Wireless Pers Commun (2017) 97:4827–4854
DOI 10.1007/s11277-017-4753-7

Preventing from Cross-VM Side-Channel Attack Using New Replacement Method

Sandeep Saxena[1] · Goutam Sanyal[1] · Shashank Srivastava[2] · Ruhul Amin[3]

Published online: 9 August 2017

Abstract As Cloud services are gaining importance, many recent works have discovered vulnerabilities unique to such systems. Specifically, like it promotes a risk of information leakage across virtual machine isolation via side-channels. Cloud environment allows mutually distrusting clients access to the shared hardware simultaneously, which can be termed as the main reason for a side-channel attack (SCA). This paper tries to investigate the current state of side-channel vulnerabilities involving the central processing unit cache and identifies the shortcomings of earlier defenses in a Cloud environment. Through cache-based SCA, fined grained information can be collected by attacker easily, and this information may be used by the attacker to infer meaningful results like a secret key, etc. In this article, we detect the SCA at an earlier stage through flush-reload based statistical techniques which exploit the vulnerabilities of Square and Multiply algorithm. Upon detection of SCA, we proposed random permutation function for cache mapping to hide the pattern of cache replacement policy. Additionally, we take the concept of hypothesis testing, deterministic formalism, and information theory to validate our approach.

Keywords Side channel attack (SCA) · Cloud computing · Virtualization · Random permutation cache

✉ Sandeep Saxena
sandeep.research29@gmail.com

Goutam Sanyal
nitgsanyal@gmail.com

Shashank Srivastava
shashank12march@gmail.com

Ruhul Amin
ruhulamin.hit@gmail.com

1 NIT Durgapur, Durgapur, West Bengal, India

2 MNNIT Allahabad, Allahabad, UP, India

3 Thapar University, Patiala, Punjab, India

1 Introduction

Virtualization concept has moved in the vanguard of the biggest trend in the IT world over the past decade. Whether it is related in running different operating systems on the same machine or partitioning one physical server to many virtual servers, it has provided a pathway to increase the utilization power of the system. Virtualization acts as a cornerstone for cloud computing in the computer industry. As it is recent in the industry, so there had been many security issues related to virtualization, some of the eminent attacks are Service Provider Attack, SCA, and Distributed Denial of Service attack.

The internet has become so vulnerable in the modern era that government, business as well as an individual we are facing the threat of cyber attack. Recently according to Ponemon University USA had 43% cloud data breaches in past year [1]. Cache based SCA first came into lime light in the year 2009 but was declared the 4th most threatening attack in 2012 and the year 2016, this has been declared the topmost dangerous attack by cloud security alliance named as shared Technology issue [2]. This attack utilizes the basic concept of sharing pages and cache memory. So we have initially tried to simulate the attack and then proposed ideas that may provide a pathway for countermeasure of this cache-based SCA.

Due to the advancement in the cloud architecture, that provides a shared environment for many users to access simultaneously. This sharing of the resource among different user can be problematic. When two distrusted user work on common hardware that is shared in the cloud environment, the malicious user can setup of SCA to learn about the other user activities. From this learned activities, the malicious user can predict user information that can cause great security threat to the user.

As a new design model, cloud computing brings with it a unique set of features and vulnerabilities. Specifically, the Cloud introduces the concept of mutually distrusting co-resident clients to execute simultaneously on common hardware. This is a relatively new concept in computing. As one might imagine, providing co-residence for clients has brought to light a new set of vulnerabilities in the model. Specifically, the attackers use of hardware as a side-channels to gain information about data and functionality that they should not have access to. Such attacks exploit co-resident systems by inferring software functionality from observed hardware phenomena. This allows an attack to be performed in any context where the attacker and victim have access to the same hardware, so long as proper safeguards are not in place.

While side-channel attacks have existed in the past [3], the novel co-residency feature of Cloud computing makes them particularly more effective in this context. As the attacker is no longer required to gain unauthorized access or otherwise restricted access to the victim hardware, this essentially bypasses all defences against such attacks. Since a side-channel requires the exploitation of a specific piece of hardware, each solution must also be adapted specifically for that hardware channel. This allows us to classify side-channel attacks and defences based on the hardware medium they exploit. The CPU cache is one of the most frequently used pieces of shared hardware, and often deals with sensitive data that makes it one of the most common targets for use in a side-channel attack as it can more easily be used to extract useful data at a high rate. An attack made on this channel is referred to as a cache-based side-channel attack.

Cloud computing is becoming more popular, but the number one concern of technologists heading to the Cloud is security. CPU cache-based side-channel attacks are currently believed to be the most dangerous, among side-channel attacks. In recent years, there have been multiple publications about Cloud-specific vulnerabilities and exploits, specifically the use of side-channels to bypass the virtualization technology used in Cloud systems [4]. Among other exploits, they have recently been used to extract private keys in a Cloud environment and yield high-bandwidth information leakage [5].

Because they are so specific to the medium, the solutions to a side-channel are specific to the hardware medium being exploited. As such, the most significant and robust threat comes from CPU-cache based attacks. Because of this, the scope of this paper is limited to side-channel attacks that exploit the CPU cache.

In response to these attacks, there have been publications attempting to remove such situations. Solutions to such problem require the client using them to modify their software to work with their technology [6, 7] or the underlying hardware [8]. But this solution still has scope for the SCA that is completely removed by enhancing design that thwarts SCA.

Rest of the paper is structured in following sections. Section 2 describes the cache based SCA and models the general procedures of cache based SCA in virtualized cloud environment to extract private key. Section 3 depicts the proposed prevention algorithm based on random permutation functions. In Sects. 4 and 5, we simulated the cache based SCA through in-house simulation and also simulate the proposed prevention algorithm. Sections 6 to 8 devoted to deterministic formal modelling of attack and information theoretic security proof. At the last we concluded the results and future direction in Sect. 9.

2 Related Existing Research Contributions

2.1 General Classes of Side-Channel Attack

Cache Based Attack Here the shared cache is exploited to gain information by analyzing the hit/miss ratio and time required to access certain cache lines.

Timing Attack Attacks based on measuring how much time various computations take to perform.

Power Monitoring Attack Attacks which make use of varying power consumption by the hardware during computation [9].

Electromagnetic Attacks Attacks based on leaked electromagnetic radiation which can directly provide plaintexts and other information.

Data Remanence In this sensitive data are read after supposedly having been deleted.

Differential Fault Analysis In which secrets are discovered by introducing faults in a computation Out of above SCA we will consider the cache-based SCA because it very important attack and it can help the attacker to gain fine-grained information through which user can easily construct the useful information.

2.2 Different Way in Which Information can be Leaked from Cache Memory

Scheduling of Preemptive Type(Uniprocessor) In this attacker and victim VM both uses the same CPU core and when the context switch happens due to pre-emption the attacker can learn from the state in which the victim leaves the cache.

Hyper-Threading Hyper-Threading is a technology that allows threads to run on a single CPU core. The threads share some CPU resources and all of the cores caches. This gives rise to some side channels, so scheduling of threads from different VMs on the same core is generally considered to be unsafe.

Multi-core Under this attacker and victim are running on the different core but concurrently on shared cache memory. The attacker can probe the L3 cache while the victim is running.

2.3 Side Channel Attack

In SCA, attacker creates secret channel over shared hardware resources through which important information are collected by the attacker. In this attack, victim performs a certain operation and this information can be utilized to derive useful information like cryptographic key or any other sensitive information [8]. Side channel attacks were present since a long time but their impact increases by many folds due to the advent of cloud technology, where the basic idea is to share resources to bring down the cost of computing.

While the measured phenomena vary with the specific properties of the hardware in question, any phenomenon that can be reliably correlated to the software function can be used as a side-channel. Examples may include the timing of specific hardware functions or the acoustic properties of a hardware device. Typically, the higher-rate hardware functions are more interesting to explore as side-channels because they can communicate information more quickly, and, therefore, can yield more details about the state of the program in execution.

Previous work has demonstrated that cache-based side-channels can be exploited to perform attack in the Cloud to glean information that service providers do not want users to have access to. Such cases include determining whether two machines are co-resident [10], and more dangerously, the extraction of cryptographic private keys from unwary hosts [11].

2.4 Cache Based Side Channel Attack in Cloud

The SCA dates long back, earlier SCA was in non-cloud environment as described by Lampson [12]. Later with the introduction of cloud architecture the SCA gain a momentum because in cloud architecture all the resources are shared that makes the task of attacker easy to carry out the side channel at any level.

The SCA can be carried in many different ways. It uses the physical property of the hardware to predict the operation performed by the hardware like power measurement attack, cache based attack, execution time-based attack, etc. Among all the most important is the cache-based SCA because this kind of attack helps the attacker to gain fine-grained information that could be very useful to draw significant useful information from it.

Various papers have been published which showed how a SCA is carried out to gain access to important information for which they were not authorized like gain information about the secret key used for encryption. The SCA has been already used to the gain the private key for the RSA encryption [3] in a cloud environment. In this paper, it is explained how the cross-VM SCA is carried out and how with the help of it the secret/private key is generated. The basic approach used is of prime and probe to measure the cache access time of VM by malicious VM. The exponent operation during the encryption is done by square and multiplication method and squaring require more time then multiplication operation and similarly by measuring the cache execution time we can relate which operation is performed and with the help of this information the key can be generated.

Similarly approach by Acicmez [13] in his paper, side-channel cryptanalysis exploits the information leakage from execution time, power consumption or any such side-channels during the computation of cryptographic operations, Cryptographic implementations leak sensitive information because of the physical properties and requirements of the cryptographic implementations and computational environments.

After discussing and analyzing cache-based SCA, we focus on two general procedure of cache-based side channel attack in virtualized cloud environment to extract private key. Here we focus on two type of cache based side channel attack, one is time driven cache attack and the other is trace driven side channel attack.

2.5 Time Driven Cache Attack

Time driven cache attack is also known as timing attack. In this type of attack, execution time acts as a side channel for attack. Memory access time directly depends upon the state of cache. Different types of process or function takes different execution time. Such time difference could be used for performing side channel attack such as deducing cryptographic key (AES, RSA or ElGamal key). Encryption process is the function of plaintext and key, so the number of cache lines accessed during encryption directly depends upon plaintext and key. Similarly decryption process is the function of key and cipher text. So the access of cache lines by decryption process depends upon key and cipher text. Such dependency leads to cache based side channel attack. Attacker deduces the cryptographic key by utilizing the execution pattern as side channel.

Bernstein [14] perform attack on AES key, as S-Box lookup of AES algorithm depends on AES key and plaintext. Here, attacker performs encryption on sufficiently large number of known plaintexts and exploits the execution time differences of S-Box table lookup process and deduce the key after performing offline analysis. Similar to Bernstein, Osvik et al. [15]; performed time driven cache based side channel attack and recovered complete 128 bit of AES key.

2.6 Trace Driven Cache Attack

Prime + Probe based attack strategy is the example of trace driven cache attack. In Prime phase, attacker fills the cache line with its own content and later in Probe phase, attacker measure the access time for accessing its own data from the same memory address (i.e. cache line). If the execution time varies sufficiently large, it means the memory location (i.e. cache line) is evicted by some other process, and the content is fetched from the RAM in place of cache i.e. the victim accessed the pre-image of the same cache lone from the RAM.

Yarom and Katrina [16] demonstrate the FLUSH + RELOAD technique is using to extract private encryption key from victim program. FLUSH + RELOAD is technique is type of PRIME-PROBE technique that depends on sharing pages on victim and attacker process.

What is FLUSH + RELOAD technique? There are spy and victim processes running on different cores. Codes are loaded in shared memory; victim accesses the code and when it executes it, the spy can access it and read data.

Initially, Spy flushes the shared code from the cache. Now the spy waits, while he waits the victim may access or may not access the specific code After spy finishes waiting, it reads the specific data , if it takes a long time then data has come from memory, otherwise it is from cache. The signature is performed using the CRT-RSA algorithm implemented in the libgcrypt shared library. This algorithm uses the Square–Multiply algorithm to perform exponentiation that results in revealing the private-key bits of the victim (see Algorithm 1).

We can see from the above algorithm that if bit of the exponent is 0 then the operation would be square followed by reduce and not followed by multiply and if the bit is 1 then the operation would be square followed by reduce followed by multiply followed by reduce. If we can recover the sequence of operations we can get the bits of the exponent which in our case is the key bits.

Gruss et al. [17] replaced the flush instruction by evict instruction and proposed evict+reload technique, but this technique has no practical application on X86 CPUs. However, it can be used on ARM CPU.

Further, Gruss proposed FLUSH + FLUSH [18] technique. This technique does not perform memory access like other similar techniques, but it exploits the timing pattern of the flush instruction to determine the victim activities.

2.7 Prevention Approaches of Cache Based Side Channel Attack

In the previous section, it is explained how side channel or covert channel is used to pass on important information of victim by attacker VM [13]. On this side channel or covert channel is setup over the shared cache memory. To prevent cache based side channel attack in virtualized environment, in the year 2007, Wang and Lee [19], proposed a hardware solution. In their solution, for each cache line, two new fields are introduced. One is locked field or L field and the other is identification or ID field. Here if the cache line is marked as locked than that cache line cannot be evicted by any other VM expect for the VM which has marked it locked and for this, the ID of VM is also stored along with the field which is locked. The limitation of this scheme is that it increases the hardware cost as hardware addition is required to existing scheme and if the number of VM increases then the ID field length is also increases.

Kong et al. [20] identified that the cache interference is the main cause of side channel attack. Authors proposed randomized approach to randomize the cache interference to short out the problem of cache interference. Now it is difficult for an attacker to gain sensitive information from the pattern of cache hits and misses. In their approach, they presented two new cache designs, one is partition locked cache and the other is random permutation cache. However, the approach provides security against SCA but at the cost of large performance degradation. Locking of cache is not a performance friendly solution as due to this cache utilization is degraded.

Zhang et al. [21] discussed the other approach of co-residency detection, which brought out the approach of securing the cloud by using a VM from the server side to detect any suspicious actions caused by any attacker and to stop them. This paper suggested that the monitoring of cloud can be done in the same way as any attacker monitors the cloud to find its victim or victims. The paper proposed prototype system that allows customers who use cloud computing services to confirm that their data is safe from attacks using the same service provider.

Raj et al. [7] proposed page coloring approach for the prevention of cache based SCA. In this approach, cache is partitioned into different region. Each region is assigned a specific color. Each VM can access the cache region of specified color for which it is allowed to access. With this idea, authors remove the problem of cache interference and hence no cache based side channel attack can be carried out to gain sensitive information. This solution can be seen as the guard against the cache based side channel attack but the problem is that this protection is gained at huge cost of performance degradation and only up to certain limit the performance degradation can be traded with protection.

Another approach to overcome the above problem is given by the Kim et al., in this paper, a new technique is proposed and it is called the stealthmem. In this technique, the basic idea is drawn from the page coloring technique explained above [22]. Here the cache is partitioned into different region as in page coloring technique, and the partition is based on the number of the cores of processor. In this way, the drawback associated with the page coloring technique has been removed and the numbers of partition is now fixed and depends on the number of cores in place of number of VMs. This approach provides isolation for the cache region but still two VMs can access the same core that would create a security question. In a scenario where two VMs, one is attacker VM and the other is victim VM operate on the same core in alternate fashion, provides a suitable condition for carrying cache based SCA.

Zhou et al. [23] proposed a mitigation approach for access driven side channel attack in Last level caches (LLC). This approach dynamically manages shared physical memory pages to disable sharing of LLC lines. This approach prevents from the FLUSH+RELOAD and Prime + Probe based attack in LLC with small performance overhead.

Liu et al. [24, 25] proposed a light-weighted protection system CATalyst for the cloud providers as well as for the customers. This system protects sensitive code and data against side channel attack based on LLC. Compared to page coloring technique CATalyst system design is simpler as CAT provides the isolation of secure and insecure cache partitions. In addition to this, CATalyst is having the feature of memory deduplication [26] which is missing in page coloring scheme.

Recently, Han et al. [9], proposed policy model that protects cloud system from co-resident attack. In this attack, attacker performs side channel attack and extract private key of co-located VM on same server. Authors improved current VM allocation policy and developed new balanced policy previously-selected server-first (PSSF), which creates difficulty for an attacker to co-locate with their targets. The proposed solution uses security metrics for accessing the attack; thereby solution not only mitigates the threat of CSA but also fulfils the requirements of workload balance as well as power consumption.

2.8 Motivation and Contributions

The Internet has become so vulnerable in the modern era that government, business as well as an individual we are facing threat of cyber attack. Recently according to Ponemon University, the USA had 43% cloud data breaches in past year. Cache based SCA first came into limelight in year 2009, but was declared the 4th most threatening attack in 2012, and in the year 2014. In Unisys conference, this attack has been declared the topmost security attack; this attack utilizes the basic concept of sharing pages and cache memory. So we have initially tried to simulate the attack and then proposed some ideas which may provide a pathway for countermeasure of this cache based SCA.

Our article has achieved the following major contributions.

- To verify whether our environment is being attacked or not we have simply generated a large sample of hit-ratios without any cache interference and then generate the normal curve plot. Using the knowledge of confidence interval and hypothesis testing we have predicted whether there is something malicious or not.
- On verifying if the result of attack is positive, then we perform Random permutation during mapping of ram data to cache. Applying this attacker fails to collect relevant

information during probing and he is unable to find a pattern. We have applied XOR of ram address and private key as the random permutation function.

- Initially, what we thought to apply random permutation for every RAM access despite of the fact that we are being attacked or not. Though this was pre-serving the key still it had a lot of overhead, to reduce the overhead we verify after some intervals whether there is attack or not and if there is attack then we do the mapping.

3 Proposed Prevention from Cache Based Side Channel Attack

In order to prevent from cache based SCA first it is needed to detect the possibility of attack. Our detection program (running on monitor VM) continuously monitor the activities performed on the cache. First it checks whether consecutive cache misses occur due to same processes in RAM. Through tag bits in cache address, it is identified that which process in RAM maps to the particular cache line. In set associative cache, mapping is performed from RAM to cache. If it happen, there are two possibilities, first legitimate system process may evicts the cache line so that misses occur or the malicious process evicts the cache line. On the occurring of multiple cache misses from the same process of RAM, may leads to the threat of SCA. When this happen, a random permutation function (RPF) works with (encryption function) (refer Algorithm 1).

Algorithm 1 Square-Multiply Algorithm

```
begin
Function exp-by-squaring (x,n)
if $n < 0$ then return exp-by-squaring ($1/x, -n$);
else if n=0 then return 1;
else if n=1 then return x;
else if n is even then return exp-by-squaring (x 2 , n/2);
else if n is odd then return x * exp-by-squaring (x 2 , (n-1))
End
```

This random permutation function is an inverse function, so to extract the data from the cache, inverse mapping takes placed. In general when a process do not find the data (word) in the cache block, it fetches the block from the RAM to the cache where this word resides, assume according to the locality of references nearby word in the block will be accessed in near future. Random permutation function permutates each word of the block to a different location of the cache. Each word is remapped to a different location.

In Fig. 1, we have shows that word address in cache memory is re-maped using Random permutation function into new word address.

After re-mapping of word in the cache, each word of the same block is re-mapped to a different block of a different set.

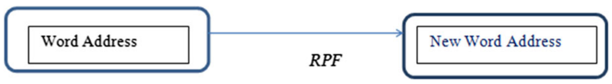

Fig. 1 Replacement of word after applying random permutation function

Before re-mapping

$word_i \in set_i$ where i: 1 → n
$word_i \in block_i$ where i: 1 → k

After re-mapping;

$word_i \in set_j$ where j: 1 → n
$word_i \in block_j$ where j: 1 → k

There is a consequence, after re-mapping of word in different block and different set; there are possibilities of various cache evictions resulting in various cache misses thereby performance overhead. To reduce, cache misses or increase performance, in general various replacement policy are there to solve the problem of cache misses. Upon missing the word in cache, a block consists of missing word is replaced from the RAM to cache. But in our case, the missing word is not replaced by the whole block of words. In our replacement policy, specific word is replaced from the RAM in spite of whole block as in Fig. 2.

In a cloud environment, there are various virtual machines are running on the server. Any virtual machine can perform cache-based SCA machine and steals the cryptographic key. So to perform a check for invoking random permutation function, the system has to learn the pattern of a cache hit and cache miss.

Before applying the Random Permutation Function, we must determine whether the victim VM is being attacked or not that helps in avoiding unnecessary overhead generated by the function if the victim is not being attacked. For this, we apply Hypothesis testing to the hit-ratio. Before we dive into the details, first lets get familiar with some basic concepts.

We apply the central limit theorem to the hit-ratios collected from operations without any malicious cache interference. It states that the arithmetic mean of a large number of random variables, each with a well-defined expected value and well-defined variance, will be approximately normally distributed. We take 100,000 samples of the hit-ratios generated from a code that consists of 4 virtual machines. Each virtual machine has a random probability of being selected to map its data, once selected they map their data on the corresponding address slot on the cache, if another virtual machine data preoccupy the slot then it is cache miss otherwise a hit. At the end overall hit ratio is computed, and a plot is generated which is the usual bell-curve representing a normal distribution. Figure 6 represents the flow of activities.

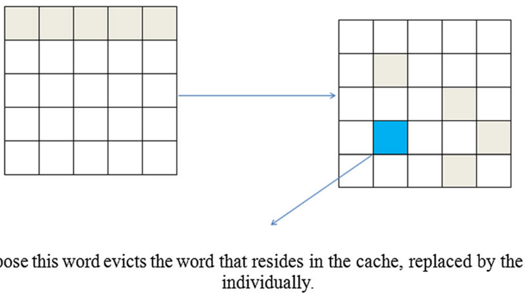
Suppose this word evicts the word that resides in the cache, replaced by the RAM individually.

Fig. 2 New replacement policy for specific word in cache memory

The confidence interval is the interval that is tolerable for any average probabilistic mean. The Central limit theorem is used to calculate the confidence interval.

Suppose $X1, X2, \ldots, Xn$ be a set of n independent variables. X_i has probability distribution $P(x1, x2, \ldots, xn)$ with mean values $(\mu_1, \mu_2, \ldots, \mu_n)$ N is the number of samples. Now if S is standard deviation of all mean values and Z is the normal distribution, then the Confidence Interval (CI) by Central Limit Theorem is shown in Eq. 1.

$$CI = \mu_i \pm Z * \frac{S}{\sqrt{N}} \tag{1}$$

Value of Z varies according to Confidence Interval (CI).

For 90% CI, Z = 1.645
For 95% CI, Z = 1.960
For 99% CI, Z = 2.576

Hypothesis testing is the use of statistics to determine the probability that a given hypothesis is true. There are two hypotheses the Null hypothesis and the alternative hypothesis. We define the Null hypothesis to be hit-ratio of the current execution to be greater than or equal to the population mean of the samples we have collected. Therefore, we apply one-sided tail test to check whether the hit-ratio lies within the 99% confidence interval. If it does not then the virtual machine is being attacked and preventing the attacker from being successful we apply a random permutation function to map the RAM address to some other location so that the attacker cannot determine whether the Square operation is performed or the Multiply operation is performed. Hence, the attacker is unable to determine the bits of the key failing to find the private key of the victim. The approach is depicted in the prevention algorithm.

4 Proposed Random Permutation Function and Prevention Algorithm

The random permutation function should be as less time consuming as possible because it is applied whenever we have to access RAM, thus accessing shared cache. We have taken the random permutation function to be the XOR function. We XOR the RAM address, and the private key of the virtual machine and the new address obtained becomes the address that should be mapped to the cache, thus giving new cache lines to the Square–Multiply operations of the victim virtual machine. The attacker now unaware of this change monitors the previous cache lines and generates the wrong key. The random permutation function is depicted in the Algorithm 2.

Algorithm 2 Proposed Prevention algorithm

```
Input: $hit\_ratio$,$ram\_addr$, $priv\_key$,n
output: $new\_addr$
// The Victim VM before performing any operation checks whether it
is being attacked or not.

if (is . attacked ($hit\_ratio$ , n) = true) then
addr $\leftarrow$ $random\_perm(ram\_addr)$
else
addr $\leftarrow$ $ram\_addr$
is.attacked($hit\_ratio$,n)
begin
$std\_err$ $\leftarrow$ 2.576*(std/sqrt(n))
if($hit\_ratio$ ! mean)then return
true
end
$random\_perm$ ($ram\_addr$)
begin
$new\_addr \leftarrow ram\_addr \oplus priv\_key$
return $new\_addr$
end
```

To verify whether our environment is being attacked or not we have simply generated a large sample of hit-ratios without any cache interference and then generate the normal plot. Using the knowledge of confidence interval and hypothesis testing we have predicted whether there is something malicious or not.

On verifying if the result of the attack is positive, then we perform Random permutation during mapping of ram data to cache. Applying this, attacker fails to collect relevant information during probing, and he is unable to find a pattern. We have applied XOR of ram address and private key as the random permutation function. Algorithm 2 describes the proposed algorithm.

5 Simulation Setup of Cache-Based Side-Channel Attack an Results Analysis

We simulated the cache-based SCA through in-house simulation. First we simulated the memory organization. For this, we have taken the word size of 8 bytes. To simulate cache, we have taken a one-dimensional integer array of size 1024, simulating that the cache can contain up to [1024] cache lines i.e. CACHE [1024].

CACHE simulated as a 1D array of 2^{10} cache lines. It is considered to be one-way set associative, i.e. each set has one block. To simulate RAM, we have taken a one-dimensional integer array of size 65,536, simulating that the RAM can contain up to 65,536 words i.e. RAM [65,536].

We build the simulator on C++, following classes were implemented to simulate virtualized environment (see Fig. 3).

- *victvm* (*Victim VM*) *class* it symbolizes the victim virtual machine.
- *attack* (*Attacker VM*) *class* it symbolizes the attacker virtual machine

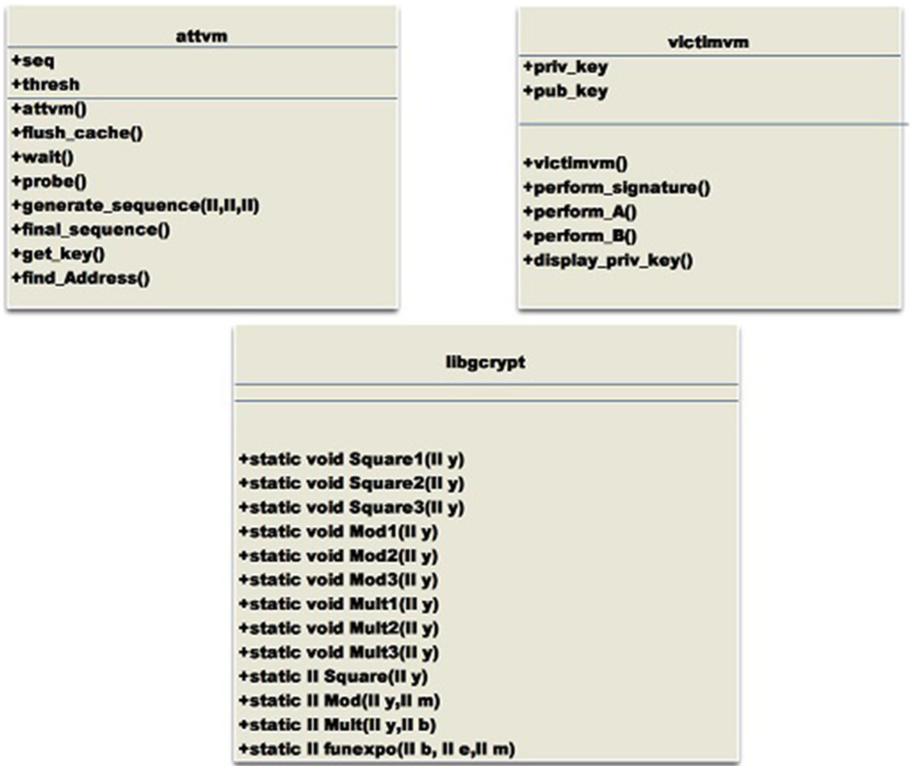

Fig. 3 Class diagram

libgcrypt it symbolizes the libgcrypt library that has all the encryption and decryption function that have been performed.

Functions and Variables used in classes:
Victim VM class:

- *private key* it is private key of victim vm.
- *public key* it is public key of victim vm.
- *victvm*() It is a constructer of class victvm which initializes public and private key of victim class.
- *perform A*() Some operation A performed by victim virtual machine
- *perform B*() Some operation B performed by victim virtual machine.
- *void display priv key*() It is used to display the private key generated by victim.
- *void perform signature*() It is an encryption operation performed by victim virtual machine.

Attacker VM class

- *sequence* It stores the sequence of the operations performed by the victim VM
- *threshold* It is the threshold value of the number of CPU cycles to distinguish between the hit and the miss operation in cache.

- *attvm*() It is a constructer of class attvm which is used to initialize the sequence and the threshold value of attvm class.
- *void flush cache*() It is used to empty the cache.
- *void wait*() It is used for attacker to wait while the victim is performing some action.
- *void probe*() It is used by attacker to probe the cache memory for finding the operation being performed.
- *void generate sequence(long,long,long)* It is used to generate the sequence of operations dynamically.
- *long get key*() It is used to obtain the final key used in the exponent operation.
- *void find address*() It is find the address of the operation in the main memory.

Libgcrypt library class

- *static void Square1(long y)* It is a part of the Square operation (the significance of this function is to perform a single Square operation in multiple stages, meanwhile in parallel the attacker is waiting for the cache to be filled).
- *static void Square2(long y)* Similar to Square1.
- *static void Square3(long y)* Similar to Square1.

static void Mod1(long y, long m) It is a part of the Mod operation(the significance of this function is to perform a single Mod operation in multiple stages, meanwhile in parallel the attacker is waiting for the cache to be filled).

- *static void Mod2(long y, long m)* Similar to Mod1.
- *static void Mod3(long y, long m)* similar to Mod1.
- *static void Mult1(long b, long y)* It is a part of the Multiply operation (the significance of this function is to perform a single Multiply operation in multiple stages, meanwhile in parallel the attacker is waiting for the cache to be filled).
- *static void Mult2(long b, long y)* Similar to Mult1.
- *static void Mult3(long b, long y)* Similar to Mult1.
- *static long funexpo(long b, long e, long m)* It is the function to implement the Square–Multiply algorithm.

In order to simulate the cache-based SCA, we implement three algorithms.

- Simulation of cache-based SCA
- Sample creation of hit and miss ratio.
- Simulation of prevention of cache-based SCA

Initially victim performs the transaction, where it uses its private key for decryption and attacker finds an opportunity to reveal it. Victim performs cryptographic signature operation through libgcrypt library. Cryptography signature exponential operation based on square, multiply algorithm.

Initially attack finds the RAM address of square, reduce and multiple operation, to map the RAM address to the cache address *ram_to_address* () function is implemented.

As victim performs the cryptographic operation, sequences of square, reduce and multiply operation are executed, first these operations are loaded into RAM for execution, victim VM executes the operation taken from RAM. As these operations are executed frequently on the basis of key pattern, so for fast execution, the operations are stored in cache (see Fig. 4).

First victim VM finds out the cache address from the RAM address as in general the mapping form RAM to cache address is set associative and try to access the S, R, M

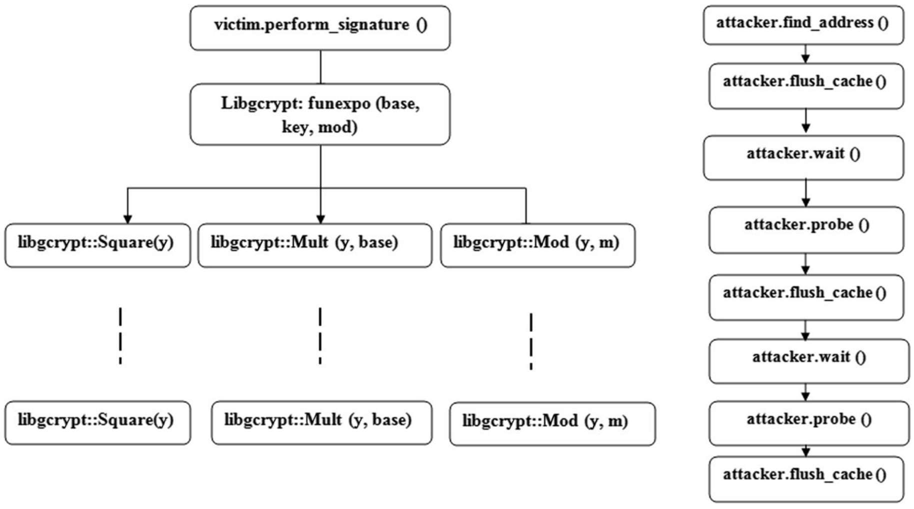

Fig. 4 Process flow of SCA

operations from the cache and if somehow if the operations are not found in the cache i.e. the cache miss occur, the respective operation is again loaded form RAM to cache then executed, but this indirect execution takes some delay as loading operation from RAM takes more time.

Attacker flushes the cache so that the victim has to reload the operation (S, R, M) it is performing into the cache and the attacker will be able to reveal the operation. Now attacker waits while the victim is performing the operation in parallel. The attacker monitors or probes the cache lines corresponding to vulnerable code (Square–Multiply functions) to check which operation was performed during the victim process. In the probe process, first attacker find the physical addresses of square reduce and multiply operation, then find the tag bits and corresponding cache addresses and finds the time to reload the square, reduce and multiply code. After monitoring cache lines, add the performed operation to the sequence with the help of respective loading time of square, multiply and reduce operations. Finally attacker generates the key with the help of sequence generated (see Fig. 5).

In simulation process (see Table 1), first we created the set of virtual machines then we declare private key and public key of the each virtual machine. The targeted victim virtual machine (line 1.5–1.7) performs the signature operation, where it uses its private key and the attacker finds an opportunity to reveal it, we simulated signature operation in line 2.1–2.13.

In simulation phase 2 (see Table 2), we simulated the behaviour of libgcrypt library. As in cryptography signature operation, large exponent computation is required to calculate, that is based on square–multiply algorithm discussed in section. Here Square–Reduce (SR) and Square–Reduce Mult-Reduce (SRMR) operations are called based on binary digits(0 or 1) of the key value.

In simulation phase 3, attacker virtual machine vm performs SCA on selected *victim_vm*.

Table 3 shows the simulation of attacker VM. Line 3.2 denotes string to store the sequence of operations performed by the attacker and the threshold value which

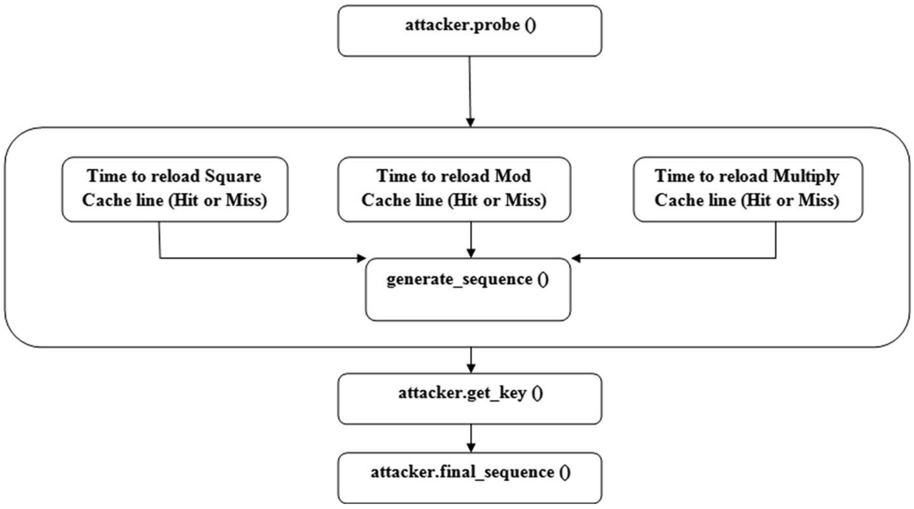
attacker.probe ()
Time to reload Square Cache line (Hit or Miss)
Time to reload Mod Cache line (Hit or Miss)
Time to reload Multiply Cache line (Hit or Miss)
generate_sequence ()
attacker.get_key ()
attacker.final_sequence ()

Fig. 5 Process flow of probe operation

Table 1 Simulation Phase 1: signature operation using Victim vm

Simulation Phase 1:Signature operationusing Victim vm

```
//Create set of virtual machines
1.1 vm[n]// vm[0],vm[1],vm[2]...vm[n]
// Create public , private keys for each vm
1.2 for i: 0 → n
1.3     public[i]= vm[i].$public\_key$
1.4     $private\_key$ = vm[i].$private\_key$
1.5 end for
// Select victim vm to perform attack
1.6 $victim_{vm}$ = vm[i] for any value of i
1.7 $victim_{vm} = victim_{vm}.perform\_signature()$;
1.8
```

distinguishes between a cache miss and a cache hit. Initially we set the seq to NULL string and the threshold value. To flush the cache so that the victim has to reload the operation it is performing into the cache and the attacker will be able to reveal the operation. We simulated the cache flush operation (lines 3.4 and 4.1–4.3) (refer Table 4). Attacker waits while the victim is performing the operation in parallel (line 3.5). In probe operation, the attacker monitors the cache lines corresponding to the vulnerable code (Square–Multiply functions) to check which operation was performed during the waiting process. During probe operation, first we find the physical address of Square, Reduce and Multiply operation and its corresponding tag bits and cache addresses. Then find the time to reload the square code, reduce code and multiply code (lines 5.1–5.6) (refer Tables 5, 6).

Line 3.7 is used to add the operation performed to the sequence with the help of respective loading times Square, Multiply and Reduce operation. After getting the final

Table 2 Simulation Phase 2: signature operation using Victim vm *perform_signature*();

```
Simulation Phase 2:Signature operation using Victim vm
    perform_signature

// Using libgcrypt:funexpo(base,key,mod)
2.1 key = bit[n],bit[n-1],...bit[2],bit[1],bit[0]
2.2 for i: 0 → n
2.3   if bit[i]== 0  //SR
2.4     libgcrypy :: square(y)
2.5     libgcrypt :: mod(y,m)
2.6   endif
2.7 else     //SRMR
2.8     libgcrypt :: square (y)
2.9     libgcrypt :: mod(y,m)
2.10     libgcrypt :: mult(y,base)
2.11     libgcrypt :: mod(y,m)
2.12 endelse
2.13 endfor
```

Table 3 Simulation Phase 3: attack simulation (attackvm)

```
Simulation Phase 3 : Attack simulation (attackvm)

3.1 START
3.2 Input: SET threshold thresh
    Output: Key_String, Key     //Key_String is the sequences
    generated in terms of SRM
3.3 //Attacker vm perform following operation to deduce key
3.4 void flush_cache(); 3.5 void wait();
3.6 void probe ();
3.7 void gererate_sequence();
3.8 string final_sequence();
3.9 long long get_key();
3.10 END
```

sequences of S, R and M, *get_key*() function is used to extract private key through S, R, M sequences (line 3.10).

6 Result Analysis

We test the results in machine having Ubuntu 14.04 operating system and Intel Core I7 6700K processors with 16 GB RAM and 8M cache (see Table 7).

In Fig. 6, we mentioned the statistical data of square reduce and multiply operations. We calculated the average CPU cycles, taken to access S, R, M operations from cache memory. Applying central limit theorem, we calculated the confidence interval for 90, 95 and 99%, and find out the lower and upper range of CPU cycles of corresponding S, R and

Table 4 Simulation Phase 4: attack simulation (attackvm) method definitions *flush_cache*()

Simulation Phase 4 : Attack simulation(attackvm) Method Definitions flush_cache()

// Methods Definitions
4.1.void attvm::*flush_cache*()
{
4.2. //sets the cache to NULL, i.e. flushes the cache.
4.3.memset(CACHE,0,sizeof(CACHE));
}

Table 5 Simulation Phase 5: attack simulation (attackvm) method definitions probe ()

Simulation Phase 5 : Attack simulation (attackvm) Method Definitions probe()

// Methods Definitions
5.1. void attvm::probe()
{
5.2. //finds the time to reload the square code
5.3. tsq=$time\_to\_reload(ram\_add, cache\_add, tag)$;
//finds the time to reload the reduce code
5.4. tmod=$time\_to\_reload(ram\_add, cache\_add, tag)$;
//finds the time to multiply the square code
5.5. tmult=$time\_to\_reload(ram\_add, cache\_add, tag)$;
//adds the operation performed to the seq string.
5.6 *generate_sequence*(tsq,tmod,tmult);
}

M operations. As according to our hypothesis, if CPU cycles to access S, R and M operations exceeds the confidence interval, it may be an attack, as accessing Square, Reduce and multiply operations from RAM takes extra time.

Figure 7 shows the normal distribution curve for S, R and M and a combined sequences of the operations under an attack situation (*Note* this is code generated graph).

7 Formalization of Cache Based Side Channel Attack

Side channel attack can be performed by cache behavioral pattern generated by time, access and traces. In our model, we consider multiple virtual machines on the same hardware, performing RSA encryption/decryption on data. In the meanwhile, attacker VM performs cross-vm SCA to retrieve victim VMs RSA decryption key i.e. private key. RSA or asymmetric key cryptographic operation is performed a large exponential operation. To perform a large exponential operation, Square and Multiply algorithm (consist of square, reduce and multiply functions) is used.

Table 6 Simulation Phase 6: attack simulation (attackvm) method definitions *generate_sequence* (ll tsq,ll tmod,ll tmult)

Simulation Phase 6 : Attack simulation (attackvm) Method Definitions generate_sequence(ll tsq,ll tmod,ll tmult)

```
// Methods Definitions
6.1. attvm::generate_sequence(ll tsq,ll tmod,ll tmult){

6.2. (tsq < thresh)
6.3. seq+='S';
6.4. if(tmod < thresh)
6.5. seq+='R';
6.6. if(tmult < thresh)
6.7. seq+='M';
}
```

Table 7 Confidence intervals of mean of CPU cycles

Operation	Mean of CPU cycles taken	SD	Confidence interval—CI (90%)	Confidence interval—CI (95%)	Confidence interval—CI (99%)
(S)	3385	573.018	±66.65	±79.41	±104.37
			3318.35–3451.65	3305.59–3464.41	3280.63–3489.37
(R)	5894.5	636.671	±74.05	±88.24	±115.96
			5820.45–5968.55	5806.26–5982.74	5778.54–6010.46
(M)	5800	465.34	±54.12	±64.49	±84.76
			5745.88–5854.12	5735.51–5864.49	5715.24–5884.76

All computation is performed on binary data (both key and data in binary form), as we discussed in the previous section. If the key bit is 0 then only SR (Square–Reduce) is called and when the bit is 1 then SRMR (Square–Reduce Multiple–Reduce) is called. By this concept, an attacker can retrieve private key after multiple traces.

Before Starting of formal modeling, we need to keep these behaviors of memory in our mind, the first and foremost behavior is Cache can store a subset of main memory, and Main Memory is logically divided into memory blocks and each memory block map into particular cache location. When memory block is cached, it caches as a whole in a cache line of the same size.

7.1 Formalization of Flush-Reload Technique

In the flush-reload technique, attacker flushes the cache memory and reloads his data (X, Y, and Z) in place of Square (S), Multiply (M) and Reduce (R). He accesses his data after a fixed period and again performs flush-reload. He calculates the hit and miss based on the time required to obtain X, Y, and Z. Suppose X replace S, Y replaces M and Z replace R. Let t1 time is required in case of a cache hit (data occur in cache) and t2 time is required when cache miss occur.

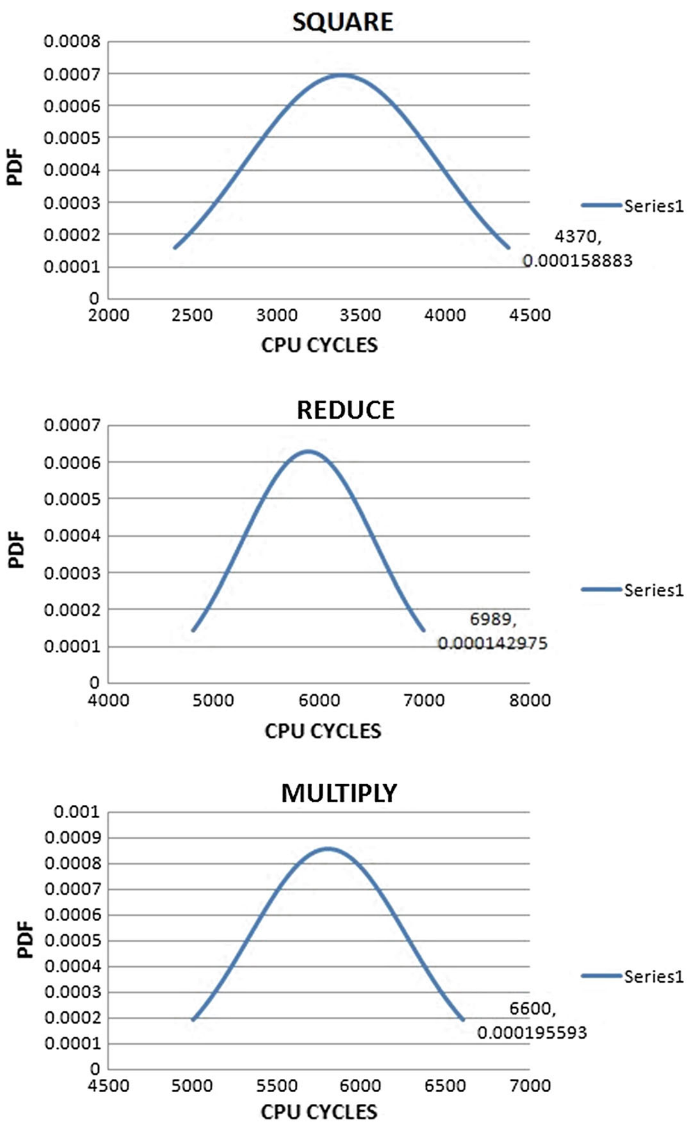

Fig. 6 Normal distribution curve for Square–Reduce–Multiply operation

Suppose attacker takes time t2 to access data X, time t1 to access data Y and time t2 to access data Z. Its mean X and Z data is not found in cache while data Y occurs in the cache. If data does not found in the cache, it is needed to load it from RAM to Cache which requires a significant amount of extra time. As functions, S and R are replaced by X and Y, it can be inferred that the operations sequences are SR i.e. bit 0.

Similarly if sequences SRMR are traced, means bit is 1.

In the formal Modelling, we consider a deterministic approach to find private key of victim VM.

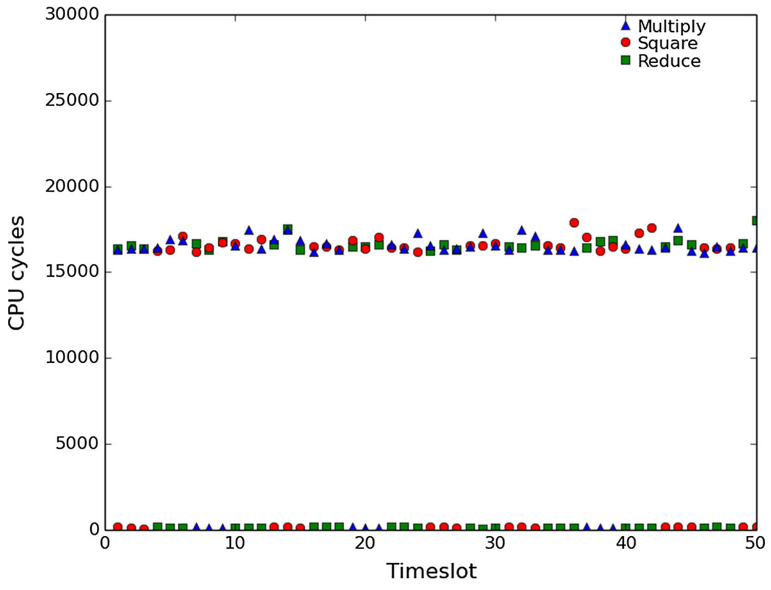

Fig. 7 Sequences of Square–Reduce and Multiply operation under attack

We store traces of hit and miss sequences of cache memory based on time t1 required for hit and t2 for miss. For this purpose, we require deterministic definition for running Attacker program and their computation.

Formal Definition of Program and Computation

Traces can be defined in 5 tuples system as mention below:

$$T = \{\Sigma, I_0, F, E, \delta\} \tag{2}$$

where Σ, set of states; I_0, set of initial states; F, set of final states; E, set of events; δ, transition relation defined as; $\delta \subseteq \Sigma x E x \Sigma$. *Note* Events are work as input for present state.

Present state is defined as instances of cache and memory at particular time.

$$\Sigma = \mathrm{M} \times \mathrm{C}$$

where M represent memory instance at particular time and C represent cache instance at particular time.

In our model, computation of key bit sequences can be find using states and inputs (events) as it simulate as deterministic finite automata, where next states always deterministic.

Here we represent state as s and Event/input as e. In computation of private key (P_{key}) is an alternate sequence of state and inputs.

$$s_0 e_0 s_1 e_1 \ldots e_n \text{ such that } s_i \varepsilon I_0 \text{ and } e_i \varepsilon E \text{ for all } i \varepsilon \{0, 1, 2 \ldots n-1, n\} \{s_i, e_i, s_{i+1}\} \varepsilon \delta$$

The set of all traces are used to collect semantics.

$$Collection(P_{key}) \subseteq \text{traces of computation}$$

where traces denotes all sequence of states and events (inputs).

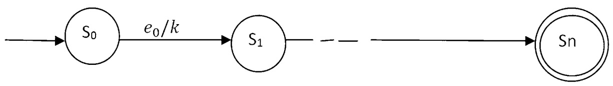

Fig. 8 DFA presentation of computation traces

We are terminating the computation of key, the trace process to collect the semantics as the fix-point of the next function/operator containing I.

$$Collection(P_{key}) = if\, P_i\, \varepsilon\, traces \bigcup next(s_i)$$

where s_i is trace or state.

$$next(s_i) = \left\{ t.s_n e_n s_{n+1} | t.s_n \varepsilon S \bigwedge (s_n, e_n, s_{n+1}) \varepsilon \delta \right\}$$

In our formal modelling computation is performed through no of states passes which can be represented by formal DFA definition:

$$Where \quad k \varepsilon \{0, 1\}$$

In the Fig. 8, we shows that traces start from initial states and on each state input is events which made changes in cache and memory. We get output on each transition of state, which is either 0 or 1.

So traces did collecting the value of k and computation process had been end when again reach to initial state.

8 Information Theoretic Security Guarantee of Proposed Prevention Approach

Information leakage may create vulnerability of confidentialities breaches that can be exploited by an attacker to perform a side-channel attack. In Cache based SCA, an attacker can use the side channel information like access times of different functions, collections of traces of hit and miss sequences of cryptographic operations. So confidentiality of secret information is one of the most important security issues. We maintain the confidentiality with our prevention scheme by random permutation of an address of S, M, and R, which generate wrong information regarding time, access and traces for an attacker. So information about the private key is not completely leaked, and system will maintain the confidentiality of private key.

We know that

$$\text{Initial uncertainty} = \text{information leaked} + \text{remaining uncertainty}$$

In our model, we try to minimize information leakage and maximize the remaining uncertainty. Here we consider deterministic probabilistic program P, that receives a high input (private key info) H, assumed to satisfy a publicly known a priori distribution and produce a low output (publicly encryption information about S, M, and R) L. P_s is a system program which perform encryption. Adversary/Attacker A, that run the program P_a to find encryption key or private key.

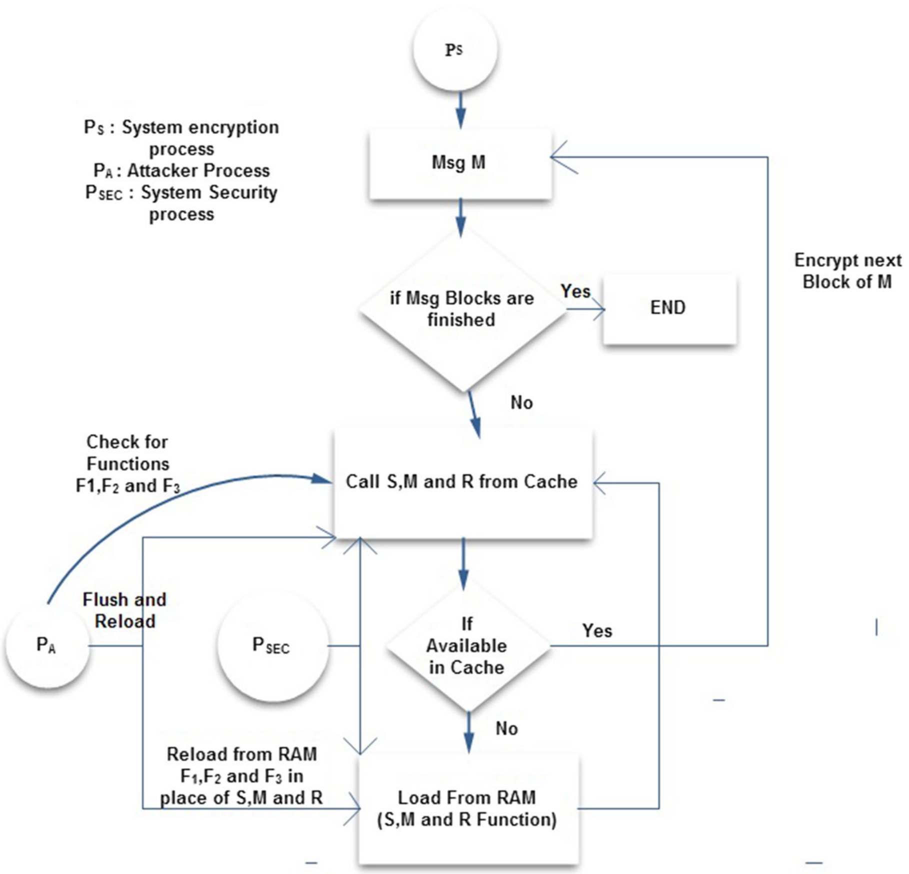
PS
PS : System encryption process
PA : Attacker Process
PSEC : System Security process
Msg M
if Msg Blocks are finished
Yes
END
Encrypt next Block of M
No
Check for Functions F1,F2 and F3
Call S,M and R from Cache
Flush and Reload
PA
PSEC
If Available in Cache
Yes
No
Reload from RAM F1,F2 and F3 in place of S,M and R
Load From RAM (S,M and R Function)

Fig. 9 Formalization process flow of proposed security model

Suppose private key length is $L_{key} = 256$.

After 64 bits, our system detects this attack (in worst case), as system detects attack, a program P_{sec} is activated. This program provides randomness by exchanging functions (S, M and R) with other system functions F1, F2, F3 without the knowledge of attacker/adversary A (Note that F1, F2, and F3 are the functions which are currently used by some other program, because it is loaded in cache memory). After 64 bit encryption time period, (approximate) again reset the S, M and R at its own place or until attacker will detect continuously by detection process (see Fig. 9).

P_{sec} is having two important features, the first one is address hiding of new generated address of S, M and R by EX-OR operation, which hide occurrence of S, M and R from attacker. And the second one is exchange of Address of S, M and R, increases the uncertainty of Information. In the next subsections we define the properties of P_{sec} in detail.

8.1 Address Hiding

As soon as we detect attack by detection process, we immediate shift our S, M and R functions to some other places of cache. We generate new address place with the help of random variable value that is perfectly masked because the new address value is statistically independent of the current address (sensitive data) value [].

$$A_{new} = A_{old} \bigoplus r.$$

where A_{new}, new address of S, M and R (separately); A_{old}, Old address of S, M and R (separately); r, random values.

Eldib et al. [27] say that any encoding is satisfiable (secure) if and only if all intermediate nodes of encoding are perfectly masked.

Note r is random variable, uniformly distributed over $(0, 1)^*$.

8.2 Hiding of Information About S, M and R Occurrence

Given program P_s, which may leak information from H (secret key) to L (occurrence of S, M and R), we want to define how much information P_s leaks.

In the literature, such definitions are usually based on information theoretic measures, such as Shannon entropy. First we briefly review some of these measures. Let X be a random variable whose set of possible values is X. The Shannon entropy H(X) is defined by

$$H(X) = \sum_{x \varepsilon X} P[X = x] log \frac{1}{P[X = x]} \tag{3}$$

The Shannon entropy can be viewed intuitively as the uncertainty about X; more precisely it can be understood as the expected number of bits required to transmit X optimally.

Given two (jointly distributed) random variables X and Y, the conditional entropy $H(X\|Y)$, intuitively the uncertainty about X given Y, is

$$H(X|Y) = \sum_{y \varepsilon Y} P[Y = y] H(X|Y = y) \tag{4}$$

where

$$H(X|Y = y) = \sum_{x \varepsilon X} P[X = x][Y = y] log \frac{1}{P[X = x][Y = y]} \tag{5}$$

Note we assume that log denotes logarithm with base 2.

If X is determine by Y Then

$$H(X|Y) = 0$$

The mutual information I(X; Y), intuitively the amount of information shared between X and Y, is

$$I(X; Y) = H(X) - H(X|Y).$$

Table 8 Entropy in best, average and worst cases

H(H)	I(H;L)	$H(H\|L)$	Comments
256	0	256	**Best Case** Attack detect immediately
256	64	192	**Average Case** Attack detect after 25% of information leakage
256	128	128	**Worst Case** Attack detect after 50% of information leakage

Mutual information turns out to go be symmetric:

$$I(X;Y) = I(Y;X).$$

The guessing entropy G(X) is the expected number of guesses required to guess X optimally; of course the optimal strategy is to guess the value of X in non-increasing order of probability.

If we assume that Xs probability are arranged in non-increasing order $P_1 \geq P_2 \geq \cdots \geq P_n$ then we have

$$G(X) = \sum_{i=1}^{n} {}^{i}P_i \tag{6}$$

Now we consider how these entropy concepts can be used to quantify information leakage.

Suppose

Initial uncertainty about H = H (H)

Remaining Uncertainty about H, conditional entropy = H $(H|L)$

Information leaked to L, the entropy is = H (L)

In SCA, we are unable to correct process P_S, where P_S is probabilistic. L is the positive entropy because information leaked from H to L.

We know that P_S is deterministic, and then L is determined by H.

So H$(L|H) = 0$, which implies that

$$I(H|L) = I(L;H) = H(L)H(L|H) = H(L).$$

In this scenario, the mutual information I (H; L) can be simplified to the entropy H (L).

So we need to reduce leakage information by masking of occurrences of S, M and R. In our prevention scheme we change the address space of S, M and R, as soon as detect SCA.

We assume secret key size is 256. Table 8 shows that in best case attack is detected immediately as $H(H|L) = 256$ i.e. with any information leakage, and in worst case attack is detected after 50% information leakage.

In consensus Definitions

In Guessing Entropy

$$\begin{aligned} G(H|L) &\geq 2^{H(H|L)-2} + 1 \\ &\geq 2^{128-2} + 1 \\ &\geq 2^{126} + 1 \end{aligned}$$

And actual expected numbers of guesses are:

$$(2_{128} + 1)/2$$

In Fano inequality

$$\begin{aligned} P_e &\geq \frac{H(H|L) - 1}{log(|H| - 1)} \\ &\geq \frac{128 - 1}{log(2^{256} - 1)} \end{aligned}$$

Since here the adversary (Attacker) has no knowledge of 128 of the bits of H, which implies that

$$P_e \geq \frac{2^{128} - 1}{2^{128}}$$

9 Conclusion and Future Work

In this paper, we have demonstrated an attack using cache-based side-channel analysis in virtualized environments. We have simulated the attack with two VMs (attacker and victim) as objects and considering the host machine architecture as a multi-core architecture. We have also proposed a solution to prevent this attack to happen with a considerably less performance degradation.

We have implemented an approach in which we first discover whether the attack is being performed on the victim or not. This is achieved by Hypothesis testing and confidence intervals. Now, if the victim is being attacked then it performs a permutation function to map the physical address of the operation to some other cache location than the one calculated by associative mapping of the cache. The permutation function is applied only when the victim finds out that it is being attacked, this ensures that the performance of the shared cache is degraded less in comparison to other solutions. We have used a simple permutation function for now but in future there is scope for enhancing complexity of the permutation function.

Compliance with Ethical Standards

Conflict of interest The authors declare that they have no conflict of interest.

References

1. http://www.usatoday.com/story/tech/2014/09/24/data-breach-companies-60/16106197/.
2. CLOUD SECURITY ALLIANCE The Treacherous 12—Cloud Computing Top Threats in 2016.
3. Osvik, D. A., Shamir, A., & Tromer, E. (2006). Cache attacks and countermeasures: The case of AES. In D. Pointcheval (Ed.), *Topics in cryptology CT-RSA 2006. CT-RSA 2006. Lecture notes in computer science* (Vol. 3860). Berlin: Springer.
4. Godfrey, M. M., & Zulkernine, M. (2014). Preventing cache-based side-channel attacks in a cloud environment. *IEEE Transactions on Cloud Computing*, *2*(4), 395–408.
5. Xu, Y., Bailey, M., Jahanian, F., Joshi, K., Hiltunen, M., & Schlichting, R. (2011). An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop (CCSW '11)* (pp. 29–40). New York, NY: ACM.

6. Kim, T., Peinado, M., & Mainar-Ruiz, G. (2012). STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX conference on Security symposium (Security'12)* (p. 11). Berkeley, CA: USENIX Association.
7. Raj, H., Nathuji, R., Singh, A., & England, P. (2009). Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM workshop on Cloud computing security (CCSW '09)* (pp. 77–84). New York, NY: ACM.
8. Page, D. (2003). Defending against cache-based side-channel attacks. *Information Security Technical Report*, *8*(1), 30–44. ISSN 1363-4127.
9. Han, Y., Chan, J., Alpcan, T., & Leckie, C. (2017). Using virtual machine allocation policies to defend against co-resident attacks in cloud computing. *IEEE Transactions on Dependable and Secure Computing*, *14*(1), 95–108.
10. Ristenpart, T., Tromer, E., Shacham, H., & Savage, S. (2009). Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on computer and communications security (CCS 09)* (pp. 199–212). New York, NY: ACM.
11. Zhang, Y., Juels, A., Reiter, M. K., & Ristenpart, T. (2012). Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on computer and communications security (CCS 12)* (pp. 305–316). New York, NY: ACM.
12. Lampson, B. W. (1973). A note on the confinement problem. *Communication of ACM*, *16*(10), 613–615.
13. Kong, J., Acicmez, O., Seifert, J.-P., & Zhou, H. (2008). Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 2nd ACM workshop on computer security architectures (CSAW 08)* (pp. 25–34). New York, NY: ACM.
14. Bernstein, D. J. (2005). Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.
15. Osvik, D. A., Shamir, A., & Tromer, E. (2005). Cache attacks and countermeasures: The case of AES. In *Topics in cryptology—CT-RSA 2006, the cryptographers track at the RSA conference 2006* (p. 120).
16. Yarom, Y., & Katrina, F. (2014). FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX conference on Security Symposium (SEC14)* (pp. 719–732). Berkeley, CA: USENIX Association.
17. Gruss, D., Spreitzer, R., & Mangard, S. (2015). Cache template attacks: Automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX security symposium* (pp. 897–912).
18. Gruss, D., Maurice, C., Wagner, K., & Mangard, S. (2016). Flush+Flush: A fast and stealthy cache attack. In J. Caballero, U. Zurutuza, & R. J. Rodrguez (Eds.), *Proceedings of the 13th international conference on detection of intrusions and malware, and vulnerability assessment—(DIMVA 2016)* (Vol. 9721). (pp. 279–299). New York, NY: Springer.
19. Wang, Z., & Lee, R. B. (2007). New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on computer architecture (ISCA 07)* (pp. 494–505). New York, NY: ACM.
20. Kong, J., Aciicmez, O., Seifert, J.-P., & Zhou, H. (2008). Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 2nd ACM workshop on computer security architectures (CSAW 08)* (pp. 25–34). New York, NY: ACM.
21. Zhang, Y., Juels, A., Oprea, A., & Reiter, M. K. (2011). HomeAlone: Coresidency detection in the cloud via side-channel analysis. In *2011 IEEE symposium on security and privacy* (pp. 313–328). Berkeley, CA.
22. Kim, T., Peinado, M., & Mainar-Ruiz, G. (2012). Stealthmem: System level protection against cache-based side channel attacks in the cloud. In *Security12* (pp. 11–15). Berkeley, CA: USENIX Association.
23. Zhou, Z., Reiter, M. K., & Zhang, Y. (2016). A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security (CCS '16)* (pp. 871–882). New York, NY: ACM.
24. Liu, F., et al. (2016). CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)* (pp. 406–418). Barcelona.
25. Liu, F., Yarom, Y., Ge, Q., Heiser, G., & Lee, R. B. (2015). Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy* (pp. 605–622). San Jose, CA.
26. Bosman, E., Razavi, K., Bos, H., & Giuffrida, C. (2016). Dedup est machina: Memory deduplication as an advanced exploitation vector. In *2016 IEEE symposium on security and privacy (SP)* (pp. 987–1004). San Jose, CA.
27. Eldib, H., Wang, C., & Schaumont, P. (2014). Formal verification of software countermeasures against side-channel attacks. *ACM Transactions on Software Engineering and Methodology, 24*(2), Article 11, 24 pages (2014).

Sandeep Saxena currently pursuing Ph.D. from NIT Durgapur, West Bengal and working as Assistant Professor in a reputed engineering institute Krishna Engineering College, Ghaziabad. He is completed his B.Tech. from U.P.T.U Lucknow and M.S. in Information Security from IIIT Allahabad. He has done 8 years of excellent work in Academic in various well known institutes. He is part of various National/International professional societies CSI, CRSI and IEEE etc. His areas of interest are Computer Network, Compiler Design, DBMS, Automata Theory, Data Structure, Information Security and Cyber Laws.

Goutam Sanyal is a member of the IEEE. He has received his B.E. and M.Tech. degree from National Institute of Technology (NIT), Durgapur, India. He has received Ph.D. (Engg.) from Jabalpur University, Kolkata, India, in the area of Robot Vision. He possesses an experience of more than 25 years in the field of teaching and research. He has published nearly 115 papers in International and National Journals/Conferences. 4 Ph.D. (Engg.) have already been awarded under his guidance. At present he is guiding six Ph.D. scholars in the field of steganography, Cellular Network, High Performance Computing and Computer Vision. He is presently working as a Professor in the department of Computer Science and Engineering and also holding the post of Dean (Faculty Welfare) at National Institute of Technology, Durgapur, India.

Shashank Srivastava currently working as Assistant Professor at Motilal Nehru National Institute of Technology, Allahabad. He recently submitted his Ph.D. on the topic security of mobile Agent. Shashank Srivastava received his B.Tech. in Computer Science and Engineering from U.P. Technical University, Lucknow, India in 2006. In 2009, He did his Master Degree (M.S.) from Indian Institute of Information Technology, Allahabad, India.

Ruhul Amin currently working as Lecturer at Thapar University, Patiala. Pursuing Ph.D. from ISM Dhanbad. Published multiple research papers in reputed international journal.