Gorka Irazoqui, Mehmet Sinan İncİ, Thomas Eisenbarth, and Berk Sunar

Know Thy Neighbor: Crypto Library Detection in Cloud

Abstract: Software updates and security patches have become a standard method to fix known and recently discovered security vulnerabilities in deployed software. In server applications, outdated cryptographic libraries allow adversaries to exploit weaknesses and launch attacks with significant security results. The proposed technique exploits leakages at the hardware level to first, determine if a specific cryptographic library is running inside (or not) a co-located virtual machine (VM) and second to discover the IP of the co-located target. To this end, we use a *Flush+Reload* cache side-channel technique to measure the time it takes to call (load) a cryptographic library function. Shorter loading times are indicative of the library already residing in memory and shared by the VM manager through *deduplication*. We demonstrate the viability of the proposed technique by detecting and distinguishing various cryptographic libraries, including `MatrixSSL`, `PolarSSL`, `GnuTLS`, `OpenSSL` and `CyaSSL` along with the IP of the VM running these libraries. In addition, we show how to differentiate between various versions of libraries to better select an attack target as well as the applicable exploit. Our experiments show a complete attack setup scenario with single-trial success rates of up to 90% under light load and up to 50% under heavy load for libraries running in KVM.

Keywords: Cryptographic Libraries, Cross-VM attacks, Virtualization, Deduplication

1 Motivation

Cloud computing has become a major building block in today's computing infrastructure. Many start-up and mid scale companies such as Dropbox1 leverage the ability to outsource and scale computational needs to cloud service providers (CSPs) such as Amazon AWS2 or Google Compute Engine. Other companies may build their computational infrastructure in the form of private clouds, harnessing cost savings from resource sharing and centralized resource management within the company.

Nevertheless, one of the main concerns that is slowing the widespread usage of such Infrastructure as a Service (IaaS) technologies are potential security vulnerabilities and privacy risks of cloud computing. Usually, CSPs employ virtual machines (VMs), allowing multiple tenants to share the same computing hardware. While this resource sharing maximizes utilization and hence drastically reduces cost, ensuring isolation of potentially sensitive data between VMs instantiated by different and untrusted tenants can be a challenge. Indeed, the main security principle in the design and implementation of virtual machine managers (VMMs) has been that of process and data isolation achieved through *sandboxing*. Although logical isolation ensures security at the software level, a malicious tenant might still extract private information due to leakage coming from *side channels* such as shared hardware resources. In short, hardware sharing create an opening for various side channel attacks developed for non-virtualized environments. These powerful attacks are capable of extracting sensitive information, e.g. passwords and private keys, by profiling the victim process. ignoring leakages of information through subtle side-channels shared by the processes running on the same physical hardware. In non-virtualized environments, a number of effective side-channel attacks were proposed that succeeded in extracting sensitive data by targeting the software layer only.

As early as in 2003, D. Brumley and Boneh [22] demonstrated timing side-channel vulnerabilities in `OpenSSL 0.9.7` [44], in the context of sandboxed execution in VMMs. The study recovers RSA private keys

Gorka Irazoqui: Worcester Polytechnic Institute, E-mail: girazoki@wpi.edu
Mehmet Sinan İncİ: Worcester Polytechnic Institute, E-mail: msinci@wpi.edu
Thomas Eisenbarth: Worcester Polytechnic Institute, E-mail: teisenbarth@wpi.edu
Berk Sunar: Worcester Polytechnic Institute, E-mail: sunar@wpi.edu

1 Dropbox grew from nil to an astounding 175 million users from 2007 to 2013 [3].
2 Amazon AWS generated $3.8 billion revenue in 2013 [1].

from an OpenSSL-based web server when victim and attacker run in the same processor. Symmetric cryptography is another popular target of side channel attacks, as demonstrated in 2005 by Bernstein [17] (and later in [20, 28, 36]). He was able to recover an AES key due to micro-architectural timing variations in the cache. Time leakage in branch prediction units give rise to another class of side channel attacks as demonstrated by Acıiçmez et al. where the authors exploited key dependent branches in a RSA computation of OpenSSL [13–15]. Recently B.B. Brumley and Tuveri [21] demonstrated that the ladder computation in the popular ECDSA implementation of OpenSSL 0.9.8o is vulnerable to timing attacks by extracting the private key used in a TLS handshake.

Until fairly recently, the common belief was that side-channel attacks were not realistic in the cloud setting due to the level of access required to the cloud server, e.g. control of process execution and more specifically the difficulty in co-locating the attack process on the machine executing the victim's process. This belief was overturned in 2009 by Ristenpart et al. [39] who demonstrated that it is possible to solve the co-residency problem and extract sensitive data across VMs. Using the Amazon Elastic Compute Cloud (EC2) service as a case study, Ristenpart et al. demonstrated that it is possible to identify when an attacker VM is co-located on the same server with a potential victim and therefore using the same hardware as the attacker VM. The work further shows that—once co-located—cache contention between the VMs can be exploited to recover key strokes across VM boundaries. By solving the co-location problem, this initial result fueled further research in Cross-VM side-channel attacks. Zhang et al. [53] utilized a cache side channel attack implemented across Xen VMs to recover an ElGamal decryption key from a victim VM. The authors applied a hidden Markov model to process the noisy but high-resolution information across VMs. Shortly thereafter, Yarom and Falkner showed in [51] that RSA is also vulnerable, as well as ECDSA, as shown by Yarom and Benger in [50]. Both attacks use the *Flush+Reload* technique that exploits the shared L3 cache. It is important to note that since the L3 cache is shared among cores, the attack works even if the victim and the attack processes are located on different cores.

At the system level, another significant consequence of these new high resolution cache attacks is that they exposed new vulnerabilities in popular cryptographic libraries such as OpenSSL, which were previously considered secure. This forced the cryptographic library developers to fix their implementations and release patches to mitigate these new attacks. It should be noted that side channel attacks are just one more threat to cryptographic libraries from a long list of vulnerabilities regardless of whether the library is executed in VMs or natively. Therefore, it is crucial to use the most recent version of the cryptographic libraries where most of the discovered vulnerabilities have been already mitigated via series of patches. Using an outdated cryptographic library renders the server vulnerable with potentially devastating consequences. Good examples of recently outdated cryptographic libraries are the ones susceptible to Lucky Thirteen and the infamous Heartbleed attacks. In 2013, AlFardan et al. discovered that most cryptographic libraries were vulnerable to a padding oracle attack that they named Lucky 13 attack [27]. The attack was able to recover the messages sent in TLS connections by just looking at the time difference caused by invalid padding in digest operations. Although they showed only results in OpenSSL and GnuTLS, most of the cryptographic libraries were affected by it. Patches were released immediately by library developers. But using a non-patched version of any of these libraries would still leave the door open to this MEE attack.

In 2014, Neel Mehta from the Google security team discovered a dangerous security bug called the Heartbleed bug in the popular OpenSSL cryptographic library [4]. In a nutshell, the Heartbleed vulnerability allows a malicious attacker to read more data in the victim's memory than it should, exposing sensitive information like secret keys. The attack quickly became popular in the media, and caused grave concern among security researchers for such a simple yet severe vulnerability to go undetected for many years. The cybersecurity columnist Joseph Steinberg argued that the Heartbleed bug is the worst vulnerability found since commercial traffic began to flow on internet. Indeed, 17% of supposedly secure web servers were found to be vulnerable to the attack. Soon after instances of Heartbleed attack were discovered in the wild. For instance, the Canada Revenue Agency reported the theft of 900 social security numbers [23]. Most of the websites patched the bug within days of its release, but it remains unclear whether the vulnerability has been exploited before its public announcement. It is believed that some attackers might have been exploiting the vulnerability for five months before Mehta's discovery.

The Heartbleed vulnerability presents a striking example on the severe consequences of using an outdated cryptographic library and a striking example of how people ignore up-to-date software. One month after the Heartbleed bug was discovered, 300k of 600k systems were patched [43]. But the month after that only 9k additional systems were patched leaving the remaining 300k systems still vulnerable. This dramatic drop in the number of patched systems show that the Heartbleed patching is almost over even though the security risk still persist. In general, even when more critical vulnerabilities are discovered, there is an inherent inertia in patching systems gives attackers a window of opportunity to penetrate computing systems. During this narrow window an attacker has to first run a discovery phase where the system is target computing platform through a series of tests to identify the most vulnerable point of entry in the subsequent attack phase 3. During the first phase it is critical for the attacker remains stealthy. Therefore, a discovery tool that permits such facility, i.e. covertly detecting a particular installation of an outdated cryptographic library, will be an indispensable tool in the hands of an attacker.

1.1 Our Contribution

In this work we introduce a method to detect the execution of specific software co-residing in the same host across VM-boundaries. In particular we show that cryptographic libraries executed in a co-located server in the cloud can be detected with high accuracy, and that specific versions of those libraries can also be distinguished reliably. After this library detection stage, we show to that the IP of the co-located VM running the target library is also recoverable in short time. The technique can enable malicious parties, to covertly detect vulnerable cryptographic libraries/versions prior to performing an attack. The detection method exploits subtle leakages of timing information at the hardware level and runs rather quickly. At the protocol level the detection technique does not interfere with the target's usual operations. Therefore, the detection method is very hard to detect by the target.
Specifically, our contributions can be summarized as follows:

- We demonstrate that it is possible to *detect* and *classify* executed code across VM boundaries with high accuracy;
- We show *how* the developed method can be applied to determine the cryptographic library being used by a co-located VM;
- Our results imply that a co-located VM can discover use of outdated vulnerable versions of a library; This means that for the first time we show how to distinguish between vulnerable and non-vulnerable crypto-library versions across VMs.
- We show that after detecting the criptographic library, the IP address of the co-located VM can be recovered in seconds to minutes depending on the network size.
- We present a test bench with a popular cloud hypervisor KVM that proves the viability of our detection method.

We present empirical results for detecting the execution of `MatrixSSL`, `PolarSSL`, `GnuTLS`, `OpenSSL` and `CyaSSL` cryptographic libraries when running inside a VM in KVM, as well as distinguishing and detecting specific *versions* of such libraries, in particular `OpenSSL` versions 0.9.7a, 0.9.8k, 1.0.0a, 1.0.1c, 1.0.1e, 1.0.1f, and 1.0.1g. Also, we present how long it takes to recover the IP address of the co-located VM running aforementioned libraries. Our detection method obtains a success rate of up to 90% under low noise scenarios and a success rate of up to 50% under heavy load scenarios, while maintaining a negligible false-positive rate. This means that even when the workload is sufficiently high, our detection method detects almost one out of two library calls made by the target, and virtually never incorrectly detects a library.

2 Background

In this study we develop a method that detects whether a co-located VM is running a specific cryptographic library and to furthermore determine the specific version of the library as well as the IP address of the co-located VM. Described attack scenario requires a detection technique that must work across cores to be realistic in a cloud setting. For this purpose we need to exploit a type of shared resource between cores in multicore systems that leaks private information. Several single core resources like branch predictors and TLBs have been exploited in the past. However, information about these shared resources in modern multicore modern processors is scant at best, and for most cases is not available to the public. In contrast, in most modern processors the cache *can* acts as the covert channel needed for our detection method. In this section we give a brief description of typical cache architectures and a quick overview

3 Directly running the attack carries varying levels of risks of exposing the attack depending on the nature of the attack.

on related work involving cache based side channel attacks. Finally we describe the tool that this study will use to detect cryptographic libraries in co-located VMs.

2.1 The Cache as a Covert Channel

2.1.0.1 Cache Hierarchy

The cache is a small memory element that is located between the main memory and the CPU cores with the purpose of providing faster access to the required data. Upon a memory request, if the data resides in the cache a cache hit occurs and the access time is small. If the memory line is not found in the first level of cache, a cache miss occurs and the data has to be fetched from the lower level caches or the main memory.

Caches base their functionality on two main principles: the *temporal* and *spatial* locality principles. The first predicts that recently accessed data is likely to be accessed soon, whereas the latter predicts that data in nearby memory locations to the accessed data are also likely to be accessed soon. Thus, when a cache miss occurs, caches fetcg an entire memory block (cache line) containing both accessed and nearby memory locations.

2.1.0.2 Related Cache Side-Channel Attacks

In general all the memory-dependent cryptographic algorithms may potentially be exploited by cache attacks, if no countermeasures are provided. Motivated by this observation, a number of researchers have targeted weakly designed software algorithms. The first study considering cache-memory accesses as a covert channel targeting the extraction of sensitive data was done by Hu [29]. However it was not until 1998 when Kelsey et al. [33] studied the cache hit ratio as a method to deploy the first cache side-channel attacks. Page in 2003 suggested theoretical methods on cache side channel attacks [37]. The application of these kind of techniques to recover information from typical table look up operations performed in symmetric ciphers was first studied by Tsunoo et al. [45]. The first practical cache attack implementation appeared in 2005. Bernstein [17] demonstrated that the timing leakage due to different cache line accesses can be used to recover the secret key used by the AES algorithm. In the same line, Bonneau et al. [19] showed how collisions in the last round of AES can affect the overall time execution and give information about the key used by the algorithm. These two attacks can be considered as time driven cache attacks, since they obtain information by just looking at the overall timing execution. At the same time, several new trace driven attacks were proposed. Osvik et al. [36] introduced two new side channel techniques and where able to extract an AES key: `evict+time` and `prime+probe`.will be translated into a higher probing time. If not, the attackers data will still be present in the cache, and the probing time is going to be smaller. Acıiçmez showed that cache attacks not only work when they target the data cache, but also when they monitor the instruction cache [12]. He used a `prime+probe` technique to monitor whether a RSA operation was calling square or multiply operations, and thereby recovered the private key.

In 2009, Ristenpart et al. [39] showed that it is possible to solve the co-location problem in cloud environments and therefore use the same hardware resources as your victim. They targeted EC2 Amazon Web services and demonstrated that an attacker can obtain co-location with 40% chance. Their work for the first time opened the door for side channel attacks in the cloud setting. Two years later, Zhang et al. used the cache as a tool to determine whether a user is co-located with someone else or not [52]. Just one year later, again Zhang et al. managed to recover an ElGamal encryption key in a cloud scenario running XEN hypervisor when the adversary is co-located in the same core [53]. They used the above described `prime+probe` technique.

In the last three years, a new technique emerged for cache analysis: *Flush+Reload*. The first research to utilize this technique was Gullasch et al. [28], in which they managed to recover an AES encryption key by using the Complete Fair Scheduler to block the encryption execution. In 2013, Yarom et al. [51] used the tool to recover a RSA decryption key running in GNUTLS. One year later they used the same technique to recover an ECDSA decryption key running in `OpenSSL` [50]. Irazoqui et al. [31] managed to recover an AES encryption key in a real cloud scenario without the necessity of blocking the AES execution (c.f. [28]). Lastly, Zhang et al. [54] demonstrated that *Flush+Reload* is also applicable in Product as a Service (PaaS) clouds by recovering sensitive information from a co-located victim.

2.2 *Flush+Reload* Spy Process

Flush+Reload is a powerful cache-based spy process first studied in [28] where Gullasch et al. demonstrated that it can be used to recover sensitive information such as AES secret keys. It was not until 2013 when the spy process acquired this name thanks to Yarom et al. when they used it to recover RSA encryption keys [51]. The spy process checks if targeted process accesses some monitored memory lines. *Flush+Reload* is works in three main stages:

– **Flushing stage**: In this stage, the spy process flushes some desired memory lines from the cache hierarchies of all the cores. This means that after this stage, the monitored memory lines will not be present in any cache level of any core in the server. Instead they will reside in the main memory. This is accomplished by using the `clflush` command.
– **Target process stage**: In this stage the target runs a program which may or may not use the monitored memory lines. If the program uses any of the monitored lines, this will be loaded in the corresponding core's cache hierarchy, from the last level of cache to the first one. However if none of the monitored memory lines is used by the target program, these will reside in memory.
– **Reloading stage**: In this stage the attacker reloads again the monitored cache lines. If the monitored memory lines reside in the cache, the reload time is going to be small. On the other hand if they reside in main memory, the reload time is going to be longer. If the last level of cache is shared across cores, a spy process working in a different core would still be able to distinguish accessed from memory lines that were not accessed.

3 Memory Deduplication Features

Memory deduplication is an optimization technique that is implemented in operating systems and cloud hypervisors with the goal of utilizing memory more efficiently. The basic idea is to merge identical pages used by different processes or virtual machines into one shared page with a Copy-on-Write flag. After the merging, if one of the processes try to modify this shared page, a duplicate page is created and assigned to the process hence cutting if from the shared page. The deduplication process is performed by checking duplicate pages in memory via first hash match checking then a bit-by-bit comparison of the data. This is especially effective in virtualized environments where multiple guest OSs co-reside on the same physical machine and therefore share the provided hardware resources. Many variations of deduplication techniques are now implemented in OSs and hypervisors e.g. Kernel Samepage Merging in Linux and Transparent Page Sharing in VMware. Although they have

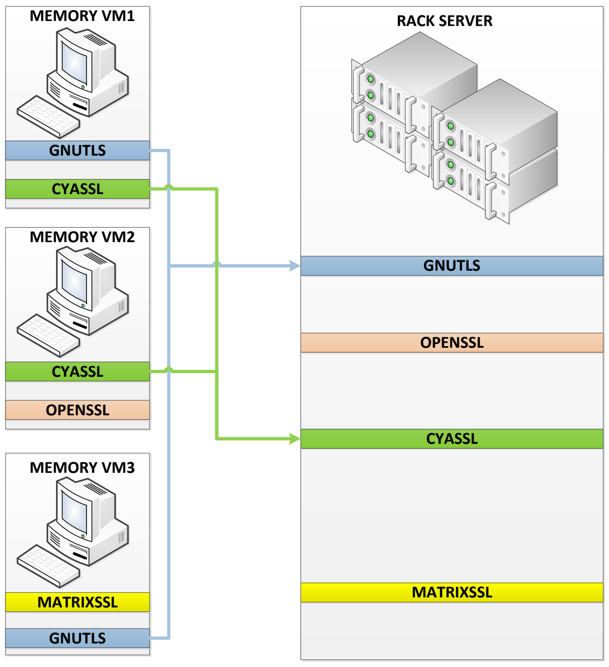

Fig. 1. Memory Deduplication Feature

different names, both implement the above described mechanism to handle duplicate pages.

Note that even though the memory deduplication saves significant amount of physical memory under certain conditions, it also opens doors to side-channel attacks like *Flush+Reload*. For this reason, it is disabled by major public cloud service providers like Amazon EC2, Rackspace, etc., and is generally used in private clouds.

In this work we will focus on KSM, since this is the deduplication technique used in our testbed hypervisor, KVM.

3.1 KSM (Kernel Same-page Merging)

KSM is the Linux memory saving deduplication feature that was first provided in Linux kernel version 2.6.32 [5, 32] aiming at avoiding redundant memory copies [2, 16]. The procedure is as following: when KSM finds a candidate page to be shared, it creates a signature and stores it in a deduplication table. Each time KSM generates a new signature it compares it against the signatures stored in the table. If KSM finds that two pages have created the same signature, it merges them. By default KSM in KVM scans 100 pages in every 200 milliseconds. This is why any memory disclosure attack, including ours, has to wait for a certain time before the deduplication takes effect upon which the attack can be performed [40–42].

The mechanism for the cross-VM scenario is very similar to the one described above. KVM for example uses the KSM mechanism among their VMs. In this case, KSM performs memory merging techniques among VMs instead of among processes. Therefore when the same page is detected as used by two different VMs, the page is shared if it is declared as shareable. The client OS still performs KSM among the processes running inside it. KVM is not the only hypervisor that implements deduplication. VMware uses a similar technique called Transparent Page Sharing (TPS) with the goal of improving performance among their VMs via memory deduplication [48].

4 Threats of Library Detection

As described in Section 2, Zhang et al. [54] demonstrated how to track the execution path of a particular victim when co-located in the same virtual machine in PaaS clouds. In contrast, our work:

– Is applied in IaaS clouds where the information leakage appears across different VMs. However, following the work by Zhang et al. [54] our work is also applicable in PaaS clouds where different users run in the same guest OS. Hence, deduplication ceases to be an issue.
– Detects vulnerable cryptographic libraries instead of the number of items in a shopping cart. Note that this is a substantially higher threat, since harmful attacks can be mounted if a vulnerable deployment is detected (e.g., the Heartbleed attack).
– Directly discovers the IP address of the target VM by tracking the execution path. In a PaaS scenario, both attacker and victim share the same IP address, eliminating the IP detection phase.
– Profiles accesses to handpicked addresses of a cryptographic library made by a target VM. Zhang et al., instead, first train a non-linear finite automaton and use it later to track the execution path of the co-located victim's processes.

As mentioned before, the motivations for detecting the execution of a specific piece of software being used can be manifold. The knowledge of the cryptographic library being used can give crucial information to a co-located adversary. For example, each cryptographic library has unique features and therefore can be used for fingerprinting. Furthermore, common attacks are not addressed in the same way in all the libraries. Some libraries have weaker patches than the others. This tool gives to the attacker the ability to determine whether a library that a co-located tenant is using is vulnerable against a certain type of attack. In this work we focus on the most popular cryptographic libraries: `GnuTLS`, `OpenSSL`, `PolarSSL`, `MatrixSSL` and `CyaSSL` [7, 11, 35, 38, 44]. These 5 libraries have different cryptographic implementations and from side-channel point of view some of them can be more secure than others. In this section we give examples why an attacker could benefit from the knowledge of the cryptographic library being used.

AES: AES is the most popular block cipher in use today. Its implementation among the different libraries varies, with `OpenSSL` having cache attack mitigation techniques in their implementation such as bit slicing techniques, while other libraries such as `PolarSSL` do not implement any technique to cope with cache leakage.

MEE attacks: The protection level of common attacks like Beast [25] and Lucky 13 [27] varies among different libraries. For example, we know that `OpenSSL` has a real constant time implementation to avoid padding leakage, whereas `CyaSSL` or `GnuTLS` do not [26, 27].

RSA: RSA is the most widely used public key cryptographic algorithm. Yarom et.al [51] showed that GNUTLS was not well protected to avoid cache leakage. Although the fix was addressed in the most recent version, an attacker could still try to detect calls to a specific non patched version of the library.

Insecure encryption ciphers: While some cryptographic libraries such as `CyaSSL` do not support weak encryption algorithms like DES-CBC, others do, e.g. `OpenSSL`. If someone is using it, our detection technique gives an attacker a good opportunity to perform attacks against it.

The usage of cryptographic libraries is very typical in many processes performed by a virtual machine. For instance, Mozilla web browser uses NSS (which could be added to the detector) whereas Chrome browser uses `GnuTLS`. However when a downloading process in the command line is performed, `OpenSSL` is used as the cryptographic library. Other applications like Steam use `OpenSSL` as well. This means that the observer could easily detect one of those widely used libraries, and in consequence, take advantage of any weakness present on it. Furthermore, This *cross-detection* method applied to cryptographic libraries allows the observer to profile the usage of certain applications (in case that the application is not using any other shareable code), as the ones mentioned above.

4.1 Additional Dangers of Version Detection

Version detection goes beyond library detection, as its main goal is to distinguish between different versions of the same software family, such as a crypto library as `OpenSSL`. There are several well-known vulnerabilities in certain `OpenSSL` versions that can enable simple attacks if a malicious party becomes aware of an unpatched implementation. When used on the cloud, an adversary can use the proposed tool to detect such outdated versions running in co-resident VMs.

Thus, the knowledge of the version enables the adversary to choose the most powerful attack against a spcific library. Outdated versions of a specific cryptographic library are common, since each OS has its own default installation of a specific version. For example, Ubuntu 12.04 uses `OpenSSL 1.0.1` and Ubuntu 14.04 uses `OpenSSL 1.0.1f`.

We work with seven `OpenSSL` versions; `OpenSSL 0.9.7a`, `OpenSSL 0.9.8K`, `OpenSSL 1.0.0a`, `OpenSSL 1.0.1c`, `OpenSSL 1.0.1e`, `OpenSSL 1.0.1f`, `OpenSSL 1.0.1g`. In the following we mention the most popular bugs related to each one of the versions of the library analyzed [6]. These are flaws that directly affect the mentioned version, but of course new flaws in more recent versions can be applied to the most outdated ones:

`OpenSSL 0.9.7`: This is a most outdated version that should not be used under any circumstance. There are many attacks targeting this version, such as AES cache attacks, branch prediction attacks in RSA or attacks on PKCS [34].

`OpenSSL 0.9.8k`: Vulnerable to Denial of Service attacks, Kerberos crash attacks or flaws in the handling of CMS structures.

`OpenSSL 1.0.0a`: Vulnerable to Buffer Overrun attacks in TLS, vulnerable to modifications of stored session cache ciphersuites, DOS attacks due to GOST parameters, memory leakage due to failure of byte clearing in SSL3.0, vulnerable to Vaudenay's padding oracle attack [47], weaknesses in the PKCS code exploitable using Bleichenbacher's attack [18].

`OpenSSL 1.0.1c`: Vulnerable to Lucky 13 attack [27], DOS attack due to failure in OCSP response, DOS attack on AES-NI supporting platform.

`OpenSSL 1.0.1e`: Vulnerable to *Flush+Reload* ECDSA attack [51], and to Heartbleed attack [4].

`OpenSSL 1.0.1f`: Vulnerable to *Flush+Reload* ECDSA attack [51], and to Heartbleed attack [4].

`OpenSSL 1.0.1g`: Heartbleed fixed.

Heartbleed attack: The Heartbleed attack is a serious threat that was discovered in `OpenSSL`. The bug allows a malicious attacker to steal sensitive information used by SSL/TLS encryption. It compromises the secret keys used to identify the service providers. The bug was addressed in `OpenSSL 1.0.1g`, which means that previous versions of `OpenSSL` are still vulnerable to the attack. This compromises seriously those users who are still using an outdated version of the popular `OpenSSL` library. Our detection method allows an attacker to detect whether one of the vulnerable versions is being used.

5 Detection Method

Here we introduce our library detection method. We distinguish between two scenarios, for which the detection methods slightly differ:

- **Library Detection:** detecting whether a library is running in a co-located VM
- **Version Detection:** detecting whether a particular version of a library is running in a co-located VM.

The Library detection method works by exploiting information leaked through deduplication, detected by using the *Flush+Reload* technique. It is clear that each library has unique functions that are called when a SSL/TLS connection is performed. This gives rise for a library identifying process.

5.1 Detection Stages

The detector performs the *library detection* or *version detection* in several stages. A detailed description of the steps of the detection method is as follows:

Unique Function Identification: The detector identifies functions that are called when an SSL/TLS connection is established. The goal is to pick functions that

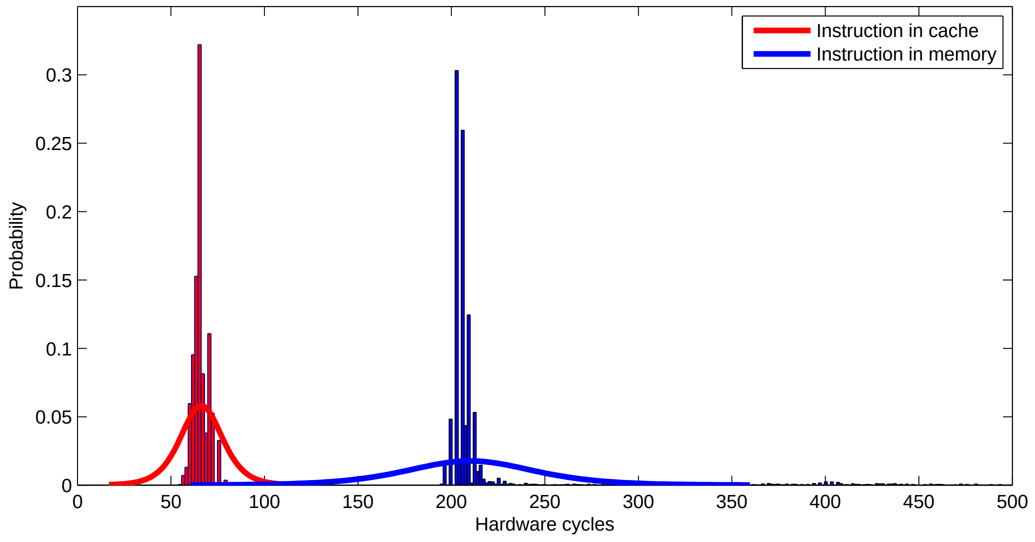

Fig. 2. Reload time in hardware cycles when a co-located VM is using the targeted instruction (red) and when it is not using the targeted instruction (blue) using KVM on an Intel XEON 2670

are unique to the library and therefore have a potential in being mapped to a unique hash during the deduplication process while at the same time marking the event we wish to be able to detect. For the cryptographic libraries we are targeting, we have preselected the identification functions as follows:

– **OpenSSL**: `SSL_CTX_new`. This function creates the context variable to start an SSL/TLS connection and is always called before a SSL/TLS connection.
– **PolarSSL**: `ssl_set_authmode`. This function is called to select the preferred authentication mode in an SSL/TLS connection performed by `PolarSSL` at the beginning of an SSL/TLS connection.
– **GNUTLS**: `gnutls_global_init`. Function for the initialization of `GnuTLS` library variables and error strings that is called before the beginning of each SSL/TLS connection.
– **CyaSSL**: `CyaSSL_new`. Every SSL/TLS connection is associated with a `CyaSSL` object. This object is created by the `CyaSSL_new` function, and therefore has to be called prior to each `CyaSSL` SSL/TLS connection.
– **MatrixSSL**: `matrixSslOpen`. Opening library function performed by `MatrixSSL` prior to any other SSL/TLS functions, making it suitable for our detection method.

Offset Calculation: The detector has to calculate the offset of these functions with respect to the beginning of the library, since the ASLR is going to move the user address space randomly. Functions such as `dlopen` allow to recover the starting virtual address of the monitored library. By further obtaining the virtual address of the monitored function, the attacker can calculate the difference between both addresses. This difference is constant for different virtual address spaces. Another possibility for the attacker is to disable the ASLR in his own OS, and refer directly to the address corresponding to the targeted function.

***Flush+Reload* as Detection Method**: The attacker will use *Flush+Reload* to detect whether a function that belongs to a library has been called or not. If any other co-located VM uses one of the functions that the attacker is trying to detect, this will be present in the last level of cache and therefore the reload time is going to be smaller. Hence the attacker can conclude that the library which the function belongs to has been called. If it was not accessed, the function will reside in main memory, having a bigger reload time. Figure 2 visualizes this last statement via an experiment outcome. The figure shows the reload times of a certain function when monitored by *Flush+Reload* technique. The reload time when a co-located VM is calling it constantly (red bars) differs significantly from the case where it is not called (blue bars). The difference is quite substantial, and if we set the threshold in a correct value the amount of noise is very small.

However, the detection method is not perfect. Therefore, there are some accesses that are not going to be detected:

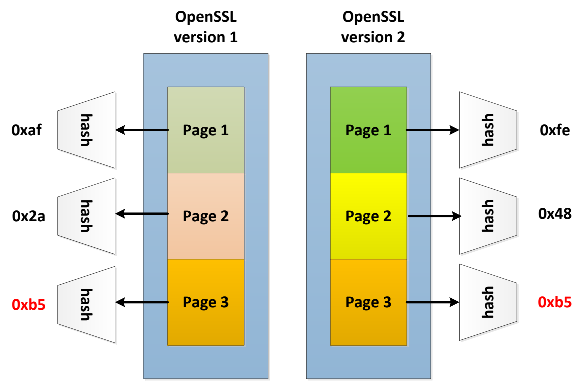

Fig. 3. KSM when two different versions have different pages, but the targeted page is the same

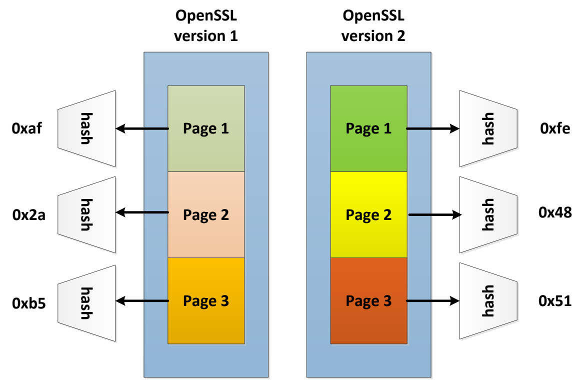

Fig. 4. KSM when two different versions have different pages, and the targeted page is different

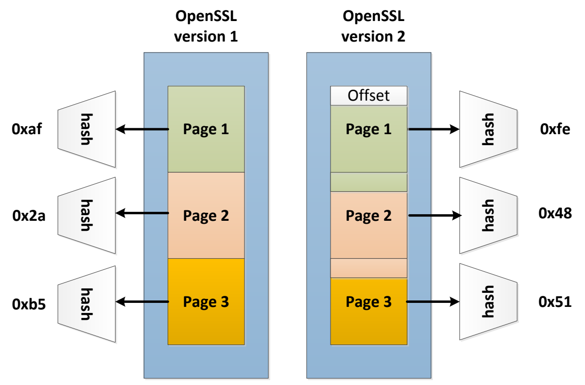

Fig. 5. KSM when an offset introduced by a modification in the library causes differences on the hash operation.

– Victim access occurs before the reload stage, and they do not overlap. In this case the access is going to be detected by the *Flush+Reload* technique.
– Victim access occurs after the reload stage, and they do not overlap. In this case the access is not going to be detected by the *Flush+Reload* technique.
– Victim access and *Flush+Reload* stage overlap. If the victim access was done slightly before the reload stage, this access will be noticeable. However, if the access was made slightly after the reload stage, it will not be detected.
– Some other process evicts the access prior to the reload stage, and therefore this is not going to be noticeable by the *Flush+Reload* technique.

5.2 Avoiding Wrong Version Detections

The detection is performed by monitoring unique functions associated to a library/version. These are easy to find across libraries since OpenSSL does not use the same function as any other library. However, this is not true anymore in the version detection scenario. The targeted functions in each version can experience three different situation:

Different Versions, Same Page: Figure 3 shows the case where two different OpenSSL versions are present and the targeted page (in which resides our targeted function) is the same for both of them. Since the memory is divided into pages of 4 KBytes and KSM works computing hashes at the page level, both pages will create the same hash and they would be merged. Therefore, an attacker would misspredict if the call was made by OpenSSL version 1 or OpenSSL version 2. In this scenario, the best route to take is to target another function that is not same in both libraries.

Different Versions, Different Functions: If the targeted functions to detect are different even though previous pages are the same, the attacker has no risk on misspredicting the versions, since KSM will never merge two different pages. This is the case we present in Figure 4

Different Versions, Same Page, Different Offset: In another situation the functions are the same in both versions of the library, but since a modification has been done in some other previous page in the library, the offset will not be the same anymore. Therefore the page address would be different and the result of the hash operation applied to the pages will be different. In this case the attacker would still be able to recognize different versions even though they have the same function. This is the situation presented in Figure 5.

We want to use a function that is called always in an SSL/TLS connections and that establishes a relationship with one single specific library. In the cases that we analyze, we keep using `SSL_CTX_new` because it meets both requirements. We already discussed the first requirement above. To test if we satisfy the second requirement, we analyze the function across all the libraries analyzed by our detector, observing different outcomes. This function changes between some of the versions, e.g. `OpenSSL 0.9.8k` and `OpenSSL 1.0.1a`, and does not change between some other versions, e.g. `OpenSSL 1.0.1e` and `OpenSSL 1.0.1f`. However the offset with respect to a page boundary is different among all of them, making the digest from the hash operation different for all of them. Therefore there is no risk that the hypervisor will merge pages from different library versions.

5.3 IP Address Recovery

After determining the library running in the co-located VM, the attacker runs the TLS handshake detector for that specific library and starts the IP address recovery step. To recover the IP address of the co-located VM, the attacker starts sending TLS communication requests to all IP addresses starting with her local subnet and trying wider range of addresses until a handshake is detected. When she triggers the co-located target VM, the specific library detector that she ran beforehand detects the TLS handshake. Only by observing the detector output and the IP scanner process, she can then pinpoint and recover the IP address at the time of a detection hit. This step is scripted and requires varying times depending on the detected library TLS setup time. In the worst case with the slowest library, it takes less than 15 seconds to scan 255 different IP addresses for a match.

6 Experiment Setup and Results

Our measurement setup mimics the cloud setup found in commercial CSPs. All experiments are performed on a machine featuring an eight core Intel Xeon e5-2670 v2 CPU, running at 2.5 GHz. This CPU is also commonly used by the `m3` instances of Amazon EC2. It features 32 KBytes of private L1 cache and 262 KBytes of private L2 cache per core, as well as 20 MBytes of L3 cache shared among the cores. KVM is used as our cloud hypervisor [10], whereas Ubuntu 12.04 is used for all the guest OSs. KVM's KSM feature is enabled for all experiments, scanning 100 pages each 200 milliseconds (the default values). Virtual machine manager is used to provide a graphic interface for the Virtual Machines.

In our evaluation setup, we create and use three virtual machines. The first one, VM1, acts as a *user* performing SSL/TLS connections with a library of its choice. The second VM, VM2, acts as *detector*: VM2 aims at detecting the library and specific version used by VM1. The last one, VM3, is used to simulate regular user load and therefore add various levels of noise depending on the **scenario**:

– **Noise-free:** In the first scenario, the victim, VM1, establishes SSL connections, the detector, VM2 executes our script to detect the library/version being called by the victim. No additional noise coming from a different VM is added, i.e. VM3 is idle.
– **Web browsing:** In the second scenario, again the victim VM is performing SSL connections while the detector VM tries to detect which library/version he is using. However, in this case a third VM is performing web-browsing operations at the same time. It will run a script that opens a web page each 5 seconds.
– **File sharing:** The third scenario is similar to the second one. In this case, while the victim is running SSL/TLS operations and the detector VM tries to run the detection script, a third Virtual Machine is going to be running a script automatically downloads a 6 KB PDF file every 5 seconds.
– **Media streaming:** In our last case, we want a scenario in which the amount of CPU load introduced by the noise adder VM is significantly higher than in the previous cases. In this scenario, while the detector VM and victim VM are executing *Flush+Reload* and SSL/TLS connections respectively, at the same time a third VM continuously streams a movie. To model this situation, we used Netflix as our media streaming software. In particular, we installed the Netflix-desktop version for Unix. Note that the content streamed via Netflix is Digital Rights Management (DRM) protected, therefore encrypted. For encryption, Netflix uses AES-128 in counter mode. The cryptographic operation as well as the media decoding process create significant background noise.

We characterize our noise scenarios in terms of 3 parameters; the additional CPU load that is observed in

Table 1. Noise parameters in different scenarios

Noise Scenario	CPU load	Network Traffic (tx-rx)	Cache usage [references/min]
Noise free	0%	0	0
Web browsing	20-25%	3.67K-21K	115×10^6
File sharing	20-25%	600-1K	12.4×10^6
Streaming	95%	82K-5.8M	$2,300 \times 10^6$

the hypervisor due to this operations, the network traffic created by these operations and the number of cache references experiences in one minute in each one of the scenarios. Table 1 summarizes the observed values for each of the characteristics analyzed for the scenarios under consideration. We used the Linux tool `top` to observe the CPU load increase, and `jnettop` to observe the network traffic value `kn` bits per second, and finally we use `perf` to calculate the number of cache references made in each scenario in one minute. We observed that in terms of CPU load, both the web browsing scenario and the file downloading scenario are similar with an increase of 25%. As expected, the media streaming scenario creates more CPU load; almost 4 times more than the previous two scenarios with an increase of around 95%. The situation changes a little in terms of network traffic and cache references, with the file downloading scenario with slightly less noise. We observe an increase of these two parameters in the web-browsing scenario. Again for the media streaming scenario, we observe a substantial increase in both parameters, making us believe that the detector will decrease its efficiency under this scenario.

In the experiments, we followed the approach of single-hit measurements meaning that we calculated the probability of successfully detecting a *single* function call at any time. It is important to note that the scanning process for all the cryptographic libraries used in the experiments takes only 5 seconds. Therefore, an adversary can easily amplify the detection rate of target library execution, even under heavy noise scenarios by scanning multiple times either continuously or in short intervals.

In our case the threshold to distinguish between accessed (function resides in the cache) and not accessed (function resides in the memory) functions is 190 cycles. As it can be seen in Figure 2, this threshold is based on experimental measurements and is sufficient to distinguish the two Gaussian distributions. Note that this is a hardware specific threshold and will have to be tuned if the experiments are performed on a different machine. The threshold value is easily obtained by running the required library function on the target platform and measuring the execution time using RDTSC(P) for both memory and cache access. In order to ascertain that the function code resides in the memory and the obtained time belongs to the memory access, CLFLUSH instruction is used to flush the code from all levels of cache.

6.1 Library Detection

To determine the efficacy of the library detection method we surveyed five cryptographic libraries, i.e. `OpenSSL 1.0.1 (OS)`, `GnuTLS 26 2.12.14 (OS)`, `MatrixSSL 3.6.1`, `CyaSSL3.0.0 and PolarSSL 1.3.7`. All the libraries are compiled as shared libraries. For our purposes we are going to suppose that one user runs a SSL/TLS connection using one of these five libraries, and a second user has to detect which one was run. We use typical functions that are used in a regular SSL/TLS connections (such as library initialization or context creation functions). We run a script that randomly chooses on of the libraries mentioned above and performs SSL/TLS connections using the following tools:

- **OpenSSL**: `s_client` and `s_server` tools provided in the `OpenSSL` library.
- **PolarSSL**: `ssl_client1` and `ssl_server` programs provided by `PolarSSL` to perform the SSL/TLS connection.
- **GNUTLS**: anonymus authentication server and anonymus authentication client examples provided by `GnuTLS` in [8, 9] for our test SSL connection.
- **CyaSSL**: client and server examples provided by `CyaSSL` to perform the SSL/TLS connections.
- **MatrixSSL**: client and server apps provided by `MatrixSSL` that establish a SSL connection between them.

We recorded a total of 100 calls per library. Results are presented in Table 2. The success rate represents the percentage of correctly detected libraries. Incorrectly detected libraries, i.e. false positives, are presented in the last row. The first thing to notice here is that we have an exceptionally low false positive rate, ranging from 0 false positives observed in the noise-free scenario to 2 false positives in both the third and forth noisy scenarios out of 500 calls. In the best case (noise-free scenario) the success rate ranges between 70% and 90% depending on the library. Furthermore, we observe that for

Table 2. Successful detection rate in library detection in all four noise scenarios.

Library	Detection success rate per scenario			
	Noise free	Web browsing	File sharing	Media streaming
CyaSSL	90%	85%	85%	50%
OpenSSL	77%	76%	79%	48%
MatrixSSL	71%	72%	76%	41%
PolarSSL	91%	88%	84%	50%
GnuTLS	83%	91%	86%	51%
False positives	0	0.2%	0.4%	0.4%

Table 3. Success detection rate in OpenSSL version detection in all four noise scenarios.

Library	Detection success rate per scenario			
	Noise free	Web browsing	File sharing	Media streaming
OpenSSL 0.9.7a	79%	79%	78%	34%
OpenSSL 0.9.8k	85%	84%	87%	50%
OpenSSL 1.0.0a	83%	84%	82%	42%
OpenSSL 1.0.1c	85%	86%	85%	41%
OpenSSL 1.0.1e	80%	86%	84%	44%
OpenSSL 1.0.1f	76%	83%	80%	44%
OpenSSL 1.0.1g	82%	84%	90%	38%
False positives	0.14%	0.29%	0.29%	0.14%

Table 4. IP detection rate results.

Library	Number of Connections	Scan Time for 255 IPs	Average Detection Hits
CyaSSL	1 tries per IP	5.57 seconds in average	3.5
OpenSSL	3 tries per IP	6.86 seconds in average	0.77
MatrixSSL	1 tries per IP	3.4 seconds in average	2.66
PolarSSL	3 tries per IP	7.33 seconds in average	1.54
GnuTLS	3 tries per IP	9.7 seconds in average	1.31

the low noise scenarios these results do not change significantly, meaning that this type of noise would not affect the results. Finally we observe that when the heavy load scenario is applied, we do not observe a significant increase in the wrong predictions, but we observe a decrease in the success rate of the libraries. These go down from almost 90% to 50% in the best cases, and from 70% to 41% in the worst cases. This means that even when the target is residing in a physical machine with heavy load, the detector can still detect with 50% accuracy.

One important fact that has to be mentioned here is that since we are using the OpenSSL version provided by the OS in the download script noise scenario, we are detecting some additional calls of OpenSSL from the downloading step. However since these calls are made by another VM and could be considered both correct or incorrect predictions, we did not include them in the results.

6.2 OpenSSL Version Detection

The second scenario is where we have a version detection tool. We used 7 different OpenSSL versions: OpenSSL 0.9.7a, OpenSSL 0.9.8k, OpenSSL 1.0.0a, OpenSSL 1.0.1c, OpenSSL 1.0.1e, OpenSSL 1.0.1f (previous to heartbeat fix) and OpenSSL 1.0.1g (heartbeat fixed). We have to be more specific here, since we have to look for functions that are different across libraries, because otherwise KSM would merge them even if they are from different libraries. However in all the libraries analyzed the offset of the function that is being tested (SSL_CTX_new) with respect to the beginning of a page is different, therefore KSM will never merge them, even when the function is the same. SSL_CTX_new is a function that is always called in a new SSL connection. We use the applications `s_server` and `s_client` that `OpenSSL` provides for testing SSL connections. We run a script that randomly chooses one of the above mentioned versions of `OpenSSL` and performs a TLS connection.

Again we record 100 calls of each one of the version, 700 in total, in each one of the scenarios. Again we are going to run 4 types of experiments: noise-free, web-browsing, download noise and heavy Netflix load. Results are presented in Tables 3. We see a similar pattern to the one we saw in the library detection. For noise-free and low noise scenarios we observe that the success probability varies from 76% to 90%. We have to remark that in this experiments we are not using the OS provided version of `OpenSSL` (plain 1.0.1) since we did it in the previous tests and therefore we do not detect the file downloads. One could always include this version to the detectors and still be able to detect it. Another observation is that again, the wrong predictions are fairly low, with 2 wrong predictions in 700 library calls in the worst case. Finally we observe that when heavy load is present in the server, the success rate decreases to 50% in the best case and to 34% in the worst case. This shows again that even though heavy load decreases the success rate, one could still detect one library call out of 3 in the worst case.

6.3 IP Detection

For this part of the experiment, we used detection functions that run when a TLS communication handshake is triggered in each particular library. After the initial library detection stage, we ran IP detector script and the TLS handshake detector to discover the co-located target VM's IP address. To be able to scan wide range of IP addresses in short time, we used the timeout command with the TLS client handshake process to eliminate IP addresses with no active TLS servers. We should note that this timeout value had to be short enough to allow fast scanning of large group of IP addresses but also provide enough time to allow the TLS client to run necessary pre-connection processes. To meet these two criteria, we tried different timeout values ranging from 0.001 milliseconds to 1 second for different libraries and determined that 0.01 milliseconds was optimal.

As for TLS Client, we experimented with different TLS clients provided with libraries and determined that the OpenSSL client was the fastest one amongst the inspected libraries. Therefore for the IP detection stage we used OpenSSL TLS client to trigger TLS handshake for all libraries. Table 4 shows the time it takes to scan 255 IP addresses in the subnet to discover co-located victim VM as well as the average detection hits from the co-located attacker VM during the IP detection stage. As seen from the table, for some libraries, namely OpenSSL, PolarSSL and GnuTLS, 3 TLS connection attempts per IP is used while for CyaSSL and MatrixSSL only 1 per IP is used. This is done in order to increase the detection rate while keeping the average scan time within acceptable limits. As the results show, all libraries except the OpenSSL have average detection hits over 1, meaning that they are always detected when the TLS handshake is triggered. As for the OpenSSL, even when 3 connections per IP are established, the detection rate is 0.77 in average and cannot be further increased by repeated measurements in acceptable scan time limits. We believe that this is due to the fact that OpenSSL handles TLS handshake faster than the detector have a chance to detect the handshake. The fact that OpenSSL TLS client was fastest to establish connections in our test and the scan time for 255 IPs was fastest for OpenSSL supports this reasoning.

7 Preventing Cross-VM Code Detection

We propose methods to mitigate cache leakage, thereby rendering our detection technique impotent.

- **Avoiding `clflush` usage**: Our detection method is based on the detector's ability of flushing specific memory lines from the cache with the `clflush` command. Prohibiting the usage of the `clflush` command would prevent the attacker from implementing the *Flush+Reload* attack. Note that disabling the clflush instruction will disrupt memory coherence in devices where memory coherence is not supported. Also note that, `clflush`-like instructions can still be substituted by cache priming techniques, as in [30].
- **Disabling Deduplication**: Disabling deduplication prevents the flush and reload based detection of executed code. Even partial disabling, e.g. by marking cryptographic libraries (or any critical software) to be excluded from deduplication, can prevent the library detection with minimal effect on performance. The main disadvantage of this countermeasure is the lack of memory usage optimiza-

tion, especially in multi-tenant systems. Keep in mind that with deduplication, it is shown to be possible to run over 50 Windows XP guest VMs on server with 16 GBs RAM [5]. Also note that, other spy processes like `Prime and Probe` may still succeed even when deduplication is turned off.

- **Dedicated hosting**: Public clouds like Amazon EC2 can provide customers with dedicated hosts. In this scenario no cache side-channel attacks can be implemented, since the attacker cannot co-locate with a victim anymore.
- **Cache Partitioning**: As suggested in [49], partitioning the cache is a hardware solution that would mitigate any kind of cache side channel attack. If the attacker and the victim have associated different portions in the shared level of cache (effectively creating *private* caches), no cache-based attacks are possible. Therefore associating some portions of the cache to some VMs/processes, even when deduplication is enabled, would avoid any cache leakage between VMs/processes. The downside of cache partitioning is that the cache utilization associated to each process decreases significantly, resulting in serious performance penalties.
- **Randomizing Cache Loads**: Another possibility also suggested in [49] is to add a random offset when the CPU fetches data to the cache. In this way, the physical memory would only have one copy of a shared page, but it would add a random offset when being loaded into the cache. This random offset is private for each of the processes/VMs. Therefore, an attacker would have to know the private offset of the victim's process/VM to be able to access the same data/set.
- **Diversifying the execution code**: One possibility to mitigate cache side-channel attacks is to create different and unique program traces (that perform identical computations) for different executions. This countermeasure, proposed in [24], will prevent the *Flush+Reload* technique since the specific location of the function that the attacker wants to monitor would be different for different users (and thus, libraries would never be deduplicated).
- **Degrading the granularity of timers**: As cache timing side channel attacks base their procedure on the accuracy of `rdtscp`-like timers, an easy solution that cloud hypervisors can adopt is to eliminate the access to fine grain timers from guest VMs. Alternatively, as stated in [46], fine grained timers could introduce a certain amount of noise so that cache side channel attacks are no longer applicable.

8 Conclusion

In this work, we presented a detection method to identify the execution of pieces of software on a target's VM across co-located VMs. While the technique is generic and applies to cross-VM settings where deduplication is enabled, our experiments focused specifically on cryptographic libraries. We believe that this is a highly relevant use case for the detection method, since it enables an attacker to covertly carry out a discovery phase with high precision and great speed. We demonstrated the viability of the detector by identifying the cryptographic library and the particular version used by a target. Our work shows for the first time that identifying a specific library version being used by a co-located tenant is possible. This enables an attacker to focus on the most viable vulnerability. One clear example is the Heartbleed bug, which was not fixed until `OpenSSL 1.0.1g` and allows an attacker the extraction of private information. We presented experiment results on OpenSSL versions under various noise scenarios. We observed that in low-noise scenarios the detection rate is up to 90%, whereas in heavy load scenarios the detection rate reaches up to 50%. Nevertheless we would like to emphasize that even in the worst case scenario with heavy load, the attacker gains the knowledge about the used library after two or three library calls.

9 Acknowledgments

This work is supported by the National Science Foundation, under grant CNS-1318919 and CNS-1314770. We would like to thank the anonymous reviewers of PETS 2015 for their helpful comments.

References

[1] Amazon AWS: 3.8 billion revenue in 2013. https://readwrite.com/2013/01/14/amazon-web-services-can-it-win-the-enterprise.
[2] Analyzing shared memory opportunities in different workloads. http://os.itec.kit.edu/downloads/sa_2011_groeninger-thorsten_shared-memory-opportunities.pdf.
[3] The dropbox blog. https://blog.dropbox.com/2013/07/dbx/.
[4] Heartbleed bug. http://heartbleed.com/.
[5] Kernel samepage merging. http://kernelnewbies.org/Linux_2_6_32#head-d3f32e41df508090810388a57efce73f52660ccb/.

[6] OpenSSL vulnerabilities. https://www.openssl.org/news/vulnerabilities.html.

[7] CyaSSL: Embedded SSL library wolfSSL. http://www.wolfssl.com/yaSSL/Home.html, May 2014.

[8] GnuTLS client examples. http://www.gnutls.org/manual/html_node/Client-examples.html, April 2014.

[9] GnuTLS server examples. http://www.gnutls.org/manual/html_node/Server-examples.html, April 2014.

[10] Kernel based virtual machine. http://www.linux-kvm.org/page/Main_Page, April 2014.

[11] MatrixSSL: Open source embedded SSL. http://www.matrixssl.org/, May 2014.

[12] AciİÇmez, O. Yet another microarchitectural attack:: Exploiting i-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture* (New York, NY, USA, 2007), CSAW '07, ACM, pp. 11–18.

[13] AciİÇmez, O., Gueron, S., and Seifert, J.-P. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. Cryptology ePrint Archive, Report 2007/039, 2007. http://eprint.iacr.org/2006/351.pdf.

[14] AciİÇmez, O., Koç, C. K., and Seifert, J.-P. On the power of simple branch prediction analysis. *IACR Cryptology ePrint Archive 2006* (2006), 351.

[15] AciİÇmez, O., Koç, C. K., and Seifert, J.-P. Predicting secret keys via branch prediction. In *CT-RSA* (2007), M. Abe, Ed., vol. 4377 of *Lecture Notes in Computer Science*, Springer, pp. 225–242.

[16] Arcangeli, A., Eidus, I., and Wright, C. Increasing memory density by using KSM. In *Proceedings of the linux symposium* (2009), pp. 19–28.

[17] Bernstein, D. J. Cache-timing attacks on AES, 2004. URL: http://cr.yp.to/papers.html#cachetiming.

[18] Bleichenbacher, D. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS1. Springer-Verlag, pp. 1–12.

[19] Bonneau, J. Robust final-round cache-trace attacks against AES.

[20] Bonneau, J., and Mironov, I. Cache-collision timing attacks against AES. In *Cryptographic Hardware and Embedded Systems—CHES 2006* (2006), vol. 4249 of *Springer LNCS*, Springer, pp. 201–215.

[21] Brumley, B. B., and Tuveri, N. Remote timing attacks are still practical. In *Computer Security–ESORICS 2011*. Springer, 2011, pp. 355–371.

[22] Brumley, D., and Boneh, D. Remote timing attacks are practical. *Computer Networks 48*, 5 (2005), 701–716.

[23] CBC news. Heartbleed bug: 900 SINs stolen from Revenue Canada. http://www.cbc.ca/news/business/heartbleed-bug-rcmp-asked-revenue-canada-to-delay-news-of-sin-thefts-1.2609192l, April 2014.

[24] Crane, S., Homescu, A., Brunthaler, S., Larsen, P., and Franz, M. Thwarting cache side-channel attacks through dynamic software diversity.

[25] Dan Goodin. Hackers break SSL encryption used by millions of sites. http://www.theregister.co.uk/2011/09/19/beast_exploits_paypal_ssl/, 2011.

[26] Duong, T., and Rizzo, J. Here come the XOR ninjas.

[27] Fardan, N. J. A., and Paterson, K. G. Lucky Thirteen: Breaking the TLS and DTLS record protocols. In *Security and Privacy (SP), 2013 IEEE Symposium on* (May 2013), pp. 526–540.

[28] Gullasch, D., Bangerter, E., and Krenn, S. Cache games – bringing access-based cache attacks on AES to practice. *IEEE Symposium on Security and Privacy 0* (2011), 490–505.

[29] Hu, W.-M. Lattice scheduling and covert channels. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 1992), SP '92, IEEE Computer Society, pp. 52–.

[30] Irazoqui, G., Eisenbarth, T., and Sunar, B. Jackpot stealing information from large caches via huge pages. Cryptology ePrint Archive, Report 2014/970, 2014. http://eprint.iacr.org/.

[31] Irazoqui, G., İncİ, M. S., Eisenbarth, T., and Sunar, B. Wait a Minute! A fast, Cross-VM Attack on AES. In *Research in Attacks, Intrusions and Defenses*, A. Stavrou, H. Bos, and G. Portokalidis, Eds., vol. 8688 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 299–319.

[32] Jones, M. T. Anatomy of Linux kernel shared memory. http://www.ibm.com/developerworks/linux/library/l-kernel-shared-memory/l-kernel-shared-memory-pdf.pdf/, April 2010.

[33] Kelsey, J., Schneier, B., Wagner, D., and Hall, C. Side Channel Cryptanalysis of Product Ciphers. *J. Comput. Secur. 8*, 2,3 (Aug. 2000), 141–158.

[34] Klíma, V., Pokorny, O., and Rosa, T. Attacking RSA-based sessions in SSL/TLS. In *in Proc. of Cryptographic Hardware and Embedded Systems (CHES), 2003* (2003), Springer, pp. 426–440.

[35] Nikos Mavrogiannopoulos and Simon Josefsson. GnuTLS: The GnuTLS transport layer security library. May 2014.

[36] Osvik, D. A., Shamir, A., and Tromer, E. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology* (Berlin, Heidelberg, 2006), CT-RSA'06, Springer-Verlag, pp. 1–20.

[37] Page, D. Theoretical use of cache memory as a cryptanalytic side-channel, 2002.

[38] PolarSSL. PolarSSL: Straightforward,secure communication. www.polarssl.org.

[39] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. 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* (New York, NY, USA, 2009), CCS '09, ACM, pp. 199–212.

[40] Suzaki, K., Iijima, K., Yagi, T., and Artho, C. Memory deduplication as a threat to the guest OS. In *Proceedings of the Fourth European Workshop on System Security* (2011), ACM, p. 1.

[41] Suzaki, K., Iijima, K., Yagi, T., and Artho, C. Software side channel attack on memory deduplication. *SOSP POSTER* (2011).

[42] Suzaki, K., Iijima, K., Yagi, T., and Artho, C. Effects of memory randomization, sanitization and page cache on memory deduplication.

[43] The Guardian. More than 300k systems 'still vulnerable' to Heartbleed attacks. http://www.theguardian.com/

technology/2014/jun/23/heartbleed-attacks-vulnerable-openssl, July 2014.

[44] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. www.openssl.org, April 2003.

[45] Tsunoo, Y., Saito, T., Suzaki, T., and Shigeri, M. Cryptanalysis of DES implemented on computers with cache. In *Proc. of CHES 2003, Springer LNCS* (2003), Springer-Verlag, pp. 62–76.

[46] Vattikonda, B. C., Das, S., and Shacham, H. Eliminating fine grained timers in xen.

[47] Vaudenay, S. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS. In *Proceedings of In Advances in Cryptology - EUROCRYPT'02* (2002), Springer-Verlag, pp. 534–546.

[48] Waldspurger, C. A. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 181–194.

[49] Wang, Z., and Lee, R. B. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2007), ISCA '07, ACM, pp. 494–505.

[50] Yarom, Y., and Benger, N. Recovering OpenSSL ECDSA nonces using the flush+reload cache side-channel attack. Cryptology ePrint Archive, Report 2014/140, 2014. https://eprint.iacr.org/2014/140.pdf.

[51] Yarom, Y., and Falkner, K. E. Flush+reload: a high resolution, low noise, L3 cache side-channel attack. *IACR Cryptology ePrint Archive 2013* (2013), 448.

[52] Zhang, Y., Juels, A., Oprea, A., and Reiter, M. K. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2011), SP '11, IEEE Computer Society, pp. 313–328.

[53] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 305–316.

[54] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 990–1003.