MIKELANGELO

D3.4

sKVM Security Concepts – First Version

Workpackage	3	Hypervisor Implementation	
Author(s)	Gabriel Scalosub		BGU
	Niv Gilboa		BGU
Reviewer	Holm Rauchfuss		HUAWEI
Reviewer	Joel Nider		IBM
Dissemination Level	Public		

Date	Author	Comments	Version	Status
2015-12-07	Niv Gilboa, Gabriel Scalosub	Document ready for review	V0.0	Review
2015-12-24	Niv Gilboa, Gabriel Scalosub	Applied reviewers comments (Round 1)	V0.1	Review
2015-12-28	Niv Gilboa, Gabriel Scalosub	Finalized document	V1.0	Final

Executive Summary

This deliverable describes the work of the BGU team during the first year of the MIKELANGELO project. The goal of our team is to provide security functionality for the sKVM hypervisor, in particular against cache-based side-channel attacks. This class of attacks is evolving, diversifying and is particularly effective in cloud environments.

The BGU team made two main contributions to MIKELANGELO over the past year. The first contribution is a software testing tool that implements a cache-based side-channel attack against particular targets running over the traditional KVM hypervisor. The target of our tool is a Virtual Machine (VM) that runs the RSA algorithm[1], which is widely used for key exchange and digital signatures in such popular applications as web servers, network management tools and virtual private networks.

Assuming that the target VM and a VM running the tool are co-located on the same physical host, the tool extracts full RSA private keys within a matter of minutes. Since RSA keys are typically valid for months or years and are used to identify important networking components the implemented attack is attractive to a wide range of malicious actors.

The second contribution of the BGU team is a design for a software module to secure virtualized systems against cache side-channel attacks. We call the module, which is projected to be a part of sKVM, the Side-Channel Attack Monitoring and Mitigation module (SCAM). We intend to use the testing tool to validate the security module in later stages of the project.

SCAM will collect data on the pattern of cache access of virtual machines running over sKVM and will use this data to profile VMs and evaluate the likelihood that a monitored VM is executing a cache side-channel attack. In addition, SCAM will use several strategies to reduce the effectiveness of such attacks and if possible to fully prevent them.

SCAM is a software module and as such will run on widely available systems without requiring specialized hardware. A second advantage is that it runs at the hypervisor level and does not require an application writer to modify the application specifically to resist cache side-channel attacks.

[1] Attacking other targets is possible, but would require more effort than we thought would be advisable for a testing tool.

Acknowledgement

The work described in this document has been conducted within the Research & Innovation action MIKELANGELO (project no. 645402), started in January 2015, and co-funded by the European Commission under the Information and Communication Technologies (ICT) theme of the H2020 framework programme (H2020-ICT-07-2014: Advanced Cloud Infrastructures and Services)

Table of Contents

Table of Figures

Table of Tables

1 Introduction

This document presents the sKVM Security Concepts, and their implementation in the first year of work on the MIKELANGELO project. The security concepts within MIKELANGELO revolve around enhancing security on the hypervisor level (as opposed to lower level hardware, or higher level applications) against cache side-channel attacks. These hardening mechanisms are to be implemented as part of sKVM -- super KVM -- the improved version of KVM which is one of the main contributions of this project.

The main focus in the first year was set on the overall design of the architecture of the sKVM security module, as well as on the implementation of an attack module which exploits cache side-channels in order to obtain sensitive information of a co-located target Virtual Machine (VM).

The overall structure of this document is as follows:

1. Introduction to cache side-channel attacks in virtualized environments, including a description of a prime-and-probe attack aimed at extracting the target's private RSA key by targeting the Last Level Cache (LLC), as well as an extensive survey of previous work. These details are presented in Sections 2 and 3.

2. A detailed description of our implementation of such an attack, including a discussion of security performance metrics and their variability depending on various tunable parameters. These details are presented in Section 4.

3. A description of the architecture of the sKVM security module -- SCAM [2] (Side-Channel Attacks Monitoring/Mitigation). These details are presented in Sections 5.

4. Potential mitigation techniques to be explored in later phases of the project within SCAM. These details are presented in Section 5.3.

5. A comparison of the SCAM design with alternative strategies including modified hardware and application-specific methods, presented in Section 5.6.

6. Key takeaways and conclusions are presented in Sections 6 and 7.

We note that the decision to focus on an attack on the LLC is due to its recent popularity in the literature, and only serves to demonstrate a working attack on standard platforms, which will serve as a testing tool to the mitigation techniques developed later in the project.

[2] Yes, SCAM. This is not a typo.

2 Security in Virtualized Environments: TLS and RSA

Securing user transactions with web servers is vitally important for both service providers and users. Services such as e-commerce, healthcare and finance regularly rely on HTTPS, the secure web protocol, to hide credit-card numbers from eavesdroppers and authenticate the web servers, thus ensuring that users are communicating with the intended service providers.

HTTPS is a layering of HTTP, the web protocol, over a security protocol called Transport Layer Security (TLS) [RFC5246] (or over its predecessor SSL). While the TLS protocol is complex and relies on several mechanisms for security, its most critical component is the server's RSA private key [21]. If an attacker obtains that key then it can masquerade as the web server, read the traffic between the user and the server and in general cause mischief. Even worse, the service is compromised for a long time if an RSA key leaks since such keys are typically valid for a year or longer.

The RSA key, which is such a tempting target for attackers, is used in a deceptively simple way in TLS. RSA actually uses two keys. A public key which is made up of a large integer *n*, which is the product of two secret primes *p* and *q* and a public exponent *e*. The server's public key *(n,e)* can be trivially obtained by any user, legitimate or otherwise, who browses to the web server. The RSA private key, *d*, must be kept secret by the server. RSA keys have the property that for an appropriate message *m* if $c=m^e \ mod \ n$ then $m=c^d \ mod \ n$. In TLS the server authenticates its identity to the client by proving that it has the private RSA key. One method to prove knowledge of *d* is for the client to choose a message m, encrypt it with the public key by $c=m^e \ mod \ n$ and send it to the server who can then compute $m=c^d \ mod \ n$ and use *m* as a shared key with the client. Although RSA seems simple, if the public modulus *n* and the private key *d* are large enough then there are no known feasible attacks on this system using just the public key and *c*. Current best practice is for the *n* and *d* to be of size at least 2048 bits (which is more than 600 decimal digits).

This is where cache-based side channel attacks come in. The only time that the server uses the all-important RSA private key is in the computation $m=c^d \ mod \ n$. Computing *m* in a straightforward way by multiplying c in a loop *d* times is out of the question. The exponent *d* is so large that the hardware computing the operation will rust away long before it returns the result. There is a much faster and more popular algorithm called *square-and-multiply* that uses the binary representation of *d*, $d=d_m,\ldots,d_0$, where d_0 is the least significant bit (lsb). The algorithm uses the equality:

$$c^d \ mod \ n = c^{\sum_{i=0}^{m} d_i \, 2^i} \ mod \ n = \prod_{i=0}^{m} c^{d_i \, 2^i} \ mod \ n = \left(\left(\left(c^{d_m}\right)^2 c^{d_{m-1}}\right)^2 \cdot \ldots \cdot c^{d_0}\right)^2 \ mod \ n.$$

A typical implementation of the algorithm has the following code snippet (in C):

```
void product (large_int *r, large_int x, large_int y, large_int n); // compute: r=xy mod n

exp (large_int *r, large_int c, char *d, large_int n) {            // compute: r=c^d mod n
    (1)       z=1;
    (2)       for (i=m,i>=0,i--) {          // iterate over all bits in the private key,
                                               starting from msb
    (3)          product (&z, z, z, n);     // square: z=z^2 mod n
    (4)          if (d[i]==1)
    (5)             product (&z, z, c, n);  // multiply: z=zc mod n
    (6)       }
```

This implementation is efficient and popular in cryptographic libraries. However, it is susceptible to a cache-based side-channel attack. The next section introduces side-channel attack in general and describes in greater detail how such attacks can be implemented using a shared cache.

3 Security in Virtualized Environments: Cache Side-channel Attacks

A side-channel attack on an information processing system is an algorithm that attempts to extract information not only from the system's input and output, but also from details of its implementation. Typical side channel attacks measure such properties of the system as its running time, power trace and the electromagnetic field it generates. These properties serve as a medium which can be observed externally. When these features of the system change as a function of the input data, an attacker can infer information on that data, thereby creating a side channel.

Cache-based side-channel attacks are a specific type of such attacks targeting software processes executed on multiprocessing and multi-tenant systems. Such systems, which include many modern operating systems and cloud environments, achieve significant savings by sharing expensive hardware resources among several software processes. Low-level software, either the operating system or a hypervisor is responsible for logical separation between different processes or virtual machines, ensuring for example that each process cannot access the main-memory space attached to a different process.

However, it is difficult to enforce such separation in some hardware resources, specifically in cache memory. The purpose of cache memory is to improve the time required for memory access and therefore its whole implementation is in hardware and cannot be readily configured by low-level software using similar methods to the logical separation of main memory.

The shared nature of cache memory and the speedup it offers to memory accesses form the basis for cache-based side-channel attacks. If an attacking process (or virtual machine) runs on the same physical machine as a target process then the attacker can obtain information on the target by the pattern of cache accesses that the target performs.

A basic building block of such attacks is the prime-and-probe technique [18]. Prime-and-probe begins when the attacker "primes" the cache by writing instructions or data to the memory space that the attacker controls. Processor hardware copies this information to the cache. The attacker process then waits for the target to execute. Since the cache is much smaller than the main memory of a process, when the target process runs, its own data and instructions replace some of the information that the attacker stored. The attacker completes the process by "probing", that is reading (or writing) the same data and instructions that were used in the priming stage, while measuring the time required to perform each access. If access time is relatively long then the attacker's data is no longer in the cache, which implies

that the target has overwritten the data in that location and the converse if the access time is short.

The applicability of the prime-and-probe attack was shown in a series of works in which the attacker obtains a cryptographic key from a target process. In [18] and [24] an attacking process exploits standard implementations of the Advance Encryption Standard (AES) encryption algorithm that use lookup tables for part of the encryption or decryption process. In AES, the locations in the lookup table that the algorithm accesses depend in a straightforward way on the bits of the secret key and furthermore it is possible to infer the secret key given these locations. Since the prime-and-probe technique enables extraction of the accessed locations in a lookup table, standard implementations of AES are vulnerable to this attack.

Prime-and-probe enables similar attacks on public-key algorithms, e.g. RSA. A typical implementation of the core exponentiation procedure of the RSA algorithm was described in Section 2.

It is clear from the code snippet that a specific set of instructions, represented by line (5), is executed in the i-th iteration if and only if the i-th bit of the secret key is 1. It follows that a prime-and-probe attack that probes the cache immediately after every iteration of the for loop can extract the whole key. Such attacks on public key algorithms under various assumptions have been shown in e.g. [30], [29], [14] and [11].

Initial cache-based side-channel attacks were implemented in non-virtualized environments. Indeed, for several years common wisdom dictated that the extra noise and memory virtualization levels imposed by hypervisors in virtualized environments would make such attacks impractical. In what was a surprise at the time [20] showed that cache-based side-channel attacks can be reliably performed in a live cloud environment (Amazon EC2). However, the attacks in that initial paper were not sufficiently refined to extract a high-value secret such as a cryptographic key.

The obstacles facing a prime-and-probe attack in a cloud environment are formidable. The micro-architecture of modern x86 processors, which are ubiquitous in cloud environments, includes multiple cores and two or three levels of cache memory such that the last level (largest and slowest of the caches) is shared among all cores, while each core has separate hardware components implementing the other cache levels. If the hypervisor assigns the attacker VM and the target VM to the same core then the attacker has access to all cache levels, but does not operate during the execution of the target. On the other hand, if the hypervisor assigns the two VMs to separate cores then the attacker has access only to the last level cache, which makes it difficult to locate the critical data of the target, since this cache

stores at least several megabytes of memory. Another obstacle is that attacks such as the extraction of AES or RSA keys require the attacker to be synchronized with the target's operation such that its probes occur at an appropriate time, e.g. after the completion of one iteration of the main loop of the exponentiation pseudo-code. In addition, the attacker must contend with significant noise in its cache probes caused by the operation of software components such as the hypervisor or other VMs besides the target running on the same physical machine.

Recent attacks have developed a variety of tools to overcome these challenges. [30] introduces two important tools: Inter Process Interrupts (IPI) to synchronize the attacker with the target process and machine-learning methods such as support-vector machines and hidden Markov models to reduce the overall noise compared to the signal induced by the key that the attack extracts. IPI is a way for different processes, or different but cooperating VMs, to communicate asynchronously. Since some hypervisors prioritize IPI over regular execution, the attacker can time the interrupt to return execution to its probing VM at the expense of the target at a time of its choosing. This attack explicitly relies on the attacker and the target sharing the same core.

Since cloud provider's policies often give preference to separating the virtual machines of different users among different cores, the attack of [30] is not always applicable. Attacks on the last level cache such by Ristenpart et al. [20] and Wu et al. [28] were not sufficient to extract cryptographic keys. The objective of Yarom and Falkner [29] was to specifically address this challenge by devising an attack that extracts keys from the last level cache. Since the last level cache is relatively large, the attacker must first reliably locate the critical cache sets. The authors assume a setting of shared libraries, or de-duplication, between the attacker and target virtual machines. De-duplication in this context means that if two VMs share the same read-only memory pages, which are typically code libraries, then these pages are loaded only once to memory and both VMs have access to them. In this way, if the attacker loads the same library as the target, e.g. a library that implements TLS, then it can predict the location of critical code in the last level cache with great accuracy, since the physical address of the code is identical for both machines.

Virtualization vendors explicitly advise against allowing de-duplication [26] and cloud providers generally follow this advice. Liu et al. [14] improve previous attacks on the last level cache by dropping the requirement for de-duplication. As an alternative to knowing the exact addresses for critical instructions and codes, the paper proposes to have the attacker scan each set in the last-level cache repeatedly and observe the temporal pattern of accesses that the target makes to the cache. The rationale is that if a cache set stores instructions or data for functionality such as a modular exponentiation algorithm then the pattern of memory accesses becomes so distinct that the attacker can identify it with a sufficient number of

probes. This technique enables the attacker to determine a small number of cache sets to be monitored instead of trying to monitor the whole cache, which is infeasible to achieve with sufficient resolution due to its size. The only remaining challenge is that the attacker only knows the virtual addresses of the memory locations that are mapped to each cache set, while the actual mapping uses physical addresses. Therefore, when the association of physical to virtual pages changes due to paging (swapping virtual pages between main memory and the disk) the attacker may be fooled. This problem is resolved in the paper by making (the very reasonable) assumption that the hypervisor arranges the virtual memory of the attacker's VM in *large pages* of at least 2 Mbytes. Using large pages is attractive in a cloud environment due to performance benefits. In such large pages, since the mapping between a physical address and a cache set is determined only by the lower order bits of the address, even when the virtual to physical association of a page is changed, the mapping to a cache set does not change.

4 Attack Implementation

Our implementation of the cache side-channel attack on the last level cache follows the lines suggested in [14] and [11]. The implementation of the attacker is composed of the following components:

(1) **Generating a cache map**: creating a complete cache-map consisting of the set of addresses which fill the different associative sets of the cache, allowing the attacker to completely fill the cache with his code, and doing so in a target cache line oriented manner.

(2) **Recognize target cache-access pattern**: recognizing the sets of the cache used by the target process for decryption operations, using predefined knowledge of the access patterns of the decryption modules used by such a process.

(3) **Collect cache-access trace**: prime-and-probe the specific cache lines used by the target in order to obtain a trace of cache hit-miss access, which in turn reflects the private-key-dependent exponentiation activity of the target

(4) **Collect multiple trace samples**: repeatedly collect samples of cache-access performed by the target

(5) **Analyze samples**: Use the various samples to identify the private key used by the target.

In what follows, we provide a detailed description of each of these components, as implemented by us. We assume that target's RSA private key length is *L* bits, where in our implementation we use *L=4096* bits as the default value. We assume the system employs no deduplication (i.e., each VM maintains its own copy of the code it requires within its allocated memory space). We configure the attacker to use Huge Pages, but make no such assumption on the target which may, or may not use Huge Pages. Our attack is mostly implemented in C, with some parts implemented in Assembly in order to ensure efficiency and accuracy at high sampling rates. We denote by *N* the number of cache lines in each set (determined by the architecture of the associative cache structure of the system), where in our system implementation we have *N=12* lines per set. We assume the attacker knows the underlying system parameters.

We focus our attack on the GMP multiple precision arithmetic library [7] which performs the exponentiation procedure within the RSA algorithm. GMP is commonly known as the fastest library for performing such operations (by a significant margin, see [2]), and provides the cryptographic backend to full TLS implementations, such as GnuTLS [8].

The main tool employed by our attack, in its various phases, is time measurements of memory read access, to determine whether a cache hit or a cache miss have occurred. In our testbed, an LLC cache miss takes ~75-200 ns, whereas a cache hit takes ~15 ns. This basic difference is the driving force of the attack, and is essentially the only access tool required for performing it (in the appropriate setup, and with the right choice of parameters).

We provide here a list of the various tunable parameters used in our implementation, as well as other notation used throughout our description, to serve as a legend for aiding the reader to better understand the roles and function of these parameters and notation in later sections. Tunable parameters are denoted by lower-case letters, target- and system-dependent parameters are denoted by upper-case letters, and other notation uses upper-case Greek letters.

Table 1: List of tunable parameters in the attack implementation, and notation used in its description

Phase	Parameter	Description	Default value
-	*L*	target private key length	4096 bits
System settings	*N*	number of cache lines per set	12 lines/set
System settings	*Q*	number of bits required to identify line in cache	6 bits (64B lines)
System settings	*R*	number of bits required to identify the set number	11 bits (2048 sets)
Prime-and-probe	*k*	time between subsequent access attempts to a cache set	2 µs
Creating baseline cache set behavior (target idle)	*m*	Number of samples extracted from a cache set	1000 samples/set
Identifying cache sets used by the target (target active)	*q*	Increase in cache miss indicating target is active in the set (compared to baseline)	70%
Identifying cache sets used by the	*n*	number of samples extracted from a cache set	3000 samples/set

target (target active)			
Identifying cache sets used by the target's exponentiation	*t*	number of highest-scoring sets considered	3 sets
Target exponentiation activity	*W*	number of bits used in each exponentiation operation	1 bit
Target exponentiation activity	*D*	constant time for performing an exponentiation operation	16 µs
Sampling cache lines	*D/k*	number of samples per exponentiation operation	8 samples/operation
Statistical analysis of candidate keys	*s*	number of candidate keys sampled	50 candidate keys
Statistical analysis of candidate keys	*Σ*	collection of candidate keys	-
Processing candidate keys	*p*	number of missing bits required for pruning a candidate key	200 bits
Extracting the key from collection of candidate keys	*u*	number of bits considered in a window for a majority decision	16 bits
Extracting the key from collection of candidate keys	*f*	number of bits to shift left/right to fix a bad candidate key	3 bits

4.1 Generating a cache-map

Figure 1 provides the schematic mapping of memory locations to cache lines in various sets.

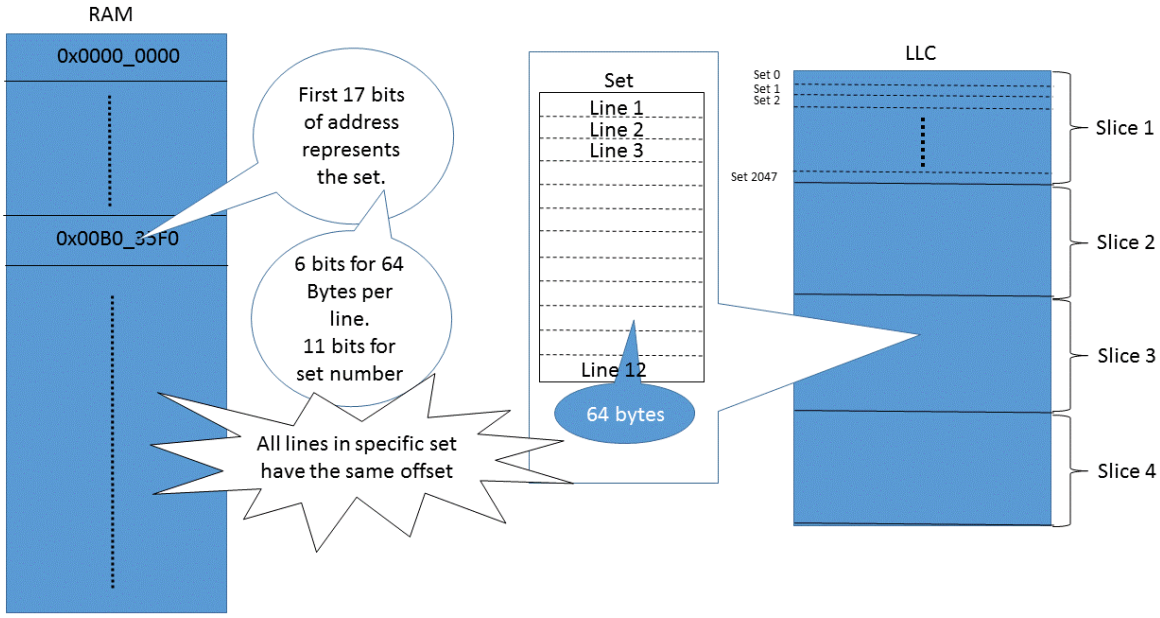


Figure 1: Static mapping between main memory and sets. Eviction rule for lines within each set is LRU.

To create the cache map in an efficient way, we use the fact that Intel processors use the first bits of an address to determine the associative set for that address. In our implementation the set is determined by the first 17 bits, consisting of Q=6 bits for the 64B per line, and R=11 bits for determining the set number, where Q and R are system-dependent parameters. We note, however, that each core in the system is associated with its own slice, and therefore an address may be mapped to a set in any one of the slices. The exact mapping of addresses into slices is determined by a more involved hash function that uses the entire address. The mapping of the target addresses is not observable by the attacker process, and we therefore do not assume we have access to slice-mapping of addresses. In order to fully understand this mapping, we perform a scan of the various addresses which share a common offset, and test whether they belong to the same associative set. This scan is performed for each offset until we fill the set of each slice.

4.1.1 Looking for potential set lines

The act of testing a collection of cache lines to check if they correspond to the same set is done in Assembly in order to avoid any unnecessary accesses to the stack, or affect any other resource that might in turn bear effect on the cache.

The procedure for constructing the mapping scans all possible offsets, and for each offset performs 2 phases. The first phase is devoted to identifying sufficiently many cache line collisions for the current offset. The second phase focuses on deducing the mapping by clustering cache lines collisions the correspond to the current offset. For every offset we repeat the process for each slice (since every offset may be associated with a set on a different slice).

We repeat the procedure for a constant number of times in order to verify the mappings we deduce for sets by each test are consistent. Furthermore, to avoid the effect of the procedure code interfering with the results, we duplicate the function code explicitly so that in each run the code itself resides in a different location in the memory.[3] This helps reduce noise, and make sure that in at least one copy of the code the code itself does not interfere with the measurements of the set.

On average, we are able to fully identify 93% of the sets; we identify on average 7646 sets, out of the overall 8192 sets (4 slices, 2048 sets per slice).

Phase 1: Finding a collision

Given the current offset being examined, we scan all yet unidentified lines that share this offset (initially this set contains all lines that share the offset), and access them in sequence. As long as no cache miss occurs, we deduce that we have not yet filled out a set. Once we have scanned sufficiently many such lines, a cache miss is bound to occur. At such a point we have a collection of lines that share the current offset, and some subset of this collection must be mapped to the same set on the same slice (since overall, it has caused a cache miss), in which case we move to phase 2.

[3] This is done using the "__always_inline" C function attribute, and is performed 10-15 times each time a set is tested/probed.

Phase 2: Deducing the mapping

Given a (potentially large) collection of lines that cause a cache miss, we wish to identify the *N* cache lines that correspond to a single set of a specific slice. We recall that in our implementation we have *N=12* lines/set.

If the collection contains exactly *N* lines, then we consider this set as identified, and associate these lines with this set. Otherwise, we must prune the collection to find a collection of exactly *N* lines corresponding to the same set. We employ the "leave-one-out" paradigm in testing the collection. For every line in the collection, we test all *other* lines in sequence, and check whether this sequence results in a cache miss. If a cache miss occurs, we deduce that the line left out does not correspond to the set. If no cache miss occurs, we deduce that the line left out does indeed correspond to the set. We repeat this procedure which results in a collection of lines that correspond to the same set. We note that the soundness of this method follows from the minimality of the collection, i.e., the fact the initial collection is determined by the *first* cache miss identified in Phase 1. We test the resulting (small) set of lines by scanning them in sequence, and verify this indeed causes a cache miss. In case of an unsuccessful scan (which may occur due to noisy samples, and other events that are not controlled by the process, such as OS cache lines access), we disregard the set, and move on to performing Phase 1 on another offset. This usually occurs when the resulting set is too small. If the resulting set contains more than *N* lines, we repeat Phase 2 for a constant number of times (in our implementations we repeat this 5 times). If in all these repetitions we fail to identify the set, we move on to performing Phase 1 on another offset.

4.2 Finding the decryption-relevant sets

We assume the attacker can run as the sole VM on the machine in the initialization phase, where it constructs the cache-map. Furthermore, we assume that once the cache-map is determined, the target constantly performs decryption, which is observable by the attacker via the cache side-channel. During this second phase the attacker focuses on identifying the sets in which the target is active, via the prime-and-probe mechanism (described in the sequel).

We note that these assumptions make the attacker stronger than one would expect in common settings, where one would probably encounter significantly higher levels of noise. Therefore, any mitigation techniques that are able to fend off such powerful attackers are expected to also deal with weaker attackers that work in much noisier environments.

We expect the attack to work also in settings where these assumptions do not hold, despite introducing increased noise into the attacker's measurements. However, once the mapping is established, the attack can proceed as described in the following sections.

4.3 Prime and Probe

The prime-and-probe mechanism tries to identify the target's activity in the collection of monitored sets short-listed during the various sampling phases, and ensure that all the lines of each such set are accessed (and hence filled) by the attacker so that any access by the target would result in a cache miss in the subsequent access made by attacker. It consists of filling the cache in all monitored sets by touching all lines in these sets, and then repeatedly testing all the lines of the set for a cache miss every *k* µs. In our experiments we usually set *k=2* µs, and thus test for cache misses in each line every 2 µs. However, this is a tunable parameter which captures the persistence of the attacker. We note that the precision of the attacker's measurements is inversely proportional to *k*.

4.4 Finding the relevant sets

The procedure for finding the collection of sets to be monitored is done by sampling each set in the cache *m* times, where *m* is a tunable parameter. In our experiments we usually set *m=1000*. In each such sample we fill all the lines of this set, and maintain the average cache hit/miss ratio for this set. This initial scan is performed when the target is inactive, and provides us with the baseline performance of each set in the absence of a target.

Once the baseline behavior is established, we continuously perform this scan over all sets, awaiting the activity of the target. Once the target begins its activity, the cache hit/miss behavior in the sets used by the target starts differing from the baseline behavior, which provides the attacker with an indication of which sets are used by the target. A set is said to be suspicious if its cache miss percentage in the scan increases by more than a factor *q*, where in our implementation we take *q=70%*. These sets are therefore determined as the monitored sets. At this point the attacker begins scanning these sets for patterns which may reveal key-relevant information.

4.5 The search for patterns

The procedure for determining the monitored sets provides a collection of sets whose activity profile differs significantly from that of the baseline activity. In the current attack, we target the exponentiation algorithm used in decryption of RSA. We assume the decrypting process tests a window of *W* bits at a time (*W* being a small constant, usually between 1 and 6), and performs exponentiation based on those *W* bits. For concreteness, we assume the target's decryption module uses *W=1*. This means that the decryption process tests one bit at a time in the exponent; if the bit is 0, then only squaring takes place, whereas if the bit is 1 then both squaring and multiplication takes place (see section 2 for an overview of the RSA decryption process).

We assume the target implementation of its decryption process is not naive, and has measures to ensure that each bit operation takes the same amount of time, regardless of the bit value. We denote this constant operation duration by *D* µs. In our implementation we have *D=16*. This eliminates the possibility of performing much simpler timing-based attacks.

Our goal is to find sets for which the cache access pattern indicates that these sets are used to perform the exponentiation procedure in RSA decryption. To this end, we perform some preprocessing in order to identify exponentiation activity in a cache set.

We focus on identifying cache lines that differentiate between handling a 0-bit, and those used in handling a 1-bit. The main difference is hence the multiplication operations, and these patterns can be distinguished by monitoring cache hit-miss in the monitored set.

Since any operation performed by the target takes *D* µs, and since we make a sample once every *k* µs, we obtain a total of *D/k* samples per operation. In our implementation, for the default values of *D* and *k* depicted above, we get *D/k=16/2=8* samples per operation. Each sample is represented by a bit such that the bit value is 1 if-and-only-if the attacker encountered a cache miss for the examined set, which is interpreted as indicating that the multiplication operation was executed by the target, thus providing evidence supporting the operation corresponding to a 1-bit in the key. It should be noted that it is not mandatory for a cache miss to be due to a multiplication operation by the target (since there might be other processes, e.g., OS-processes that may be accessing the cache alongside the target). However, our pattern search provides significant statistical evidence for this to be the case. The characteristic patterns encountered in our implementation for the 8-sample sequences are 11110000 for a 0-bit operation, and 11111111 for a 1-bit operation. Hence, a cache set for which 8-sample sequences are in most cases drawn from the above two 8-sample sequences are likely to be the cache set used by the exponentiation algorithm. We note that identifying the above patterns requires running extensive statistical analysis of 8-sample sequences, and they are sensitive to the variable tunable parameters (most notably *D* and k).

We further note that the above procedure for identifying patterns can be done offline, as long as the setup on which the search for patterns is performed is the same as the one in the actual virtualized environment. This means that the attacker may perform this search before ever accessing the virtualized environment.

4.6 Finding the best set

At the current point, the collection of sets identified by the increase in cache miss is conjectured to be associated with the target. However, we still need to identify which of these sets is associated with the *exponentiation* activity of the attacker.

In order to find the correct set corresponding to the exponentiation activity of the target, we sample each potential set for *n* times, where in our implementation we take *n=3000*. The reason for this repetition is the fact that our sampling procedure can (and in most cases is) noisy, and thus one cannot guarantee, or rely on, the fact that samples would exactly correspond to the two representative 8-sample sequences. The above scanning procedure results in every potential set getting a score representing the likelihood of the set being the one used by the target when performing exponentiation. The score depends on how many of the two 8-sample sequences were observed in the set, and more generally how close the *n* samples are to a typical sequence of a set handling the exponentiation. In particular, since the pattern corresponding to a 1-bit (11111111) is harder to discern, we mostly focus on identifying patterns corresponding to a 0-bit (11110000). The sets obtaining the highest such score are then considered as the sets to be monitored for obtaining the private key via the cache side-channel.

We note that it is useful to consider the *t* sets that obtain the highest scores, and not just the single highest-score set. In our implementation we take *t=3*, and consider each of these (small) number of sets as the set to be monitored by attempting to extract the key from each of them, as described in the sequel.

4.7 Extracting the key

4.7.1 Preparing the key samples

The procedure for sampling bits from the private key is based on the characteristic patterns identified previously. The attacker constantly scans the activity in the monitored set, using the prime-and-probe mechanism, awaiting sample sequences associated with exponentiation operations on either a 1-bit, or a 2-bit. For example, as stated previously, for a sampling rate resulting in an 8-samples sequence per operation, these patterns are 11110000 and 11111111 for the exponentiation associated with a 0-bit, and a 1-bit, of the private key, respectively.

Once the attacker observes the beginning of the exponentiation performed by the target, it registers the patterns encountered, and associates each such pattern with its corresponding bit (either 0 or 1). This process continues until no further patterns are encountered, resulting in the termination of the sequence, which serves as a candidate private key.

Clearly, due to noise, a candidate private key deduced in this manner may not be the correct private key used by the target. The attacker therefore generates *s* such candidate keys, where *s* is a tunable parameter. In our implementation we use *s=50* by default. We note that each

one of these candidate keys is obtained independently by a separate scan of the monitored set. We denote this set by Σ.

In the generation of each candidate key, the sampling may err in one (or more) of the following manners:

1. Swapped bits, where a true 1-bit was mistakenly identified as a 0-bit, or the other way round,
2. Missing bits, most probably due to delays on the attacker side that are artifacts of potential context-switches, or high cache miss rates for the attacker code, and
3. Extra bits, most probably due to delays on the target side. These occurrences tend to be relatively rare.

In order to overcome these sampling errors, one has to process and analyze the candidate keys, in order to extract the target's true private key. Errors where extra bits appear in the candidate keys are relatively rare, and they usually do not introduce significant pollution into the set of candidate keys. Swapped bits may introduce noise, but they are usually not co-located in the different candidate keys, so statistical methods applied later on suffice to cope with such errors. The hardest errors to overcome correspond to missing bits. In our observations we have identified the following properties of the candidate keys, with these errors:

1. Candidate keys tend to have relatively few missing bits. For example, for a 4096-bit private key, the number of missing bits is usually below 200.
2. The missing bits are distributed almost uniformly throughout the candidate key.
3. In cases where true keys have various 0-bits as the most significant bits, it is common for these bits to be missing in the candidate keys.

The first effect missing bits errors have on candidate keys is the fact that such keys are shorter than the true key. Since these missing bits can be arbitrarily located throughout the candidate key, we first prune our collection of candidate keys and remove all keys which are significantly shorter than the target's true key length L. More specifically, we prune any candidate key that is missing more than p bits (compared to the true key length), and thus remove all such candidate keys from Σ. In our implementation we usually set $p = 200$, and therefore disregard any candidate key whose length is less than $4096 - p = 3896$ (for $L=4096$-bit private keys).

The second effect missing bits errors have on candidate keys is the fact that if missing bits occur before we start accumulating bits of a candidate key (corresponding to problem (3) above), then in the statistical analysis performed in the sequel, targeted at identifying the true key, we encounter a synchronization problem, where different candidate keys are misaligned. Our study shows that missing bits tend to be 0-bits, and that identifying the first 1-bit still results in significant misalignment. We therefore focus our attention on identifying the *second* 1-bit in the candidate key, and provide a sample-trace of the entire key from this identified 1-bit onwards, until exponentiation terminates, which is identified by obtaining a sequence of consecutive samples that do not correspond to any of our two prescribed patterns.

We then turn to align all sample-traces of the candidate keys, starting from this second 1-bit identified, in order to obtain the target's private key. This key will require further padding by a sequence of type 0...010...0, in order to obtain a valid-length key. The length of the second sequence of 0-bits (residing between the first 1-bit of the key and the second 1-bit of the key) is taken as the average such number over all candidate keys. The length of the first sequence of 0-bits is then set so as to obtain a valid-length key.

4.7.2 The key extraction algorithm

Once a collection of candidate keys is obtained, all (presumably) aligned to the second 1-bit of the true key, we turn to perform statistical analysis of this collection, including sequence alignment to compensate for missing bits (as extra-bits are relatively rare, and we can therefore disregard the effect of such samples).

The basic idea is to employ majority decisions to find the true bits. However, performing such a majority rule bit-by-bit leads to significant errors, due to misalignment. We therefore consider blocks of u bits, where u is a tunable parameter, and perform majority votes on each block separately. In our implementation we set $u=16$. This approach addresses two problems in tandem; it handles both local swap errors (which are usually uncorrelated, and therefore the majority rule, even bit-by-bit, would suffice to address them), and also missing bit errors (which occur sparsely throughout the candidate keys).

We try and remedy any candidate key which disagrees with the majority on a certain window. To this end, we note that such disagreement tends to be more prevalent as we consider later bits (or windows) of the key. This is due to shifts occurring in the candidate key introduced by missing/extra bits. We address this problem by testing any such disagreeing window of a candidate key and shifting the window left and right for at most f bits, where f is a tunable parameter, shorter than the window length (i.e., $f < u$). In our implementation we use $f=3$. If any of these shifts results in the window agreeing with the majority for that window, then we

assume the number of missing bits, and their location, is hence known, and we assume the key is corrected accordingly. Otherwise, we remove the candidate key from the collection Σ, and disregard it in all later statistical analysis (e.g., in all later majority decisions).

This procedure eventually results in a single key being extracted from the set Σ of s candidate keys sampled from the cache side-channel. We recall that this single sequence is (presumably) aligned with the second 1-bit of the true key, and we perform the additional padding described earlier to obtain a valid-length key.

4.7.3 Verifying the correctness of the key

Once we extract the key from the set of candidate keys, it is straightforward to verify whether or not we have indeed extracted the target's private key. The attacker simply encrypts a message using the target's public key, and decrypts it using the extracted key. The key is correct if-and-only-if the decryption reveals the original message (assuming standard cryptographic assumptions on the hardness of RSA).

4.8 Testbed

Our testbed system consists of a single server, running 2 virtual machines, one hosting the attacking process and the other one hosting the target process, each running on its own core.

The server is a Dell machine equipped with a 4-core Intel Sandy Bridge 3.1GHz i5-2400 CPU and 3.7GB of RAM memory.

The host OS is Ubuntu 14.04 LTS, the hypervisor in use is KVM/QEMU 2.0.0, and the guest OS of the VMs is Ubuntu 14.04.2 server edition, configured with one VCPU and 0.5GB of memory each. Both VMs use huge pages in order to reduce memory fragmentation. The version of GMP used by the target is 2:5.1.3.

4.9 Preliminary results

In this section we provide some preliminary results on the performance of our attack implementation.

We study the accuracy level of our attack, as a function of the number of operations sampled by the attack. Recall that each operation takes $D=16$ µs, and we sample the cache for hit/miss once every $k=2$ µs, yielding 8-sample sequences capturing the exponentiation activity of the target as represented by cache activity in the monitored set. We consider running the attack using s samples, where s takes value in the range {5,10,15, ...,65}. For each such sample size s, we perform 1000 attempts to extract a private key. Figure 2 shows the average accuracy of

our attack. One can see that if the attacker uses more than 25 samples then the success probability is above 70%, which can be further amplified by repetition.

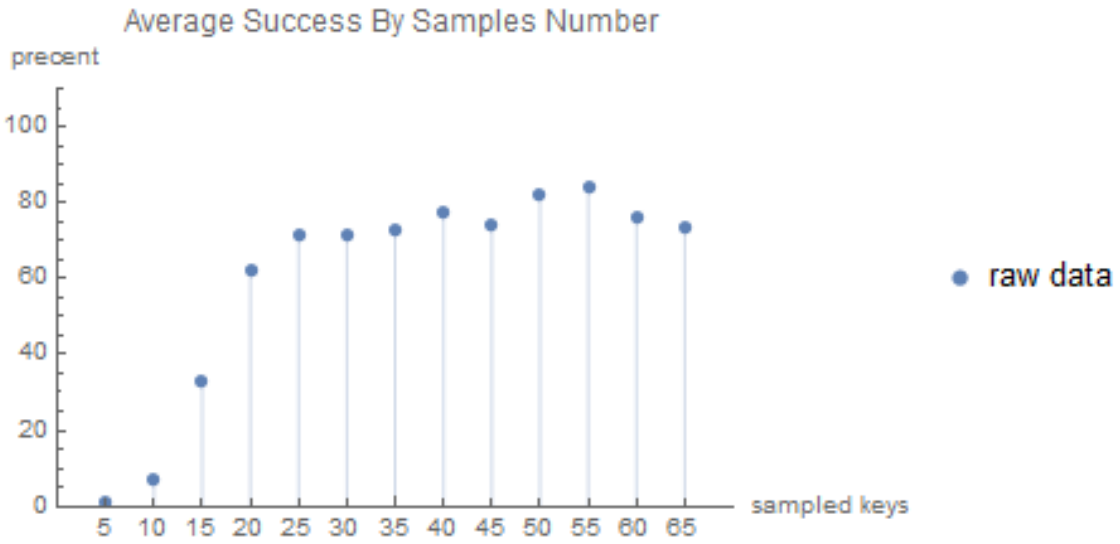


Figure 2: Average success as a function of the number of samples used to extract the key.

5 SCAM Architecture

The following diagram provides a high-level view of our proposed architecture for SCAM -- the sKVM Side-Channel Attack Monitoring/Mitigation module. The execution of this module will be controlled by a switch in sKVM, so that sKVM can decide at runtime whether SCAM operates, thus improving security and reducing performance. The role of SCAM is to provide a varied granularity of monitoring, profiling, and mitigation capabilities. SCAM will identify VMs that are attempting to extract information from co-located VMs via cache side-channels and mitigate the effects of these attacks.

SCAM is designed as a module within sKVM, where most of its functionality and components reside in user-space, along with a lower-level sub-module residing in kernel space. The role of the kernel sub-module is to facilitate the manipulation and access to kernel-level features and subsystems, most predominantly the physical memory of the host. The main modules of SCAM are the following:

(1) Monitoring module,
(2) Profiling module,
(3) Mitigation module, and
(4) Kernel module.

The operation of SCAM will most probably have an impact on the performance of the system as a whole, including on the VMs running over sKVM. Evaluating the tradeoff between security and performance is one of the key tasks in the design of SCAM and will be performed later in this project.

A detailed description of each of these modules, and the feature space that is planned to be explored within these modules, is provided in the following sections.

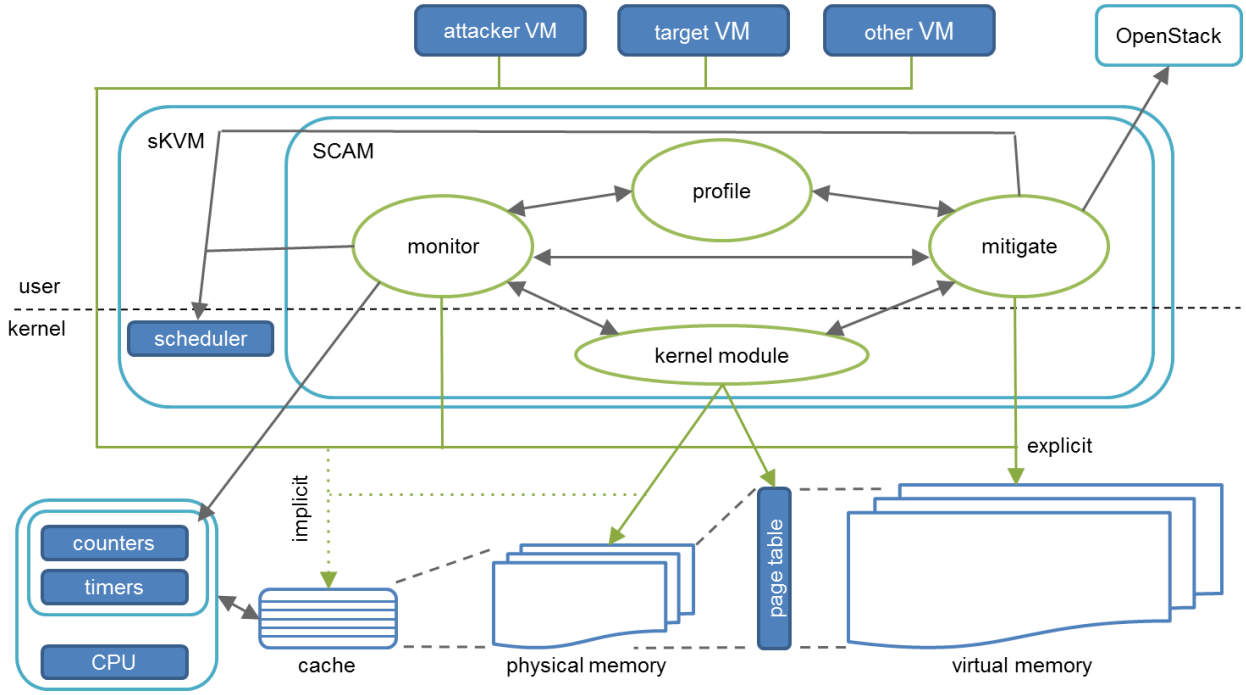


Figure 3: SCAM Architecture

5.1 Monitoring Module

The goal of the monitoring module is to collect data on the cache accesses of the VMs running on the host. Since SCAM has no prior information as to the identity of a potential attacking VM, this module collects information which will later be used to profile VMs by their cache activity. The profiling module, decides whether the activity of the monitored VM is deemed benign or hostile (see below).

The module is planned to monitor the cache access of the monitored VM. Since such access cannot be observed directly, we plan to explore several methods for performing indirect monitoring, including:

- **Tracing of cache hit/miss ratio**, the monitoring module samples counters of cache activity for all levels of the cache, attempting to deduce from them unusual VM behavior. These counters, described in more detail in the API section below, present a view of the total number of cache hits and misses.
- **Prime-and-probe**, the monitoring module essentially mimics an attacker's activity and obtains information about the access of the monitored VM to specific cache lines.
- **Emulation**, the monitoring module executes the code on behalf of the VM, and thus can trace its memory access.

The monitoring module interacts with the kernel module which is assumed to have unrestricted access to the physical memory. The monitoring module further receives privileged rights in terms of scheduling (by interacting with the sKVM scheduler), and may enforce its scheduling on any of the given cores (in order to monitor a VM that has been running on that core). Furthermore, the monitoring module requires access to counters and timers available at the kernel level in order to obtain timing and count information about the execution performed on each of the cores, along with cache access, which can later be used to infer the activity profile of the monitored VM.

5.2 Profiling

The role of the profiling module is to analyze the pattern of cache accesses of each VM and assign a score that represents the risk that a VM is conducting a cache-based side-channel attack. The input of the profiling module is the data that the monitoring module collects on each VM. The profiling module may trigger the operation of the mitigation module.

The basis for profiling VMs is a common characteristic of all currently known cache-based side-channel attacks, namely priming and probing specific cache-sets persistently. The profiling module characterizes the risk posed by a VM by the degree of similarity between the cache accesses of the VM and that of a generic attack. Given this basic assumption on profiling, the tasks that the profiling module performs are:

- Use monitoring information to identify such persistent probing of cache sets. The module may use low-overhead mechanisms such as comparing the number of probes of a cache-set, or the total number of successful and unsuccessful probes in the cache, with a given threshold. It may augment this approach by machine-learning methods as well.
- The module generates a risk score for each VM. If the risk score crosses a given threshold then the profiling module initiates the mitigation module.

5.3 Mitigation

The objective of the mitigation module is to reduce the effectiveness of cache-based side-channel attacks and prevent them completely where possible. The module takes action based on input from three possible sources. The profiling module may initialize mitigation action against a VM based on the risk score that is assigned to that VM. In addition, user applications may request protection for specific pages in memory even without any indication that there are malicious VMs running on the same hardware platform. This second option is a form of cross-layer interaction that significantly reduces the overhead incurred

compared to mitigating side-channel attacks aimed at data extraction from arbitrary memory locations. Finally, the mitigation module may be configured to perform some mitigation operations on the whole system regardless of the presence of malicious VMs. The mitigation module has a number of possible tools at its disposal. These include the following:

Page coloring - a form of software-configured unique assignment of part of the last-level cache to a core or a process. In this case the mitigation module requests the kernel to assign all the pages in physical memory that map to a specific area in the cache to a specific VM. If too many VMs run in parallel, page coloring results in fragmentation - partitioning the LLC to small sections which are each assigned to a single VM. The performance of the whole system is significantly reduced at this point.

Stealth pages - this approach combines a limited form of page coloring together with application level awareness of the mechanism. The idea, described in [13], is for the hypervisor to reserve a small number of pages for each VM. These reserved pages, called *stealth pages*, are locked into the cache for the duration of the VM's execution. The hypervisor must also expose an interface allowing the VM to be aware of the virtual addresses of the stealth pages. The VM ensures that all its sensitive code and data, e.g. instructions and data required for decryption, are stored on the stealth page. As long as the hypervisor and VM cooperate, and the VM is able to move sensitive operations and data to a relatively small page, this approach can be very effective. The Stealthmem paper [13] reports on minimal overhead, amounting to several percentage points, in the implementation of several popular symmetric encryption algorithms. Note that if the hypervisor uses large pages, which are two megabyte each, then it is likely that assigning a single stealth page to each VM covers the whole LLC with locked pages. Unless *all* the code and data of a VM can be compressed into a single page (which is highly unlikely with modern OS and applications) the result would be severe performance degradation, since most memory accesses would be to main memory rather than to any of the caches.

Cache flushing - the mitigation module repeatedly clears all or a number of cache sets before allowing a potentially harmful VM to run. This technique is especially effective when the attacker and the target share the same core. There are several ways to achieve flushing. The two simplest methods are to use the clflush instruction to flush a specific cache line or the privileged wbinvd instruction to flush the whole cache. In both cases all cache levels are flushed and in case of dirty memory (the cache has a newer value of the data compared to the address in main memory), there is write-back of cache data into RAM. The wbinvd instruction must be used with great care because its execution time is long (typically around 2000 clock cycles) and it has system-wide implications.

Tweaking OS scheduler and IPI - the mitigation module influences the priority of a potential attacker and its inter-process interrupts to reduce the effectiveness of an attack.

Changing memory page size - changing the page size of the system, e.g. from large pages to regular pages. Large pages simplify attacks on the last-level cache, due to improved visibility of the mapping of a virtual address to a cache set, whereas this measure has the opposite effect.

Changing virtual-physical memory mapping - the mitigation module modifies the page table, changing the mapping of physical pages to virtual pages. Together with using regular pages instead of large pages, this measure has the effect of reducing an attacker's knowledge on the mapping between virtual memory and the last-level cache. This method can be used on a potential attacker, a potential target, or without even identifying attackers.

Core migration - moving the VM of a potential attacker to a different core, e.g. to block attacks on a target executing on the same core as the attacker.

Eviction - requesting the OpenStack component to migrate an attacker to a different host, or even killing the VM in case of highly-reliable identification of an attacker

Oblivious RAM techniques - The concept of oblivious RAM was introduced in [9] and has since been improved and extended by a long line of works, e.g. [10] and [22]. The basic model considers a trusted CPU with a small attached memory unit (modeling a CPU's registers) and an untrusted Random Access Memory (RAM) unit. The CPU runs a program and stores the results in memory, but wants to make sure that the memory unit does not have access to the actual stored values or to the access pattern, i.e the sequence of accesses to memory addresses.

Oblivious RAM schemes work by using a data structure that stores encrypted pairs of the type <memory address, data>. The data structure resides in main memory and the encryption & decryption key is held by the CPU. The CPU accesses data by locating the required encrypted address in the data structure, decrypting the data and then processing it as the current instruction dictates. After completing the instruction the CPU proceeds by first re-encrypting the address and the contents stored in that address and then inserting the new encrypted pair into the data structure. The encryption ensures that the memory unit does not have access to the values that are actually stored. The data structure is constructed in a way that guarantees that any sequence of accesses to encrypted memory addresses results in the same distribution of accesses to the addresses of the data structure, which are the only addresses that the memory unit can observe.

A scheme for oblivious RAM effectively prevents cache-based side-channel attacks, because not only cache accesses, but even accesses to main memory are distributed independently of the data and code of the program. The main drawback of oblivious RAM is that its performance is not sufficient in most real-world scenarios. Apart from encryption and decryption of each memory block that is read by the CPU, the manipulation of the data structure places an overhead that is poly-logarithmic in the size of the memory for each access. Considering the memory requirements of modern applications that overhead could translate to a thousandfold slow-down.

However, oblivious RAM is an overkill as a mitigation method against cache-based side-channel attacks. Encryption of data content is unnecessary, because as long as the hypervisor is not compromised, a malicious VM can't view the data of a target VM. In addition, there is no need to hide the access pattern to the target's memory, but only the target's access pattern to the LLC, which is much coarser-grained than main memory. An attacker using prime+probe can only discover the target's access to cache sets. Therefore, hiding the access pattern to these sets is sufficient. Since the number of cache sets is far smaller than the number of addresses in main memory, significant savings may be achieved compared to an implementation of oblivious RAM. Furthermore, it may be possible to achieve further savings by allowing some information leakage on the access pattern to cache sets, as long as the rate of leakage is small enough.

The mitigation module requires access to the kernel module and a networking link that facilitates interaction with OpenStack. Both the mitigation module and the monitoring module should have a mapping of physical memory to the cache.

5.4 Kernel Module

The kernel module of SCAM provides the kernel services that the other modules require. These include access to timers and counters, read and write permissions to the page table, manipulation of VM scheduling, VM memory assignment and VM core assignment.

5.5 API

SCAM components require several interfaces to external modules. Some of these interfaces are standard in user space, while some of them need privileged services from the hypervisor. Standard interfaces include:

- A socket interface for communication with cloud middleware such as OpenStack.
- An interface to system counters that store hit/miss statistics on all cache levels. In Intel chips these counters include:

  - MEM_LOAD_RETIRED.LLC _MISS, that counts total misses on LLC cache during the monitoring interval (event CB in MSR register, unmask 10).
  - MEM_LOAD_RETIRED.HIT_LFB (all memory requests, that miss the L1 data cache and hit a line fill buffer, event CB in MSR register, unmask 40).
  - MEM_LOAD_RETIRED.L2_HIT (all memory requests that hit the L2 cache, event CB in MSR register, unmask 02).
  - MEM_LOAD_RETIRED.LLC_UNSHARED_HIT (all memory requests that hit the local LLC without snooping other sockets' LLC).
  - MEM_LOAD_RETIRED.OTHER_CORE_HIT_HITM (all memory requests that hit the other sockets' LLC).
  - MEM_LOAD_RETIRED.LLC_MISS (all memory requests that miss LLC and access the main memory).
  - L1D_CACHE_LD.MESI - all requests for L1 data cache reads (event 40, unmask 0F).
  - L1D_CACHE_ST.MESI - all requests for L1 data cache stores (event 41, unmask 0F).
  - L1D_PEND_MISS.PENDING – all outstanding L1 data cache misses at any cycle (event 48, unmask 01).
- Time measurements - are performed by the standard rdtsc instruction that returns the value of the Time Stamp Counter (STC), which counts the number of clock cycles since the last reboot of the system.
- Assuming a cross-layer in sKVM, it may present a method for a VM to define a critical memory page that should be secured against attacks.

Services that SCAM requires from sKVM (with the exact API yet to be defined):

- Page table configuration - SCAM needs to assign specific memory pages to a VM and to be able to invalidate memory pages.
- Page table reading - SCAM needs to read the mapping of physical memory pages to virtual memory pages and the assignment of such pages to virtual machines.
- Priority and scheduling - SCAM must be able to read the priority of a VM, assign it a new priority and manipulate its scheduling, e.g. stopping the execution of a VM for a configured time interval.
- Core assignment - SCAM must be able to configure core assignment of VMs and assignment of a SCAM monitoring or mitigation process to a given core.
- Cache flushing - SCAM needs an interface to execute the wbinvd instruction, which is privileged and executes a complete flush of all levels of cache memory.

5.6 Alternative Mitigation Strategies

SCAM attempts to monitor and mitigate cache-based side-channel attacks at a specific software layer, namely the hypervisor. Alternative approaches have been both proposed in the academic literature and implemented in practice. The following provides an illustration of the various mitigation methods published in the literature.

(1) Hardware-based approaches: custom hardware and hardened cache design,

(2) Application-specific approaches: modification of code to obfuscate secret-dependency of cache access [24],

(3) Tailored programming-languages: developing programming language primitives that enable the programmer to thwart timing variations at high-level code implementations [32],

(4) Cache-based approaches: reserving allocation of cache lines to secure functions [13], flushing core-bound caches (L1-L2) upon context switch following a secure operation [31], and page coloring which allocates L3 cache segments to a specific process by restricting access of other processes to those segments [23,3].

(5) Prefetching: introducing new cache prefetchers that obfuscate cache access by the target process [6],

(6) Scheduling-based approaches: ensuring no frequent context switches occur (which significantly reduces the rate of attack) by altering the host OS process scheduler [25],

(7) Producing various semantically-equivalent code fragments which exhibit diverse run-traces (e.g., timing, cache access, etc.) and randomly switching between them at runtime [4], and

(8) Restricted access to fine-grained timers and counters [15].

Furthermore, recent tools have begun to explore the possibility of statically analyzing the vulnerability of code fragments to cache side-channel attacks, by analyzing potential cache-states under various cache eviction policies [5].

An attacker who tries to extract information from a target VM via a cache-based side-channel attack can often use an additional side-channel based on time measurements. The overall approach in mitigation techniques of remote timing-based attacks is ensuring that observable events are independent of data (e.g., private keys). This is usually achieved using padding instructions inserted into the code in order to ensure that all execution paths are of roughly the same length, thus eliminating the variance required for performing such attacks.

Padding usually results in significant degradation in performance, as the target execution path length is determined by the longest such path. This latter trait has recently been improved to require padding only for secure execution paths [3]. These approaches are sometimes complemented by application-specific modifications of the code in order to obtain such independence. However, these latter approaches are extremely hard to design and implement as running-time parameters (such as cache usage) are not available in design time, which leads to insufficient information availability in order to successfully implement such approaches.

Two of the most important alternatives to the SCAM approach are modifying cache hardware to prevent side-channel attacks and changing the memory access patterns of applications. The importance of these methods is due to the fact that both are implemented in commercially available chips and in open-source software.

There have been several proposals in the academic literature for cache designs that prevent cache-based side-channel attacks, e.g. [19] and [27]. Similar designs already appear in some commercial chips [12] (which we discuss in greater detail in a following section), although the main driver for their adoption is performance in certain use-cases rather than security. It is possible to create a comprehensive hardware solution that completely prevents cache-based side-channel attacks, e.g. by providing physically separate caches to different VMs. However, the jury is still out on whether such measures will be widely accepted and what their impact on performance and cost will be. As long as cloud services do not universally use these hardware measures in a strict way, cache-based side-channel attacks will still be possible and software monitoring and mitigation of these attacks will still be desirable.

Application specific measures to mitigate side-channel attacks involve modifying the memory access pattern of the application so that the same memory lines of the cache are accessed regardless of the specific value of any secret data item that the application uses. Even if an attacker learns the exact sequence of cache accesses which the target application performs that knowledge does not provide any information on the application's secret data, given such a defense.

A prime example of changing the access pattern of specific applications is implemented in the OpenSSL library [16], which we describe in more detail in the next section. OpenSSL is the most popular open source implementation of the SSL/TLS protocol, and is one of the most important open source cryptographic libraries. Like any implementation of TLS it includes a computation of RSA signatures. However, unlike many other implementations, in OpenSSL there are specific measures to prevent a cache side-channel attack on RSA computation.

Application specific measures can be very effective but suffer from several flaws. The first is that it is often difficult to enumerate all the possible secrets of an application. For example, in TLS the RSA private keys are the most attractive target, but session keys that are used by such symmetric-key algorithms as AES are also important as they hide sensitive information including credit card numbers. However, these algorithms are also vulnerable to certain cache-based side channel attacks [18], [24]. Protecting AES against cache-based side-channel attacks requires a completely different memory access modification from the method used in OpenSSL to secure RSA.

However, the worst problem for application specific defenses is that every application writer has to become intimately aware of all the secret data in the application and invent ad-hoc methods to secure it against cache-based side-channel attacks. This task is far from trivial as is shown in an attack on anonymity reported in [17]. The attack identifies which web sites a user accesses when browsing through TOR. The target of the attack is new and it is not even clear which application should be responsible for mitigating the attack.

The two following sections provide a more detailed description of two of the specific alternative strategies to mitigate or even prevent cache-based side-channel attacks.

5.6.1 Defense Mechanism - OpenSSL

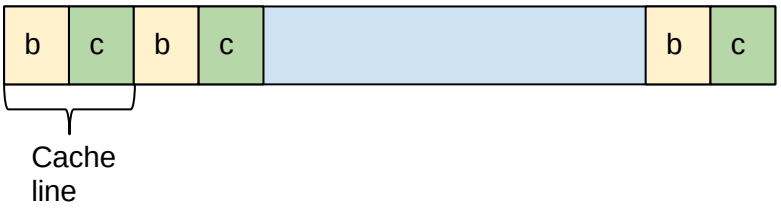
While the basic algorithm used in OpenSSL is slightly different than the pseudo code we show for RSA exponentiation (it uses the so-called window method), the same defense mechanism against side-channel attacks can be used in the square-and-multiply algorithm we have shown. Recall that for each bit in the private key the pseudo-code computed the square of a variable z and if the bit is 1 it also computes a product $z=zc \bmod n$. The basic step of the side-channel attack is to look for cache sets that are written when computing $z=zc \bmod n$, thus learning a bit of the secret key.

The protection mechanism keeps three variables: **c**, which is already part of the code, **b** which is equal to the constant 1 and **a**. As a first step consider changing the if statement to

```
if (d[i]==1)
  a=c;  //(1)
else
  a=b;  //(2)
product (&z, z, z, n);     //z=z^2 mod n
product (&z, z, a, n);     //z=za mod n
```

This code snippet computes the same function, but always computes a product of z and c. However, the attack is still possible in this case because the attacker can distinguish between the cache sets used to execute instruction (1) or instruction (2).

OpenSSL solves this problem by creating a single array **v** that is large enough to hold both b and c. These two smaller arrays are embedded in v in an interleaved fashion such that a memory access to either one writes *both* b and c to the cache. That way the same cache lines are accessed regardless of the binary representation of the private key. In more detail, b and c are represented as *k* blocks, *b=b1,...,bk* and *c=c1,...,ck*, where each block is half the length of a cache line (that typically translates to 32 bytes). The following diagram shows the array *v*



and the following code replaces the previous if statement:

```
#define HALF_LINE 32
index=d[i]*HALF_LINE;
for(i=0;i<k;i++)
  memcpy (a+k*HALF_LINE,v+2*k*HALF_LINE+index, HALF_LINE);
product (&z, z, z, n);      //z=z^2 mod n
product (&z, z, a, n);    //z=za mod n
```

Due to the fact that each pair of successive blocks bi and ci are loaded (together) to the same cache line the access pattern to the cache is the same independently of the value of d, which prevents the type of attack we implemented in our testing tool.

5.6.2 Defense Mechanism - Cache Allocation Technology

The Intel Corporation has introduced several technologies to speed up Network Function Virtualization [12]. Two of these technologies are Cache Monitoring Technology (CMT) and Cache Allocation Technology (CAT) which are implemented in the Intel Xeon processor E5-2600 v3 family of chips.

CMT provides information to a hypervisor (or a VM with the necessary authorization) on the cache usage profile of different VMs. Some VMs with low priority may use the last-level cache so intensively that they degrade the performance of other, high-priority VMs. CAT can be used either independently or in conjunction with CMT to partition the LLC between different

VMs. Memory accesses of each VM are mapped only to its own partition, thereby preventing inter-VM contention for cache lines.

The main objective of Intel in introducing CMT and CAT is performance in specific circumstances. However, it can also be used to prevent cache-based side-channel attacks. The attacker can only deduce information from cache addresses that it shares with the target. If the two VMs run on different cores than they do not share L1 and L2 caches and if CAT partitions the cache such that they don't share the LLC then there are no shared cache lines.

Wide scale adoption of CAT relies on many factors besides security, including first and foremost performance across many use cases, compatibility, cost and others. As long as a large number of servers in the cloud do not support CAT, software solutions of the type that we propose will be beneficial. In addition, even widespread use of CAT does not necessarily guarantee safety against cache-based side-channel attacks if CAT is used dynamically, based on CMT or other measurements of cache activity.

6 Key Takeaways

The two main contributions of this deliverable are an implementation of a testing tool for cache-based side-channel attacks and an initial design for SCAM, which is a hypervisor module that identifies and mitigates such attacks. The key takeaways for the testing tool are:

- The testing tool extracts RSA (or Diffie-Hellman) private keys from co-located targets.
- The tool makes the following mild assumptions, which occur often in real-world cloud environments:
    - The attacking VM has a dedicated core.
    - The target VM uses the square-and-multiply algorithm for computing private-key operations.
    - Both VMs run over KVM or are processes running over bare-metal.
- The tool reliably extracts standard RSA keys of length 2048 bits or extra-secure keys of length 4096 in very short time, typically within several minutes.

The key takeaways for the SCAM module are:

- The module will be made up of four components:
    - A monitoring sub-module to collect information on VM cache usage.
    - A profiling sub-module to evaluate whether a VM is conducting a cache-based side-channel attack using the information from the monitoring component.
    - A mitigation sub-module to diminish the effect of cache-based attacks and even prevent them.
    - A kernel sub-module that acts as an interface between kernel services and the rest of SCAM.
- A variety of methods are available for the monitoring and mitigation components.
- SCAM will work in the hypervisor software layer and will require no hardware modifications.
- SCAM mitigation may work independently of the application layer, but can benefit from information from potential target applications.

7 Concluding Remarks

In this document we present the first version of the security concepts developed within the MIKELANGELO project. We describe both the attack implemented in the first year of the project, which will serve as a testing tool for the mitigation techniques to be developed later on in the project. We also describe the overall architecture of the SCAM module within sKVM, which will orchestrate and perform the various security-oriented tasks within the project.

In the future, it would be highly interesting to see if such attacks can also be implemented in other virtualized technologies, such as the ones based on containers, and also evaluate whether the mitigation techniques could be applied in such environments.

8 References and Applicable Documents

[1] The MIKELANGELO project, http://www.mikelangelo-project.eu/

[2] A. Abusharekh and K. Kaj, "Comparative Analysis of Software Libraries for Public Key Cryptography," in SPEED 2007, Amsterdam, the Netherlands, Jun 2007, pp. 3-19.

[3] B. A. Braun, S. Jana, and D. Boneh, "Robust and Efficient Elimination of Cache and Timing Side Channels". Manuscript (available at http://arxiv.org/abs/1506.00189)

[4] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity,", in NDSS, San Diego, CA, US, Feb 2015.

[5] G. Doychev, B. Köpf, and A. Rybalchenko, "CacheAudit: A Tool for the Static Analysis of Cache Side Channels," ACM TISS, no. 18, vol. 1, Jun 2015.

[6] A. Fuchs and R. B. Lee, "Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs," in SYSTOR, Haifa, Israel, May 2015.

[7] https://gmplib.org/

[8] http://www.gnutls.org/

[9] O. Goldreich and R. Ostrovsky. "Software protection and simulation on oblivious RAMs." Journal of the ACM (JACM) 43.3 (1996): 431-473.

[10] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. "Oblivious RAM simulation with efficient worst-case access overhead." In Proceedings of the 3rd ACM workshop on Cloud computing security workshop, pp. 95-100. ACM, 2011.

[11] M. S. Inci, B. Gülmezoglu, G. I. Apecechea, T. Eisenbarth, and B. Sunar, "Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud", IACR Cryptology ePrint Archive 2015: 898 (2015)

[12] Intel Corporation, "Enabling NFV to Deliver on its Promise", http://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/nfv-packet-processing-brief.pdf

[13] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud," in USENIX Security, Bellvue, WA, US, Aug 2012, pp. 189–204.

[14] F. Liu, Y. Yarom, Y., Q. Ge, G. Heiser and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical." 36th IEEE Symposium on Security and Privacy (S&P 2015). 2015.

[15] R. Martin, J. Demme, and S. Sethumadhavan, "TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-Channel Attacks," in ISCA, Portland, OR, US, Jun 2012, pp. 118–129.

[16] https://www.openssl.org/.

[17] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: practical cache attacks in JavaScript and their implications." Proceedings of

the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015.

[18] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," http://www.cs.tau.ac.il/~tromer/papers/cache.pdf, Nov 2005.

[19] D. Page, "Partitioned Cache Architecture as a Side-Channel Defence Mechanism." IACR Cryptology ePrint Archive 2005 (2005): 280.

[20] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off my cloud: Exploring information leakage in third-party compute clouds," in CCS, Chicago, IL, US, Nov 2009, pp. 199–212

[21] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," CACM, no. 2, pp. 120–126, Feb 1978.

[22] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. "Path oram: An extremely simple oblivious ram protocol." In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pp. 299-310. ACM, 2013.

[23] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory Deduplication as a Threat to the Guest OS," in EUROSEC, Salzburg, Austria, April 2011.

[24] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks in AES, and countermeasures," J. Cryptology, no. 2, pp. 37–71, Jan 2010.

[25] V. Varadarajan, T. Ristenpart, and M. M. Swift, "Scheduler-based Defenses against Cross-VM Side-channels," in USENIX Security, San Diego, CA, US, Aug 2014, pp. 687–702.

[26] VMware Inc., "Security considerations and disallowing inter-virtual machine transparent page sharing," VMware Knowledge Base 2080735 http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=2080735, Oct 2014.

[27] Z . Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks." ACM SIGARCH Computer Architecture News. Vol. 35. No. 2. ACM, 2007.

[28] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyperspace: High-speed covert channel attacks in the cloud," in USENIX Security, Bellevue,WA, US, 2012, pp. 159–173.

[29] Y. Yarom and K. Falkner, "FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack," in USENIX Security, San Diego, CA, US, Aug 2014.

[30] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in CCS, Raleigh, NC, US, Oct 2012, pp. 305–316.

[31] Y. Zhang and M. K. Reiter, "Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud," in CCS, Berlin, Germany, Nov 2013, pp. 827–838.

[32] D. Zhang, A. Askarov, and A. C. Myers, "Language-Based Control and Mitigation of Timing Channels," in PLDI, Beijing, China, Jun 2012, pp. 99–110.