Delta t_i \times \left\lceil \frac{L_i}{l_i} \right\rceil$. If memory is not remapped during T_i, the correct position would be found before this time.

Our basic principle is to remap target physical page before this end time. Intuitively, we can conduct remapping operation at the time of $\frac{T_i}{2}, \frac{T_i}{3}, \ldots, \frac{T_i}{k} (\mathrm{k} = 2, 3, \ldots)$. It is obvious that the shorter the time between remapping is, the safer secret *i* is. However, shorter time means more operations of remapping which costs more resource. Hence, we need to balance between security and performance. A shorter interval is advised for a lower security level environment, while a more secure platform is suitable for a longer interval. So what is best time interval for those in an unknown/initial security situation? In order to compare different choices, a new indicator E_t^k showing the effect of defense is proposed in this paper. It is the ratio between probability of failure for the attacker's scanning operations and times of the defender's remapping operations at time t for interval $\frac{T_i}{k}$. Then it can be express as:

$$\mathrm{E}_t^k = \frac{(\frac{k-1}{k})^n}{n}, t = \frac{nT_i}{k} \tag{27}$$

Here is the reason: When $t = \frac{T_i}{k}$, if the security-critical memory for secret i is initially located in the first $\frac{1}{k}$ part of all possible positions, the attacker is able to locate the memory, so success probability is $\frac{1}{k}$. When $t = \frac{2T_i}{k}$, if the security-critical memory for secret *i* is located in the second $\frac{1}{k}$ part of all possible positions after one remapping operation, the attacker is able to locate the memory, so success probability is $\frac{1}{k} + \frac{k-1}{k} \times \frac{1}{k}$. Accordingly, we have success probability of $\frac{1}{k} + \frac{k-1}{k} \times \frac{1}{k} + \ldots + (\frac{k-1}{k})^{n-1} \times \frac{1}{k} = 1 - (\frac{k-1}{k})^n$ when $t = \frac{nT_i}{k}$. So failure probability is $(\frac{k-1}{k})^n$, with which we can get E_t^k. With different *k*, we compare different results at time of *t* which aligned at the boundary of T_i since all of these options could only be comparable at such times, which is shown in **Fig. 8** below.

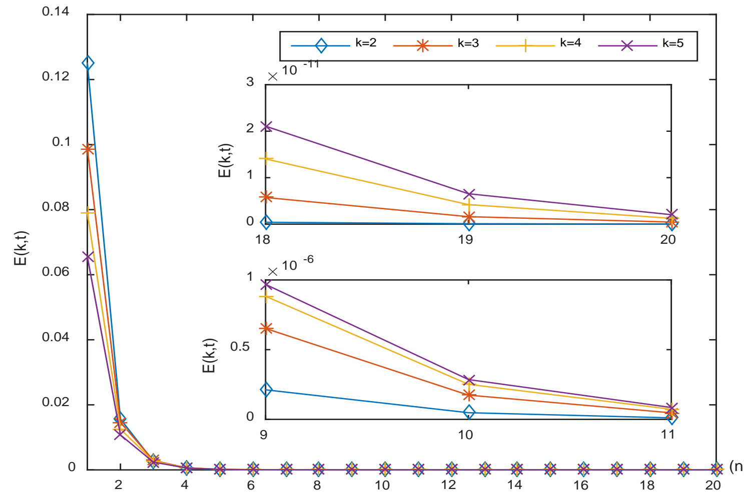


Fig. 9. Simulation results of defense when k=2, 3, 4 and 5

Fig. 9 shows how E_t^k changes after remapping for n times of T_i with different *k*. From **Fig. 9** we can know that at the beginning when remapping is carried out a few times (*n* is small), a smaller *k* can bring with a larger E_t^k, which means more efficient protection. However, it can be found that E_t^k of small *k* decreases faster than that of large *k*, and when remapping is performed more times (*n* is relatively large, see subfigures in **Fig. 9**), larger *k* in turn provides more efficient protection. This result indicates that for small number of remapping operations, smaller intervals brings with relative worse performance, but turns out better as number of operations increases. So we advise a large *k* for an security-unknown/initial environment, such as 16 as a matter of experience.

Based on the above analysis, we can further set the range $[MIN_{Interval}, MAX_{Interval}]$ as $(0, \mathrm{T}_i]$. $Interval[i]$ can be $\frac{T_i}{16}$ as an initial value. When an attack is detected, $Interval[i]$ should be decreased. In this paper, it is randomly selected in the range $[\frac{T_i}{k_t+1}, \frac{T_i}{k_t}](k_t = 1, 2, ...)$ ($\frac{T_i}{k_t}$ is the lower bound of current interval as it is in range $[\frac{T_i}{k_t}, \frac{T_i}{k_t - 1}]$) in order to prevent the attacker to infer the value of $Interval[i]$. When there is no attack found in a time duration of T_{safe} (24 hour as in our system), we increase the interval, and randomly select it in range $[\frac{T_i}{k_t - 1}, \frac{T_i}{k_t - 2}](k_t = 3, 4, ...)$.

7.3 Comparison with Existing Method

We will compare our method with Düppel in this section. Düppel is a cache cleansing system developed to defend against cache-based side-channel attacks. It repeatedly cleans the whole L1 cache (or per-core L2 cache, if present) alongside the execution of its tenant workload, at a pace that it adjusts based upon the possibility with which timings reflecting the workload

execution could actually have been observed from another VM. The interval between two operations is different in “sentinel mode” and “battle mode”.

In our experiment, we compare its defense operation with ours for L2 cache because it does not involve LLC, and we also use cache cleansing for side-channel attacks on L1 cache. Besides, we use the same experimental setup in section 7.1. Since Düppel cleans the whole L2 cache, its defense overhead is:

$$R_{EO}^{D} = W_{L2} \times N_{L2} \times T_{R} \tag{28}$$

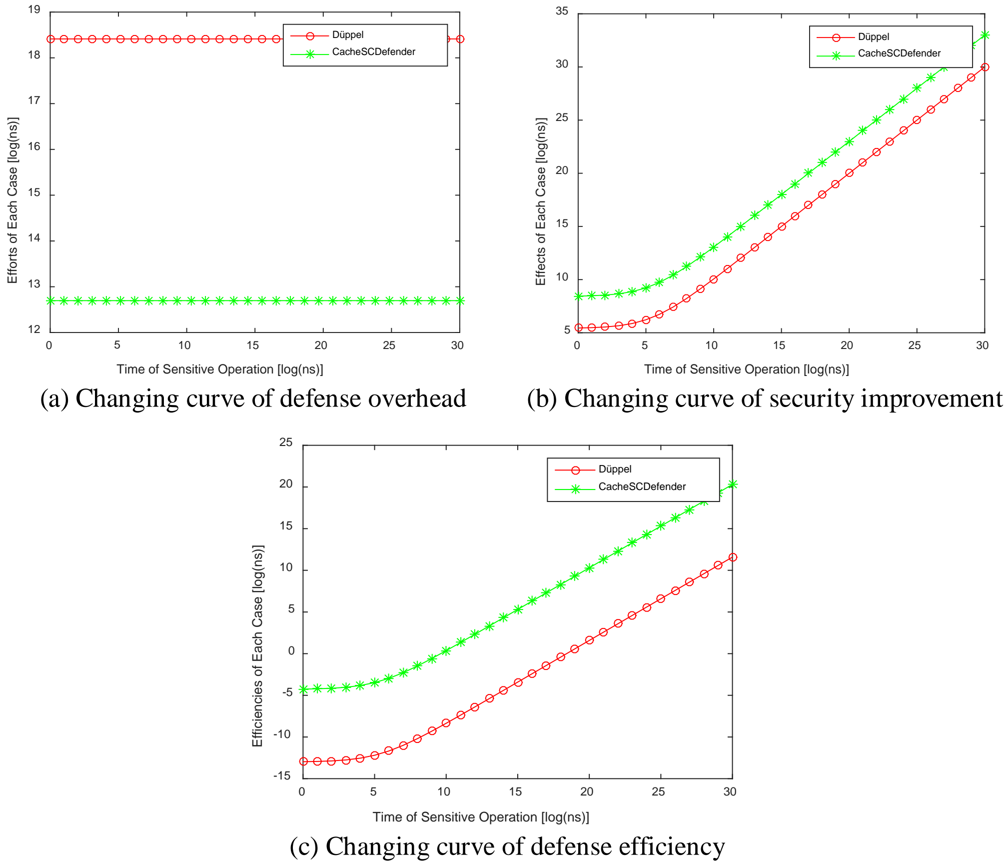
After cleaning cache, the security improvement is:

$$R_{EE}^{D} = W_{L2} \times T_{R} + T_{SO} + W_{L2} \times T_{RC} \tag{29}$$

So with Eq. (28) and Eq. (29), we can get the defense efficiency as follows:

$$R_{EI}^{D} = \frac{W_{L2} \times T_{R} + T_{SO} + W_{L2} \times T_{RC}}{W_{L1} \times N_{L2} \times T_{R}} \tag{30}$$

Furthermore, we can get the defense overhead, security improvement and defense efficiency from section 6.1. Using the experimental environment in 8.1, we can compare these three aspects of our method and Düppel with variation of T_{SO}, as is shown in **Fig. 10** below.



(a) Changing curve of defense overhead (b) Changing curve of security improvement

(c) Changing curve of defense efficiency

Fig. 10. Result of quantitative comparison between CacheSCDefender and Düppel

Fig. 10(a) shows the change of defense overhead in each case with the change of T_{SO}, **Fig. 10(b)** shows the change of security improvement, and **Fig. 10(c)** shows the change of defense efficiency. From **Fig. 10(a)**, we can know that efforts of Düppel and CacheSCDefender are

constant with the change of T_{SO}, and Düppel obviously costs more than CacheSCDefender. This can be explained that Düppel cleans the whole L2 cache while CacheSCDefender conducts memory operation just on protected page. In **Fig. 10(b)**, we can get that efforts are increasing as T_{SO} increases. This is due to the fact that each Prime+Probe operation needs to wait for end of sensitive operation, and when it takes longer time, the attacker has to spend more time completing single operation. We can find out that for single operation, CacheSCDefender brings us with more security improvements. When $T_{SO} = 2^{30} ns \approx 1s$ which is one of common cases, enforcing CacheSCDefender will make the attacker spend at least $2^{33} ns \approx 8s$ to complete locating of only one piece of sensitive data, and enforcing Düppel will cost the attacker about $2^{30} ns \approx 1s$ to conduct the locating job. It indicates that CacheSCDefender brings us with greater security improvement, compared with Düppel. When comparing two curves in **Fig. 10(c)**, we will find out that CacheSCDefender is definitely more efficient. On the contrary, efficiency of Düppel is much worse due to the fact it costs cleaning the whole L2 cache but just takes the attacker a single Prime+Probe operation to confirm the location.

From the above analysis we can find that our method is far more efficient than Düppel in preventing cache-based side-channel attacks on L2 cache.

8. Conclusion

In this paper, we propose CacheSCDefender, an attack-aware comprehensive VMM-based defense framework to defend against all levels of cache attacks. According to which level of cache the attacker is targeting at, we divide the attacking scenes into three cases, two of which can be handled by dynamic remapping that is a new random permutation method we propose in this paper. Since it is not applicable in the third case, we use traditional cache cleansing, a less optimal defense as a supplement. Then we formalize our defense overhead and security improvement in each case, based on which we provide formal quantitative demonstration for that dynamic defense is more effective than cache cleansing, and compare the efficiencies of three remapping operations, the latter of which is used for guiding scheduling of dynamic remapping. Analytical and experimental results show that our defense is not only comprehensive and effective, but also practical for deployment. It should be pointed out that our formalization model is the first model to quantitatively evaluate defense method for cache attacks, and it can be applied to other cases, such as defense with adding noise and defense against other side-channel attacks.

Of course, we admit that our work is not that perfect. On the one hand, our defense needs fine-grained attack detection, while bad resolution of current methods limits their application in more precise defense. In order to solve this problem, we may expand our defense to the guest's operating system level where we can interfere with the whole address translation process, thus providing move precise protection. On the other hand, our defense does not mitigate the basis of side channels, that is, co-residency, which might facilitate other side-channel attacks. To this end, we can combine other protection mechanisms such as VM migration with our method to provide a comprehensive defense.

References

[1] Amazon EC2. Article (CrossRef Link)
[2] Microsoft Azure. Article (CrossRef Link).

[3] Rackspace. Article (CrossRef Link)
[4] Tromer E, Osvik D A, Shamir A., "Efficient Cache Attacks on AES, and Countermeasures[J]," *Journal of Cryptology*, 23(1):37-71, 2010. Article (CrossRef Link)
[5] Bernstein D J., "Cache-timing attacks on AES[J]," *Vlsi Design IEEE Computer Society*, 51(2):218 – 221, 2005.
[6] Irazoqui G, Eisenbarth T, Sunar B, "S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing -- and Its Application to AES[C]," in *Proc. of IEEE Symposium on Security & Privacy. IEEE*, p. 591-604, 2015. Article (CrossRef Link)
[7] Yarom Y, Falkner K, "Flush+Reload: a high resolution, low noise, L3 cache side-channel attack[C]," in *Proc. of 23rd USENIX Security Symposium (USENIX Security 14)*, p.719-732, 2014.
[8] Liu F, Yarom Y, Ge Q, Heiser G, Lee R B, "Last-level cache side-channel attacks are practical[C]," in *Proc. of IEEE Symposium on Security and Privacy*, p. 605-622, 2015. Article (CrossRef Link)
[9] Wang Z, Lee R B, "A novel cache architecture with enhanced performance and security[C]," in *Proc. of 2008 41st IEEE/ACM International Symposium on Microarchitecture. IEEE*, p. 83-93, 2008. Article (CrossRef Link)
[10] Wang Z, Lee R B, "New cache designs for thwarting software cache-based side channel attacks[J]," *ACM Sigarch Computer Architecture News*, 35(2):494-505, 2007. Article (CrossRef Link)
[11] Zhang Y, Reiter M K, "Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud[C]," in *Proc. of ACM Sigsac Conference on Computer & Communications Security*. p. 827-838, 2013. Article (CrossRef Link)
[12] Kim, Taesoo, Peinado, Marcus, Mainar-Ruiz, Gloria, "STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud[C]," in *Proc. of USENIX Conference on Security Symposium. USENIX Association*, p. 352-353, 2012.
[13] Carlet C, Guilley S, "Complementary Dual Codes for Counter-Measures to Side-Channel Attacks[M]," *Coding Theory and Applications. Springer International Publishing*, 97-105, 2015. Article (CrossRef Link)
[14] Blömer J, Guajardo J, Krummel V, "Provably Secure Masking of AES[C]," in *Proc. of International Conference on Selected Areas in Cryptography. Springer-Verlag*, p. 69—83, 2004. Article (CrossRef Link)
[15] Crane S, Homescu A, Brunthaler S, et al, "Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity[C]," in *Proc. of NDSS Symposium*, 2015.
[16] Vattikonda B C, Das S, Shacham H, "Eliminating fine grained timers in Xen[C]," in *Proc. of ACM Workshop on Cloud Computing Security Workshop. ACM*, p. 41-46, 2011. Article (CrossRef Link)
[17] Varadarajan V, Ristenpart T, Swift M, "Scheduler-based defenses against cross-VM side-channels[C]," in *Proc. of USENIX Conference on Security Symposium. USENIX Association*, 2014.
[18] Shi J, Song X, Chen H, et al, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring[C]," in *Proc. of IEEE/IFIP, International Conference on Dependable Systems and Networks Workshops. IEEE Computer Society*, p. 194 – 199, 2011. Article (CrossRef Link)
[19] Raj H, Nathuji R, Singh A, et al, "Resource management for isolation enhanced cloud services.[C]," in *Proc. of ACM Cloud Computing Security Workshop, CCSW 2009*, Chicago, Il, USA, p. 77-84, 2009. Article (CrossRef Link)
[20] Li P, Gao D, Reiter M K, "StopWatch: A Cloud Architecture for Timing Channel Mitigation[J]," *ACM Transactions on Information & System Security*, 17(2):1-28, 2014. Article (CrossRef Link)
[21] Hu W M, "Lattice Scheduling and Covert Channels[C]," in *Proc. of Research in Security and Privacy, 1992. Proceedings. 1992 IEEE Computer Society Symposium on. IEEE Xplore*, p. 52-61, 1992. Article (CrossRef Link)
[22] Kong J, Aciicmez O, Seifert J P, et al, "Hardware-software integrated approaches to defend against software cache-based side channel attacks[J]," p. 393-404, 2009. Article (CrossRef Link)

[23] Blömer J, Krummel V, "Analysis of Countermeasures Against Access Driven Cache Attacks on AES[M]," *Selected Areas in Cryptography. Springer Berlin Heidelberg*, p. 96-109, 2007. Article (CrossRef Link)
[24] Moon S J, Sekar V, Reiter M K, "Nomad:Mitigating Arbitrary Cloud Side Channels via Provider-Assisted Migration[C]," p. 1595-1606, 2015. Article (CrossRef Link)
[25] Payer M, "HexPADS: A Platform to Detect "Stealth" Attacks[M]," *Engineering Secure Software and Systems*. 2016. Article (CrossRef Link)
[26] Coppens B, Verbauwhede I, De Bosschere K, et al, "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors[J]," 73(7):45-60, 2009. Article (CrossRef Link)
[27] AIDA64. Article (CrossRef Link).
[28] Xen Project. Article (CrossRef Link).
[29] Newsome J, Song D, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software[J]," *Chinese Journal of Engineering Mathematics*, 29(5):720-724, 2005.
[30] Ainapure, B. S., Shah, D., & Rao, A. A, "Understanding Perception of Cache-Based Side-Channel Attack on Cloud Environment," *Progress in Intelligent Computing Techniques: Theory, Practice, and Applications*, Springer, Singapore, pp. 9-21, 2018. Article (CrossRef Link)
[31] Anwar, S., Inayat, Z., Zolkipli, M. F., Zain, J. M., Gani, A., Anuar, N. B., ... & Chang, V, "Cross-VM Cache-based Side Channel Attacks and Proposed Prevention Mechanisms: A survey," *Journal of Network and Computer Applications*, vol. 93, p. 259-279, 2017 Article (CrossRef Link)
[32] Disselkoen C, Kohlbrenner D, Porter L, et al, "Prime+ abort: A timer-free high-precision l3 cache attack using intel TSX[C]," in *Proc. of 26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, p. 51-67, 2017.
[33] García C P, Brumley B B, "Constant-Time Callees with Variable-Time Callers[J]," *IACR Cryptology ePrint Archive*, 2016: 1195, 2016.
[34] Green M, Rodrigues-Lima L, Zankl A, et al, "AutoLock: Why Cache Attacks on ARM Are Harder Than You Think[C]," in *Proc. of 26th USENIX Security Symposium*, 2017.
[35] Schwarz M, Lipp M, Gruss D, et al, "KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks[C]," *NDSS*, 2018. Article (CrossRef Link)
[36] Gruss D, Lettner J, Schuster F, et al, "Strong and efficient cache side-channel protection using hardware transactional memory[C]," in *Proc. of USENIX Security Symposium*, 2017.
[37] Meng W, Tischhauser E W, Wang Q, et al, *Ieee Access*, 6: 10179-10188, 2018. Article (CrossRef Link)
[38] Lin Q, Yan H, Huang Z, et al, "An ID-based linearly homomorphic signature scheme and its application in blockchain[J]," *IEEE Access*, 6: 20632-20640, 2018. Article (CrossRef Link)
[39] Jiang F, Fu Y, Gupta B B, et al, "Deep Learning based Multi-channel intelligent attack detection for Data Security[J]," *IEEE Transactions on Sustainable Computing*, 2018. Article (CrossRef Link)
[40] "Handbook of research on modern cryptographic solutions for computer and cyber security[M]," *IGI Global*, 2016. Article (CrossRef Link)

Chao Yang, born in 1990. PhD candidate in National Digital Switching System Engineering Technology Research Center. His main research interests include cloud computing, reverse engineering and cyber security.

Yunfei Guo, born in 1963. PhD supervisor and professor in National Digital Switching System Engineering Technology Research Center. His main research interests include cloud security, telecommunication network security and cyber security.

Hongchao Hu, born in 1982. PhD, associate professor in National Digital Switching System Engineering Technology Research Center. His main research interests include cloud computing, software-defined network and cyber security.

Wenyan Liu, born in 1986. PhD, lecturer in National Digital Switching System Engineering Technology Research Center. His main research interests include cloud computing, software-defined network and cyber security.