Raj and Dharanipragada *Journal of Cloud Computing: Advances, Systems and Applications* (2017) 6:28
DOI 10.1186/s13677-017-0101-4

Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH Open Access

Keep the PokerFace on! Thwarting cache side channel attacks by memory bus monitoring and cache obfuscation

Arun Raj* and Janakiram Dharanipragada

Abstract

Cloud instances are vulnerable to cross-core, cross-VM attacks against the shared, inclusive last-level cache. Automated cache template attacks, in particular, are very powerful as the vulnerabilities do not need to be manually identified. Such attacks can be devised using both the Prime+Probe and the Flush+Reload techniques. In this paper, we present PokerFace, a novel method to identify and mitigate such attacks. This approach allows us to identify suspicious cache accesses automatically, without prior knowledge about the system or access to hardware metrics. PokerFace consists of two components, Poker and Face. Poker executes a memory bus benchmark to measure the available bus bandwidth and derive information about cache accesses and possible side channel attacks. Our experiments with cache attacks show a reduction of up to 14% in the memory bandwidth during the attack. When an attack is detected, Poker triggers Face which performs cache obfuscation. We demonstrate the effectiveness of our approach against keypress logging attacks. We also test it against generic Prime+Probe and Flush+Reload attacks and show that it is practically useful against a variety of cache timing attacks. PokerFace incurs modest overheads (< 8%) and moreover, does not require support from the cloud provider or changes to the hypervisor. Unlike previously proposed techniques, it can be implemented by cloud subscribers.

Keywords: Cloud security, Side channel attacks, Cache obfuscation

Introduction

Cloud services provide virtualized resources to the end users on a pay-per-use model. Infrastructure-as-a-service vendors provide virtualized hardware by providing a separate virtual machine (VM) to each tenant. The high resource utilization achieved by sharing is fundamental to the economy of cloud providers: they co-host multiple VMs on the same hardware and rely on the underlying hypervisor to provide isolation and security, in addition to scheduling and sharing of system resources. Only the high-end instances are hosted on a dedicated hardware, e.g., D15 v2 on Microsoft Azure[1]. However, such instances are expensive and also lead to over provisioning for most use cases. While virtual machines might give the impression of isolation and dedicated resources, the dedicated virtual resources are mapped to shared physical resources. This results in potential for interference and side channel attacks [1].

Cache-based side channels have been used to attack cryptographic implementations [2, 3] on inclusive last-level caches. Ristenpart et al. [4] observed cache activity to get keystroke timing information on a system. However, these attacks were limited by their sophistication. Extensive knowledge about the victim algorithm or software is required to identify vulnerable memory accesses. In certain cases, even modifications to the source code is required [5] for manual execution of specific code fragments. To overcome these difficulties, a generic approach was proposed in the form of cache template attacks [6]. These attacks can automatically determine memory addresses which are accessed by a program depending on cryptographic keys or specific events. The authors also propose to use the cache template attacks as a system service to identify attacks, which can be mitigated by disabling page sharing or adding noise to the specific lines under attack.

*Correspondence: arun@cse.iitm.ac.in
Department of Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India

During our experiments on keypress logging, we observe that the profiling phase of the cache template attacks takes multiple hours to execute and also renders the system unusable due to the continuous stream of keypresses. Most cloud instances have larger shared caches and are rented on a per-need basis, often for short durations to handle spikes in the load. Hence, keeping in mind the performance and economy, we require a faster method to detect, and if possible, disrupt these attacks.

In this paper, we make use of memory bus monitoring and cache obfuscation [7] to present the PokerFace defense mechanism, a generic approach for detecting suspicious activity on the inclusive last-level cache and mitigating them by cache obfuscation. PokerFace is based on the following concepts about cache and cache attacks.

1. Cache attacks are based on flushing and reloading cache lines (or priming and probing cache sets, as the case may be) and measuring the timing information.
2. Cache attacks target the commonly used (shared) libraries. Hence, the flushes (or probes) will be followed by large number of cache misses.
3. Data and instructions will have to be fetched from memory, resulting in memory bus usage.
4. Knowledge about currently executing applications on the monitored VM and the exact duration of the attack can be used for switching the mitigation techniques like cache obfuscation on and off as needed.

Based on these observations, we demonstrate how to monitor the memory bus and detect cache side channel attacks, based on the constant measurement of memory bandwidth. We also provide a simple cache obfuscation technique to impede the attacks. We test these methods against the real-world attack scenario of keypress loggers and find them to be highly effective. We further test them against generic Prime+Probe and Flush+Reload attacks. The defense mechanism is triggered only when safety-critical applications are being executed. These set of applications can be edited any time by the user. Our approach can be used on any hardware and any operating system, as long as the last-level cache is inclusive and shared. We also show that our approach incurs a very low overhead when compared to existing techniques. We demonstrate our approach on Intel Xeon processors, which have an inclusive last-level cache and are commonly used by public cloud providers to host cloud instances. Our mechanism can also be extended to platform-as-a-service (PaaS) clouds.

The paper is organized as follows. We discuss the relevant background about cache attacks in "Background" section. We present the design and implementation of the PokerFace framework in "Design and implementation" section. We empirically evaluate our security framework against common cache attacks in "Evaluation" section. In "Related work" section, we explore existing related work. We discuss the relevance of our design choices along with alternative approaches in "Discussion: the case for guest-based solutions to cache attacks" section and finally conclude in "Conclusion" section.

Background

Virtualization

Xen [8] and KVM [9] are the most popular hypervisors used in cloud systems. Xen is used by Amazon EC2, Rackspace, etc., while KVM is the hypervisor choice for OpenStack and Google Compute Engine. Both Xen and KVM allow multiple VMs to be created on a physical host and rent them to customers. We perform our experiments on the KVM hypervisor. Hypervisors provide a certain level of isolation between virtual machines running on the same host. CPU cores can be statically pinned to VMs and memory can be partitioned so that the region allocated to one VM can not be accessed by other VMs. However, the last level cache (LLC), memory bus, secondary storage, network adapter, etc., are shared among all the co-hosted VMs. Modern processors have a sliced LLC with a different slice for each core, but all slices can be accessed by all cores, the design being motivated by data locality rather than isolation.

Cache attacks

Cache attacks are a type of side-channel attacks. They exploit the impact of cache memory on the execution time of algorithms. The first attacks propounded were theoretical in nature [10, 11]. In 2004, the first time-driven cache attack against AES was proposed by Bernstein [12]. Gullasch et al. [2] implemented a powerful attack on the L1 cache based on the fact that different processes can have shared pages loaded into the same cache sets.

Cache attacks are primarily of two types:

- Prime+Probe Attack: Here, the attacker occupies a specific cache set and monitors it to deduce when the victim accesses the set [1]. If the victim loads data into the set, the attacker's data will be flushed, resulting in cache misses. It does not require the sharing of memory pages between the victim and the attacker and are practical on cloud infrastructure. Oren et al. [13] showed that such an attack can be launched from within a browser running JavaScript code, using which the attacker can eavesdrop on mouse movements and other activities.
- Flush+Reload Attack: Yarom and Faulkner [3] proposed the Flush+Reload attack targeting the shared L3 cache. This attack relies on shared libraries between the attacker and victim programs. The

attacker uses the `clflush` instruction to constantly flush cache lines. After the victim accesses the shared memory, the attacker measures the time taken to access the same address. The access time reveals whether the victim used the library, thereby loading it into the cache or not. Memory deduplication is often disabled on infrastructure-as-a-service (IaaS) clouds, but Flush+Reload attack has been shown to be practical on platform-as-a-service (PaaS) clouds like DotCoud [14, 15].

These attacks are indigenous to inclusive caches, where flushing data from the last-level cache will flush them from the upper level caches as well.

Cache template attacks

Gruss et al. [6] presented cache template attacks on the shared, inclusive last-level cache. These attacks can exploit any cache vulnerability in any program executing on any operating system or hardware with shared memory enabled. Though they devise their attack on the Flush+Reload attack, the same concept can be utilized even with Prime+Probe attacks at the granularity of the cache set. Cache template attacks consist of two phases:

1. **Profiling phase:** The profiling phase determines the number of cache hits on a specific address for a specific event. This is performed for each address and each event in the target binary and the results stored in a cache template matrix. This is a highly exhaustive and intensive process.
2. **Exploitation phase:** In the exploitation phase, all the addresses in the cache template matrix are continuously monitored and cache hits are recorded. Events are detected by cross-referencing the information gathered with data in the matrix.

Both the phases are based on Flush+Reload and hence result in heavy cache activity (Prime+Probe attacks also have similar characteristics, which will be explained in detail in subsequent sections). These attacks are a generalized template for different kinds of cache attacks. The authors have used them to launch keypress logging attacks, attacks on GDK key remapping, OpenSSL AES T-Table attack, etc. In other words, they provide the generic method which is the key to launch any kind of cache-based side channel attack.

The relevance of the profiling phase in cloud

Many implementations [1, 3] discuss cache attacks in terms of 'monitored cache lines', 'generated eviction sets', etc., which suggest that these attacks have no or minimal profiling period. However, the attacker needs to know which cache lines or sets to monitor since the large size of the last-level cache makes it impractical to continuously monitor the whole cache and no prior knowledge is available on third party cloud instances. Knowledge about cache access patterns are unusable without information about what those specific cache lines or sets represent. For e.g., let us assume the address `0x40` corresponds to the keypress of letter *a*. Without this knowledge (which can be obtained only via profiling), the attacker can only ascertain that the victim is performing some activity and nothing more.

Detecting cache attacks

Most of the existing techniques for detecting side channel attacks rely on detecting the unique signatures of attacks using hardware performance counters [16]. The application behaviour is compared with pre-identified attack signatures, which requires an exhaustive set of signatures. Though the rate of false positives is low, there can be false negatives since it can be evaded using metamorphic code. Moreover, this can not be implemented by cloud users since hardware counters are unavailable, which serve as the basis for generating the signatures. Other techniques include anomaly-based detection, which flag any anomalous behaviour as an attack. They can potentially identify "zero-day" and any new attacks. However, since cache attacks resemble memory intensive benign applications, it is difficult to precisely model attack behaviour. This can lead to false positives, though there can never be any false negatives.

CloudRadar [17] utilizes both these mechanisms to identify side channel attacks by monitoring hardware performance counters on both the victim and attacker VMs. They propose CloudRadar as a value added service by the cloud provider, but to the best of our knowledge, such a service is not available in practice. Moreover, they require the victim to submit signatures of all security-critical cryptographic applications to the service.

Non-temporal instructions

When produced data is not immediately consumed, storing them in cache is detrimental to performance since they will evict other cache lines which might have an earlier temporal reference. Also, for large data structures, their sheer size might end up evicting their own elements, making caching ineffective. To avoid the eviction of data in such cases, non-temporal write operations are provided by the processors. The support for SSE (Streaming SIMD Extensions) [18] is found in Intel Pentium III and subsequent processors. Since the data is non-temporal, i.e., it will not be used anytime soon, it need not be cached and can be written directly to memory. Non-temporal instructions are represented by MOVNT* (MOV non-temporal) such as MOVNTI (double word), MOVNTQ (quad word), MOVNTPS (single-precision floating point),

etc. They avoid cache pollution but these writes result in higher memory bus usage for transferring the address and data bits.

Design and implementation

The system model for the attack and defense scenario is shown in Fig. 1. The attacker and victim VMs are hosted on the same hardware. The shared resources include the last-level cache and the memory bus. We introduce Poker and Face into these shared resources in order to detect and obstruct cache attacks.

Poker: anomaly-based detection of cache attacks

CloudRadar is a provider-centric mechanism which relies on hardware counters. It flags an attack if 1) the victim is executing an application with signature similar to cryptographic applications, and 2) the attacker shows anomalous behaviour at the same time. Poker does something similar, though with a lower resolution. Since Poker is running on the victim VM, it does not require complex techniques to ascertain if cryptographic code is being executed or not. It can simply obtain the information from the linux process tree. In essence, Poker flags an attack if

1. A security-critical application is running on the victim VM. The default set of applications contains AES, RSA, etc., and the user can add his own custom choices. This information is directly available from the process tree with no overhead.
2. An anomalous behaviour in the memory bus is observed. Since Poker has knowledge about the applications running on the victim, it can filter out any anomalies generated by the victim itself.

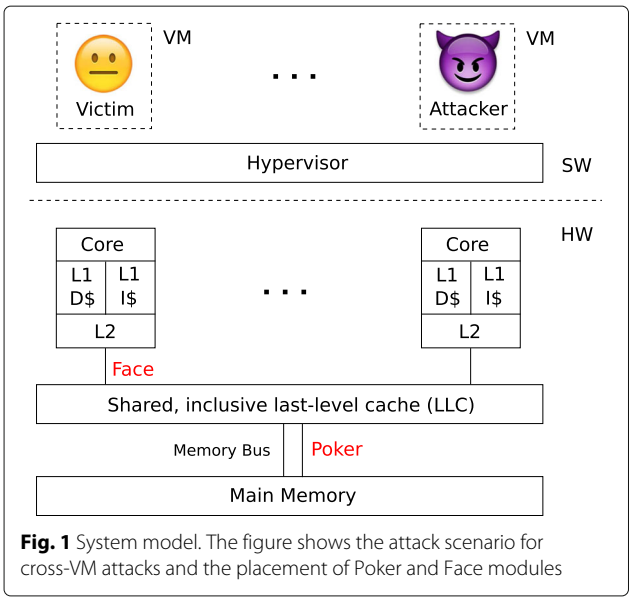

Fig. 1 System model. The figure shows the attack scenario for cross-VM attacks and the placement of Poker and Face modules

Implementation

Poker is essentially a memory bandwidth monitor. The memory bus connects the shared last-level cache with the physical memory and is used when there is a miss in the cache. Poker is developed in C and assembly language using SSE primitives. These streaming instructions avoid cache usage and perform non-temporal writes directly to the memory. The user interface is written in C for maintainability, but the actual SSE instructions are coded in assembly to assist hand coded optimizations and avoid compiler optimizations. The pseudocode of Poker is given in Listing 1.

Poker utilizes the XMM SSE double quad word instructions. MOVNTDQ moves a 16-byte aligned 64-bit quad word value from a 128-bit XMM register to a 128-bit memory address. Poker continuously writes 8 KB blocks to the memory and measures the bandwidth by calculating the time taken for the writes. (In other words, it continuously **pokes** around in the memory bus to measure the available bandwidth.) In a virtualized environment, the time measured for a memory write is liable to be inaccurate due to virtualization overheads. However, we are concerned with the relative decrease in the bandwidth (and hence, with the corresponding increase in write latency) and not the actual time taken by the operation. Even with virtualization overheads, we can estimate the memory bandwidth as viewed from within the virtual machine.

```
// poker.c
extern int poker(int blksize);

while (true) {
 poker(8K);
}

; poker.s
 poker:
          MOVNTDQ %XMM0 , 0(% R8)
          MOVNTDQ %XMM0 , 16(% R8)
          ...
          MOVNTDQ %XMM0 , 496(% R8)
          DECQ % RAX
          JNZ poker
```

Listing 1 *Poker* Pseudocode

Poker uses the gathered information to ascertain the state of the last-level cache. It is based on the following fundamental concepts:

A. The Memory Hierarchy: The last level-cache is connected to the main memory (RAM) via the memory bus. A miss in the last-level cache will force the usage of memory bus, in order to fetch the data or instructions from main memory.

B. The Nature of Cache Attacks: Cache attacks build upon the effect of cache memory on the runtime of algorithms. During Prime+Probe, the attacker repeatedly probes certain cache sets and checks for cache misses after the victim executes. Hence, cache misses will be higher when the victim executes the monitored code. During Flush+Reload, attackers repeatedly flush and reload the cache lines and measure the access times to ascertain if the victim has accessed the same lines. The attacks are targeted on frequently used shared libraries. This results in large number of cache misses on the victim after every flush. With multiple lines and sets being monitored, the memory bus usage is high. This similarity in the anomalous behaviour of Prime+Probe and Flush+Reload has been observed previously [17, 19].

We surmise that *during a cache attack, the load on the memory bus will be higher than usual and hence, available bandwidth will be lower*. We validate our claims by subsequent experiments. Since cache attacks have behaviour similar to applications like media streaming, anomaly-based detection will flag such applications also as attacks leading to false positives. However, an attack is flagged only when secure applications are being executed on the victim VM and anomalous behaviour is detected on other co-located VMs. Since security is of paramount importance during this period, even with false positives, the defense mechanism must be activated.

Face: opportune cache obfuscation

Cache obfuscation has been proposed as a technique for online mitigation of side channel attacks. Zhang et al. [7] used it on L1 and L2 caches between different iterations of the attack, with promising results and termed it cache cleansing. In [6], the authors propose the usage of cache template attacks as a system level service to constantly monitor the last-level cache. They further suggest that noise can be generated on specific address ranges when an anomaly or attack is observed. Such precision, however, comes at a price and is not always feasible. The complete cache needs to be profiled, which is a time-consuming process followed by constant monitoring of all the cache lines. This is highly unsuitable in cloud environments where the instances might be rented only for short periods, leaving insufficient time for profiling.

Since time is the limiting factor here, we focus on the *when* and not the where. Poker can detect a cache attack while it is under progress, but not the exact cache region (and thereby the shared library) being targeted. We follow a similar principle here as well: generate enough random noise to obfuscate the data while it is being gathered. (Always show a poker **face** to the attacker when you suspect him to be peeking.)

Face is a temporal cache cleanser, i.e., it generates random noise during specific intervals of time. An attacker will commonly target the standard libraries which are routinely utilized. We argue that the same libraries can be used to generate random cache events to obscure the attacker's view of the cache. As a proof of concept, we implement Face to generate a sequence of random alphanumeric keypresses, which access a series of different addresses in the cache lines occupied by the related shared library. We test this against keypress logging attacks on the GDK library and find it to be highly efficient. We believe that a more rigorous implementation using multiple events pertaining to different libraries can impede a variety of cache attacks.

The process is very lightweight in nature, but normally renders the system unusable due to the continuous stream on keypresses on screen. In order to isolate the process and restrict the resources used, we execute Face in a containerized, single core environment. Linux containers [20] are a lightweight alternative to conventional VMs with kernel namespace isolation and resource guarantees based on cgroups. We run Face on Docker [21], which is a popular containerization platform. This also makes Face portable, since it is packaged with all its dependencies and does not require any support from the underlying platform.

The addresses accessed by the GDK-based proof-of-concept implementation of Face are shown in Fig. 2 for two separate executions. We map the address ranges for the shared GDK library which handles keystrokes and represent them in a 180×60 grid (approximately 10,800 addresses in a 10 MB last-level cache). The cache regions are shown in the form of a heat map. White represents 0 hits. Blue, yellow, orange, red, gray and black portray progressively increasing number of cache hits. It can be observed that the noise is sufficiently spread across the cache regions which are likely to be used by the GDK library during keypress events. Some addresses are common to multiple keys and hence, frequently accessed. It can also be observed that the addresses which are more frequently accessed are different in the two cases, showing that the noise is adequately random and non-deterministic in nature.

Non-standard libraries

Cloud infrastructure is used for a variety of applications and the same instance need not necessarily run the same programs all the time. Moreover, detailed knowledge about the exact processes is not possessed by the attacker. An attack on a non-standard library like libhadoop (used by HDFS) is more likely to fail than that on a commonly used library like libgdk since keypress events always occur but HDFS might not be present on the instance at all.

It is primarily for this reason that cache attacks usually target commonly used libraries like GDK (keypresses), AES (encryption), etc. They are also more likely to leak

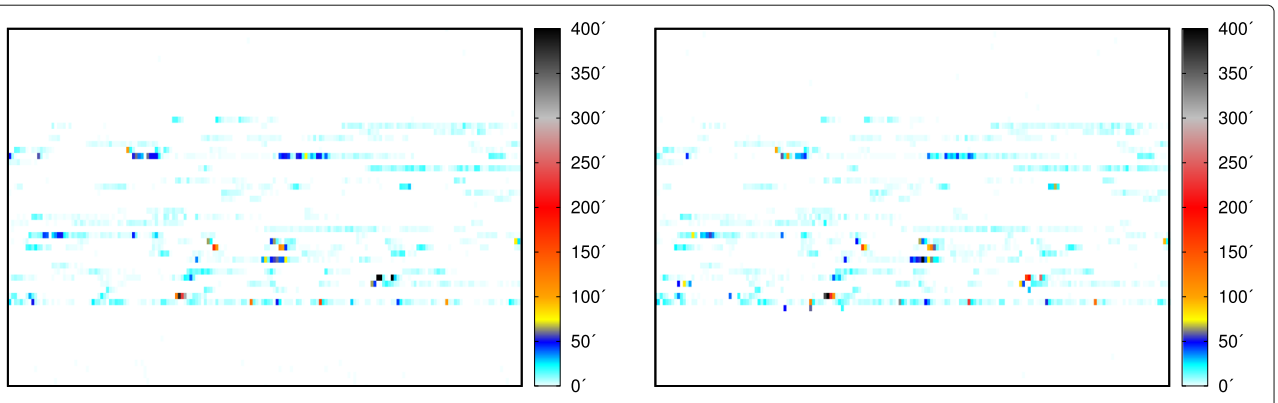

Fig. 2 Cache accesses made by Face during two different executions. The figure shows the cache hits made during cache obfuscation represented in the form of a heat map. They depict approximately 10,800 addresses in a 10MB last-level cache in a 180 x 60 grid. This is just to show that the hits are sufficiently random to make the obfuscation process effective

sensitive information like passwords. Our approach allows any library to be plugged into Face to generate cache noise. A safe approach would be to use a mix of standard libraries and specific non-standard libraries which are being used regularly on the instance at a given time.

Side effects of obfuscation

Cache obfuscation generates random noise in the last-level cache which, in turn, evicts the data already present in those cache lines. If the evicted data is needed again, it is fetched from the main memory which can result in moderate overheads. However, incorrect data is never supplied and data consistency is always maintained. In essence, Face behaves like any other cache intensive workload and since it is executed only when an attack is suspected by Poker, the effect is minimal. Also, only one instance of Face is executed. If Face is already running and Poker detects an outer anomaly (Poker filters out the effect of locally executing programs), Face is not executed multiple times.

Evaluation

We perform our experiments on two virtual machines, hosted using the KVM hypervisor on a HP Z420 workstation with 8 Intel Xeon E5-1620 CPUs @ 3.60 GHz and 16 GB of RAM. The L1i and L1d cache are 32 KB in size, the L2 is 256 KB and the shared L3 cache is 10 MB. Both the VMs have 4 cores and 4 GB of RAM. We demonstrate the utility of our approach by using a specific attack, but the technique is generic enough to detect any kind of cache-timing side channel attacks. Since cloud instances are also commonly hosted on Intel Xeon processors and many frameworks like Google Compute Engine and OpenStack use KVM hypervisor, we expect comparable results to those in public cloud environments.

Poker

We constantly run Poker on the victim VM to assess the memory bandwidth and launch the automated keypress logging attack proposed in [6] from the attacker VM. We divide the timeline into epochs of 100 s each and launch the cache template attack between the 30th and 65th seconds. Figure 3 shows the memory bus bandwidth measured by Poker averaged over a hundred executions. We ensure that no other applications are running on the host machine during the experiment, to avoid third-party interference. As we can see, there is a significant and constant decrease of 2-3 GBps in the bandwidth while the attack is in progress. This because the cache is continuously being flushed by the attacker, leading to cache misses. Since this includes shared libraries which are routinely utilized, they need to be fetched from the main memory after every flush. This leads to significant usage of the memory bus, leading to reduction in available bandwidth.

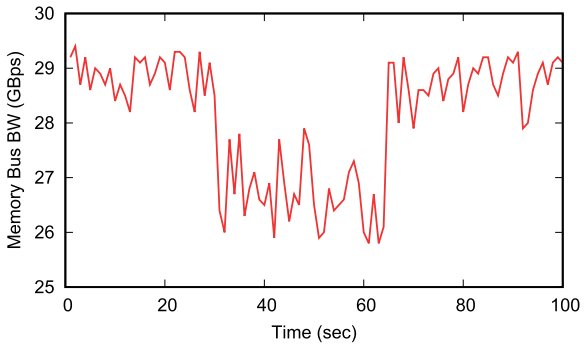

Fig. 3 Memory bandwidth during a cache template attack, as measured using Poker. A cache template attack is launched from a co-located VM between the 30th and 65th seconds. The figure shows the decrease in bandwidth measured by Poker during that interval

We use Poker to benchmark the memory bus performance of an idle Azure D14 instance with 16 cores, 112 GB of RAM and a last-level cache with size 20 MB. The results are shown in Fig. 4. The results depict the bandwidth measured during a day at intervals of 30 min. We continuously monitor the instance and ensure that there is no constant drop which might signify a cache attack or cross-VM interference. It can be seen that the bandwidth varies between 31.5 and 33 GBps and is unpredictable. The results depict the general variation in the available memory bandwidth on public cloud instances which we can observe to be erratic in nature but varying within a small range. Hence we surmise that if we notice a constant drop in bus bandwidth over long periods of time, we can reasonably assume the presence of abnormal or suspicious cache activity.

Scalability with multiple tenants and effect of background noise

Multiple guest VMs are hosted on the same hardware in cloud infrastructure. Each instance can run a variety of processes with varying cache access patterns, thereby generating noise. However, this noise can only reduce the available memory bandwidth. Since Poker suspects an attack when it notices a decrease in the available bandwidth, any background noise can only amplify the requirements.

Non-temporal instructions are often used during applications like media streaming and sparse matrix computations. In such cases, flagging a decrease in bus bandwidth as an indicator of a possible attack might not be correct. One possible solution would be check for disk contention as well in addition to bus contention, since these applications are associated with disk writes (for e.g., storing the media file or the computed results on disk). Disks are also shared across VMs and have been shown to be liable to performance interference [22]. Sysbench [23] file benchmarks can be utilized to measure the disk write latency.

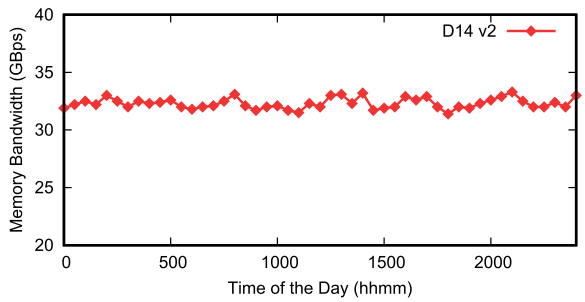

Fig. 4 Variation of memory bandwidth on an Azure D14 instance. The graph shows the variation in memory bandwidth on an idle Azure cloud instance during the course of a day. This depicts the general trend on public clouds and lays the foundation for the inference that a sudden steep decrease in bandwidth can be due to a side channel attack

Face

As we have described in previous sections, Face is a temporal cache cleanser which can be augmented with different libraries to perform cache obfuscation. The libraries need to be properly chosen for the obfuscation to be successful. The attacker will target shared libraries which are regularly used by the victim. The same libraries can be used for performing obfuscation. To justify our claim, we provide a proof-of-concept implementation against keypress logging attacks. Since the attacker targets the GDK library, we use the same to perform obfuscation.

We first launch the attack from the second VM and generate the cache profiles. In the next step, we run Face in the first VM and generate the profiles again. We perform extensive experiments on individual keys and opine that enough noise is generated in the profiles to render them ineffective in their purpose of identifying individual keypresses. Due to lack to space, we present the consolidated results for the five vowels in Fig. 5. We have chosen them because they are among the most used alphabets in the English language. The graphs show the number of hits for each address in the last-level cache (approximately 16,000 addresses) with and without obfuscation. The bars are concentrated in the region which is used by the GDK library for keypresses. Majority of the cache lines accessed by alphabetic keys are the same and the frequency of individual hits is the key to distinguishing them. The number of accesses on certain addresses decrease due to Face interfering with the attacker and decreasing the bandwidth available for the attack. On the other hand, hits on different addresses increase due to the random cache obfuscation being performed by Face.

Cache obfuscation can be considered an effective defense mechanism only if the noise generated is random and makes the profiles indistinguishable. As we can see from Fig. 5, the noise added by Face to the profiles of different alphabets renders them similar for all practical purposes. For instance, the normal profile of *a* can be easily distinguished from that of *e* or *o*, but the same can not be done for the corresponding obfuscated profiles. As a result, it becomes difficult for the attacker to distinguish between different keys and renders the attack ineffective.

Indistinguishability of obfuscated profiles

Chi-square (χ^2) distance or metric [24] is a distance measure used to compare two histograms. We use the formula given by the authors in [25] since it is symmetric w.r.t. both the variables. The chi-square metric for the

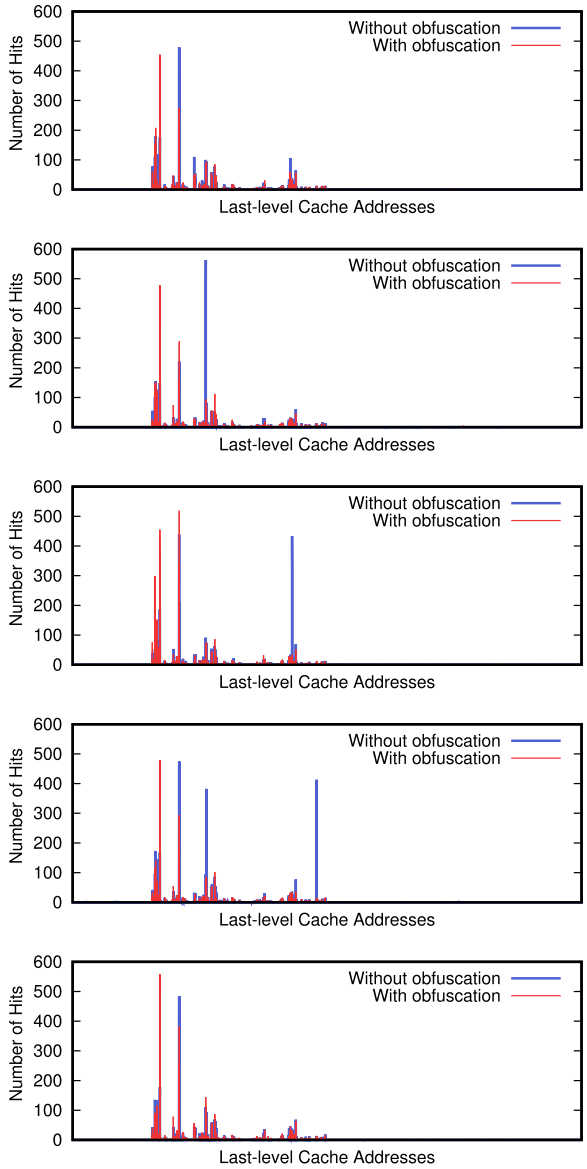

Fig. 5 Effect of Face on the profiling of vowels (*a*, *e*, *i*, *o* and *u* from top to bottom). The graphs show the number of hits for each address in the last-level cache (approximately 16,000 addresses) with and without obfuscation. The bars are concentrated in the region which is used by the GDK library for keypresses. As we can see, obfuscation changes the profiles considerably and also makes them indistinguishable. The normal profiles of the five vowels can be easily differentiated, but the obfuscated profiles of multiple vowels look the same. The blue bars have been drawn slightly wider for clarity during overlaps with the red ones

distance between two binned histograms, whose values are represented by variables x_i and y_i, is given by

$$\chi^2 = \sum_{i=1}^{n} \frac{(x_i - y_i)^2}{(x_i + y_i)} \quad (1)$$

To remove noise from the calculation, we discard the bin indexes where the difference is less than 45. We chose this value since the highest number of cache hits recorded are in the range of 450 and 10% is within reasonable bounds of fluctuation. We apply the formula to the histograms in Fig. 5 and tabulate the results in Table 1. As we can see, the χ^2 metric reduces significantly in almost all of the cases. Since the normal profiles of *a* and *u* are highly similar in nature, the distance measure increases after obfuscation. However, it is still lower than the average distance between other pairs of normal profiles. On average, the χ^2 metric drops by more than 60% after cache obfuscation.

Performance overhead

Both Poker and Face are extremely lightweight single-threaded processes. We evaluate the performance impact of PokerFace on the STREAM [26] and Sysbench [23] micro benchmarks and also on the Parsec [27] benchmark. We have chosen them since they represent different types of workloads at different scale. The STREAM benchmark performs operations like copy, add, scale, etc. on arrays larger than the caches. The Sysbench CPU workload verifies primes less than 2000. The Sysbench memory benchmark performs continuous reads and writes on a 5 GB memory buffer and the file benchmark does the same on a 2 GB file. The Parsec benchmark suite consists of different programs which perform a variety of tasks like cache-aware simulated annealing (canneal), frequent itemset mining (freqmine), online clustering (streamcluster), image processing (vips), video encoding (x264), etc. Since cloud instances are regularly used for machine learning and image/video processing applications, these set of benchmarks are a representative set of real world use cases.

Figure 6 shows the overhead imposed on basic operations from the micro benchmarks. As we can see, Poker causes an overhead of close to 4% during memory

Table 1 χ^2 metric for the histograms in Fig. 5, without (top) and with obfuscation (bottom)

	A	E	I	O	U
A	-	1247.71	1308.60	1230.45	230.35
E	1247.71	-	1587.27	1819.73	968.50
I	1308.60	1587.27	-	2067.68	1129.04
O	1230.45	1819.73	2067.68	-	953.80
U	230.35	968.50	1129.04	953.80	-
A	-	655.77	764.80	459.05	534.83
E	655.77	-	433.50	305.31	351.23
I	764.80	433.50	-	551.15	434.12
O	459.05	305.31	551.15	-	373.71
U	534.83	351.23	434.12	373.71	-

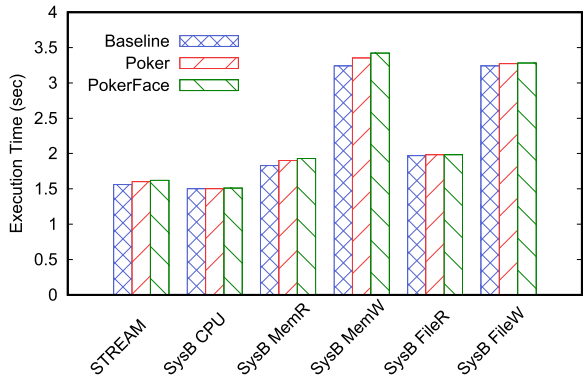

Fig. 6 PokerFace performance overhead on micro benchmarks. The graph shows the overhead incurred by Poker on the Sysbench and STREAM micro benchmarks. These benchmarks represent CPU, memory and file workloads. Since PokerFace essentially operates on the memory subsystem, the memory benchmarks suffer an overhead of up to 7%, of which 4% is due to Poker and always sustained. The additional 3% overhead due to Face occurs only when an attack is suspected and cache obfuscation is triggered. CPU and file benchmarks incur negligible overheads of 1-2%

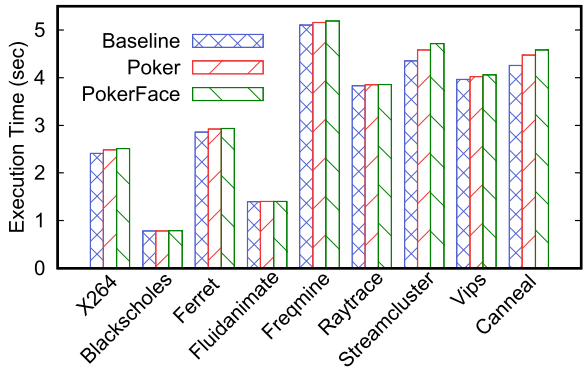

Fig. 7 PokerFace performance overhead on Parsec benchmark. The graph shows the overhead incurred by Poker on the Parsec benchmark suite. The benchmark suite consists of different programs which perform a variety of tasks like cache-aware simulated annealing (canneal), frequent itemset mining (freqmine), online clustering (streamcluster), image processing (vips), video encoding (x264), etc. Since cloud instances are regularly used for machine learning and image/video processing applications, these set of benchmarks are a representative set of real world use cases. Memory intensive applications like streamcluster and canneal suffer from 5% overhead due to Poker and a maximum of 8% overall. Other applications which are predominantly CPU intensive like blackscholes (Black-Scholes partial differential equation), ferret (content similarity search) and fluidanimate (fluid dynamics for animation) incur overheads of up to 3%

benchmarks. This is even lower in case of CPU and file operations. Since Poker is always executed, this is the minimum overhead incurred. Face is utilized only when Poker detects abnormal cache activity. It adds an additional overhead of around 3% to memory operations during the period, which is insignificant considering the task it performs. The maximum overhead due to the approach does not exceed 7% and is very nominal.

Figure 7 depicts the performance overhead incurred on applications from the Parsec benchmark. Programs which are only CPU intensive like blackscholes (application of Black-Scholes differential equation) and fluidanimate (simulating an incompressible fluid for interactive animation purposes) incur close to no overhead, since both Poker and Face are memory intensive and are restricted to a single CPU core each. Programs with frequent memory access patterns like streamcluster and canneal suffer from up to 5% overhead from Poker and up to 8% overall. Other programs incur around 2-3% overhead which is highly minimal. Even with real world applications, performance overheads due to PokerFace are limited to 8%.

Application of PokerFace during the exploitation phase of Prime+Probe and Flush+Reload attacks

We have discussed the application of PokerFace during the profiling phase of cache template attacks in detail due to the following reasons:

1. The profiling phase is highly significant since it helps the attacker to intelligently launch attacks on critical locations and also make sense out of the data gathered.
2. Cache template attacks automate the process to reduce human involvement.
3. It is easier to understand the effect of cache obfuscation on attacker profiles.

In essence, the exploitation phase is similar to the profiling phase in nature. Here, we demonstrate the effectiveness of our strategy during different kinds of attacks and also during the exploitation phase. Gruss et al. [19] demonstrated that Prime+Probe attack incurs a larger number of cache misses for the attacker as well as the victim when compared to Flush+Reload attack. This is because both the attacks are similar in approach. The attacker fills the last-level cache with his data and waits for the victim to use the cache. Prime+Probe operates at a higher granularity of cache sets compared to Flush+Reload which operates on cache addresses. After the prime phase, the victim incurs cache misses since the attacker would have placed his data in the cache. During the probe phase, the attacker would have cache misses if the victim accessed the data in the meantime. The authors also propose a variant of the Flush+Reload attack called Flush+Flush, which relies only on the execution time of the flush instruction. However, this attack can be easily mitigated by fixing the execution time of the instruction without hampering the performance of the system.

We launch Prime+Probe and Flush+Reload attacks from our attacker VM on specific cache addresses (acquired during the profiling phase of our previous experiments) and deploy PokerFace on the victim VM. During our experiments, we encrypt a large file using AES on the victim. It is important to note that a single address is the lowest granularity at which the profiling and exploitation phases can proceed. The attacker can slow down his attack rate by increasing the interval between targeting different addresses, but he/she cannot slow down the process on a single address since the behaviour of the victim needs to be regularly monitored. However, for the attack to be practically successful, monitoring one single address is insufficient. From the histograms in Fig. 5, we can see that at least 5-10 addresses need to be simultaneously monitored to roughly distinguish key presses. Hence, in our experiments, we monitor 5 addresses as a minimal representation of practical attacks.

Figure 8 shows the decrease in bandwidth measured by Poker when the attacks are launched. As we can see, during Flush+Reload, the observed bandwidth is close to what was observed during the profiling phase with a drop of 2-2.5 GBps. During the Prime+Probe attack, the decrease is a significantly higher and varying between 2.5 and 4.8 GBps. This is similar to what was observed by the authors in [19] though they considered cache counters and we observe memory bandwidth.

Though the profiling phase is imperative to successful attacks, we test the robustness of Face with the assumption that the attacker has access to genuine profiles and is monitoring specific cache addresses. Let us consider the profiles of alphabets *a* and *e* generated during the profiling phase. We list a subset consisting of significant addresses in Table 2. Under the assumption that the attacker is constantly attacking these addresses and trying to detect when the victim presses the keys, we launch both

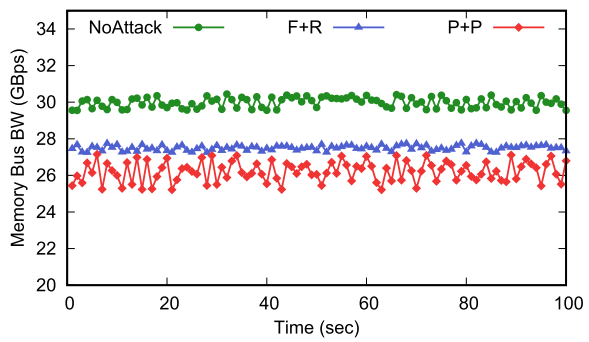

Fig. 8 Memory bandwidth during Flush+Reload and Prime+Probe attacks, as measured using Poker. The figure shows the decrease in bandwidth measured by Poker on the victim VM when Flush+Reload and Prime+Probe attacks are launched independently from the attacker VM

Table 2 Set of significant addresses from the profiles of the vowels *a* and *e*

Address	A	E
0x2b640	○	×
0x34f40	○	○
0x34f80	○	○
0x35040	○	○
0x3c700	○	×
0x42300	×	○
0x42340	×	○

(○ denotes the cache address being accessed and × denotes a miss)

Prime+Probe and Flush+Reload attacks on the specified addresses with and without Face running on the victim. We add the results for an additional address to consider the different cases.

The consolidated results are shown in Table 3. We observed similar behaviour with both Prime+Probe and Flush+Reload and hence, show them together. As we can see, the attacker can discern useful information and distinguish between keypresses when no security mechanism is in place. When Face is running on the victim, it adds noise in the cache with the result that the attacker can not distinguish between keys. This is a minimalistic example of how cache obfuscation will function. As the keys monitored and the related cache addresses increase in number, the patterns will become more and more complex, making it difficult for the attacker to distinguish individual events. We believe that obfuscation during both the profiling and the exploitation can increase the effectiveness of the strategy.

Related work

Multiple approaches to detecting and defeating cache attacks have been proposed in literature. They can be broadly classified into two categories: host-based and guest-based. The host-based techniques rely on access to the system and hardware can be implemented only by the cloud provider or vendor. Wang and Lee [28] proposed security-aware cache designs (RPcache or random permutation cache) to prevent side channels. StealthMem [29] locks a set of cache lines per core which the users can use to store sensitive information. CATalyst [30] describes hardware cache partitioning using Intel cache allocation technology to avoid cache side channels completely. HexPADS [31] gathers information about individual processes from hardware counters to correlate and detect attacks. CloudRadar [17] monitors co-located VMs using a combination of signature-based and anomaly-based detection techniques to check for abnormal cache behaviour. The authors compare a set of common linux commands like `ls`, `grep`, `ssh`, etc. with cache attacks to show a low rate

Table 3 Effect of Face during the exploitation phase of keypress logging attack on *a* and *e*

	Profiles		Keypress without Face			Keypress with Face		
Address	Profile of A	Profile of E	No keypress	*'A'* keypress	*'E'* keypress	No keypress	*'A'* keypress	*'E'* keypress
0x2b640	○	×	×	○	×	○	○	○
0x34f40	○	○	×	○	○	○	○	○
0x34f80	○	○	×	○	○	○	○	○
0x35040	○	○	×	○	○	○	○	○
0x3c700	○	×	×	○	×	○	○	○
0x42300	×	○	×	×	○	○	○	○
0x42340	×	○	×	×	○	○	○	○
0x4eec0	×	×	×	×	×	×	×	○

(○ denotes the cache address being accessed and × denotes a miss)

of false positives. However, such commands do not have a behaviour similar to attacks and will never generate false positives. In [32], the authors utilize hybrid hardware-software approaches to further secure existing techniques like RPcache. However, they incur performance overheads of up to 45% and more than 12% on average. To the best of our knowledge, no purely guest-based approach has been proposed in literature.

Düppel [7] uses periodic cache cleansing on the time-shared L1 and L2 caches to evict as much of the victim data as possible. Gruss et al. [6] proposed a type of spatio-temporal cleansing on the last-level cache, which is not suitable to cloud infrastructures due to its high preprocessing time. We, on the other hand, propose an online temporal cleansing approach which generates noise on random addresses. Our method has no pre-processing steps and has reasonably low performance overheads.

Discussion: the case for guest-based solutions to cache attacks

Cache attacks can be detected and mitigated both by the cloud provider as well as the cloud subscriber. All of the existing techniques rely partly or wholly on hardware support and hence, can only be implemented by the cloud provider. We have advocated the need for security approaches which can be implemented by the common cloud user, without relying on the provider's SLAs. We compare the essential features of the two approaches here.

Host-based solutions

The provider can statically partition the cache among multiple tenants hosted on the same hardware such that they do not share common cache lines, thereby removing the attack scenario. This can be done using Intel's Cache Allocation Technology (CAT) [30]. However, resource sharing is vital to the economy of clouds. Moreover, static partitioning might lead to wastage of resources which might otherwise be used by other non-adversarial tenants. The overheads incurred due to such methods have been shown to be on the higher side. Kong et al. [32] state that the performance of memory intensive benchmarks degrades by up to 45% and the average overhead is greater than 12%.

Alternatively, the provider can monitor hardware performance counters like LLC_MISS (last-level cache misses) and MEMORY_BW_READS (memory bandwidth consumed by reads) offered by the Intel memory controller. Whenever a malicious cloud subscriber launches a cache attack, the LLC_MISS will increase across all cores, due to the flushing of the shared last-level cache. Subsequently, the MEMORY_BW_READS counter will increase due to higher bus usage. Once the host detects unusual or abnormal cache activities from a particular cloud instance, CAT can be used to allocate a portion of the last-level cache to it, leaving the other instances to share the remaining cache.

However, there is no way to distinguish between cache attacks and legitimate cache accesses without intruding into the suspected tenant, which is against the service level agreements (SLAs). A cache attack will necessarily increase the LLC_MISS and MEMORY_BW_READS counters, but the converse is not true. Any memory intensive computation can lead to cache line evictions and increased memory bus usage. Hence, the provider is faced with a dilemma: *whether to ignore the situation and risk a cache attack or partition the cache on mere chance and lose out on the economic advantages of resource sharing?* Further, cache partitioning might lead to underutilization and wastage of resources since all the instances might not use the portion allocated to them.

The service level agreements (SLAs) provided by cloud providers[2,3,4] are oriented towards high availability and do not provide any performance or security guarantees regarding side channels. With cache-based side channel

attacks shown to be practical on public clouds [1, 4, 14], there is a need for other approaches to tackle them.

Guest-based solutions

Since cloud subscribers do not have access to hardware counters, they need to measure the memory bus bandwidth by performing writes and measuring the latency. If a subscriber can detect the occurrence of an attack, he/she can take appropriate measures depending on the nature of the instance and the processes running on it. For e.g., if the reservation on the instance is expiring in a short period of time, a new instance can be reserved rather than renewing the current one. If the processes running on the instance are not performing secret or sensitive computations, it might be reasonably safe just to ignore the attack. Also, since no guarantees regarding side channel attacks are provided in the SLAs of any cloud provider, the subscriber-centric approach is the only alternative.

If the attack is liable to affect the cloud user, he/she can either move the application to another instance or mitigate the attack by cache obfuscation. In essence, guest-based, subscriber-centric solutions provide the subscriber with a plethora of choices, which is not feasible in the case of host-based, provider-centric solutions since the provider does not have information regarding the processes running on the different instances. Moreover, these solutions have a very low performance overhead. The notable differences between the two approaches are listed in Table 4.

One drawback which both types of solutions suffer from is their inability to precisely differentiate between adversarial attacks and legitimate, non-malicious cache accesses. However, we trigger the defense mechanism only when security-critical applications are running on the monitored VM and the extra overhead (even if it is a false alarm) is justified.

Potential evasive attacks

To evade detection by Poker, the attacker can reduce the profiling speed, so that a significant decrease in memory bus bandwidth might not observed. However, that would drastically increase the duration of the profiling period, making the attack less practical and more difficult. Also, cloud instances are often rented for short durations and the victim VM might not be active long enough for the extended profiling phase to finish.

Conclusion

Cloud instances are usually virtual machines hosted on shared hardware. The inclusive last-level cache is susceptible to cross-core side channel attacks. Since resource sharing is paramount to the economy of the cloud, public cloud instances are left vulnerable. These attacks can be detected and impeded either by the cloud provider or subscriber. Most of the existing solutions like cache partitioning and system-level monitoring have been proposed from the perspective of the provider. To the best of our knowledge, cloud providers do not actively adopt these techniques. We present PokerFace, a user-level cache monitoring and obfuscation technique, which can work on unmodified clouds without any changes to the hardware or hypervisor. It can be safely and conveniently used by security-conscious cloud subscribers. Poker detects abnormal cache activity by observing the load on the memory bus and Face performs cache obfuscation while the attack is in progress. Unlike existing techniques in literature, our approach allows the cloud user to use mitigation techniques only when cryptographic applications which need to be protected are executing and an attack

Table 4 A comparison of Host-based and Guest-based solutions against cache-based side channel attacks

Property	Host-based (CAT, RPCache, etc.)	Guest-based (PokerFace)
Detection	Hardware counters	Bus monitoring
Mitigation	Cache partitioning, locked cache, VM migration	Cache obfuscation, app migration
Can differentiate between attacks and legit accesses (free from false positives)	No	No
Underutilization of resources	Yes	No
Mitigation policies active even when attack is not in progress	Yes	No
Mitigation approaches can adapt to the workload on victim instance	No	Yes
Implementable in practice	Subject to the SLAs and economics of provider	Easily, by the subscriber
Performance overheads	Subject to high drop in performance due to no resource sharing (up to 45%)	Modest overhead of $< 8\%$

is suspected. We show PokerFace to have modest performance overheads of less than 5% without obfuscation and less than 8% with cache obfuscation enabled. We also compare host-based and guest-based solutions and infer that guest-based solutions are more flexible and adaptable in the scenario of public clouds.

Poker and Face have been developed as decoupled components with no dependencies on each other. We are exploring other approaches to mitigate cache attacks. With containers supporting cross-cloud portability, security-critical applications can be migrated to a different instance when an attack is detected. Poker and live container migration together can be used to build advanced moving target defense mechanisms against persistent attackers.

Endnotes

[1] https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/

[2] https://aws.amazon.com/ec2/sla/

[3] https://azure.microsoft.com/en-us/support/legal/sla/virtual-machines/

[4] https://cloud.google.com/compute/sla

Abbreviations

CAT: Cache allocation technology; IaaS: Infrastructure-as-a-service; LLC: Last-level cache; PaaS: Platform-as-a-service; SLA: Service level agreement; SSE: Streaming SIMD extensions; VM: Virtual machine

Acknowledgements

The authors would like to thank the reviewers for their detailed comments and suggestions for the manuscript.

Funding

The work presented in this paper was funded by the Ministry of Communications and Information Technology, Government of India.

Availability of data and materials

Not applicable. The benchmarks used are publicly available.

Authors' contributions

This research work is part of AR's dissertation work. The work has been primarily conducted by AR under the guidance of DJ. Both authors read and approved the final manuscript.

Authors' information

AR is a Ph.D. candidate in the Department of Computer Science and Engineering, Indian Institute of Technology Madras. He holds a B.Tech. degree in Computer Science and Engineering. His research interests include cloud computing, cloud security, virtualization and blockchain.

DJ did his Ph.D from IIT Delhi and is currently a professor in the Department of Computer Science and Engineering, Indian Institute of Technology (IIT) Madras, India, where he heads and coordinates the research activities of the Distributed and Object Systems Lab. His current research focus is on building large scale software systems focusing on distributed systems especially cloud and grid computing systems and challenges in big data processing. He is the founding Chair of ACM Chennai Chapter. He was awarded the Boyscast Fellowship in 1997.

Competing interests

The authors declare that they have no competing interests.

Publisher's Note

Received: 23 July 2017 Accepted: 27 November 2017
Published online: 15 December 2017

References

1. Liu F, Yarom Y, Ge Q, Heiser G, Lee RB (2015) Last-level cache side-channel attacks are practical. In: 2015 IEEE Symposium on Security and Privacy (SP). IEEE. pp 605–622
2. Gullasch D, Bangerter E, Krenn S (2011) Cache games–bringing access-based cache attacks on AES to practice. In: 2011 IEEE Symposium on Security and Privacy (SP). IEEE. pp 490–505
3. Yarom Y, Falkner K (2014) FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In: USENIX Security, vol 2014. USENIX. pp 719–732
4. Ristenpart T, Tromer E, Shacham H, Savage S (2009) Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. ACM. pp 199–212
5. Brumley BB, Hakala RM (2009) Cache-timing template attacks. In: International Conference on the Theory and Application of Cryptology and Information Security. Springer. pp 667–684
6. Gruss D, Spreitzer R, Mangard S (2015) Cache template attacks: Automating attacks on inclusive last-level caches. In: 24th USENIX Security Symposium (USENIX Security 15). USENIX Association. pp 897–912
7. Zhang Y, Reiter MK (2013) Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. ACM. pp 827–838
8. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A (2003) Xen and the art of virtualization. In: ACM SIGOPS Operating Systems Review, vol 37. ACM. pp 164–177
9. Kivity A, Kamay Y, Laor D, Lublin U, Liguori A (2007) kvm: the linux virtual machine monitor. In: Proceedings of the Linux Symposium, vol 1. pp 225–230
10. Kocher PC (1996) Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Annual International Cryptology Conference. Springer. pp 104–113
11. Kelsey J, Schneier B, Wagner D, Hall C (2000) Side channel cryptanalysis of product ciphers. J Comput Secur 8(2-3):141–158
12. Bernstein DJ (2005) Cache-timing attacks on AES. Technical report
13. Oren Y, Kemerlis VP, Sethumadhavan S, Keromytis AD (2015) The spy in the sandbox: Practical cache attacks in javascript and their implications. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM. pp 1406–1418
14. Zhang Y, Juels A, Reiter MK, Ristenpart T (2014) Cross-tenant side-channel attacks in paas clouds. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM. pp 990–1003
15. Zhang W, Jia X, Tai J, Wang M (2017) Cacherascal: Defending the flush-reload side-channel attack in paas clouds. In: International Conference on Wireless Algorithms, Systems, and Applications. Springer. pp 665–677
16. Chiappetta M, Savas E, Yilmaz C (2016) Real time detection of cache-based side-channel attacks using hardware performance counters. Appl Soft Comput 49:1162–1174
17. Zhang T, Zhang Y, Lee RB (2016) Cloudradar: A real-time side-channel attack detection system in clouds. In: International Symposium on Research in Attacks, Intrusions, and Defenses. Springer. pp 118–140
18. Intel (2007) SSE programming reference. Intel Softw Netw intel. com/avx 2(7). https://software.intel.com/sites/landingpage/IntrinsicsGuide/. Accessed 4 Dec 2017
19. Gruss D, Maurice C, Wagner K, Mangard S (2016) Flush+ flush: a fast and stealthy cache attack. In: Detection of Intrusions and Malware, and Vulnerability Assessment. Springer. pp 279–299
20. Helsley M (2009) LXC: Linux container tools. IBM developerWorks Tech Libr11. https://www.ibm.com/developerworks/library/l-lxc-containers/. Accessed 4 Dec 2017
21. Merkel D (2014) Docker: lightweight linux containers for consistent development and deployment. Linux J 2014:2

22. Novakovic D, Vasic N, Novakovic S, Kostic D, Bianchini R (2013) Deepdive: Transparently identifying and managing performance interference in virtualized environments. In: Proceedings of the 2013 USENIX Annual Technical Conference. USENIX. pp 219–230
23. Kopytov A (2004) Sysbench: a system performance benchmark. https://github.com/akopytov/sysbench. Accessed 4 Dec 2017
24. Mantel N (1963) Chi-square tests with one degree of freedom; extensions of the mantel-haenszel procedure. J Am Stat Assoc 58(303):690–700
25. Ahonen T, Hadid A, Pietikainen M (2006) Face description with local binary patterns: Application to face recognition. IEEE Trans Pattern Anal Mach Intell 28(12):2037–2041
26. McCalpin JD (2002) Stream benchmark. http://www.cs.virginia.edu/stream/stream2. Accessed 4 Dec 2017
27. Bienia C, Kumar S, Singh JP, Li K (2008) The parsec benchmark suite: Characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. ACM. pp 72–81
28. Wang Z, Lee RB (2007) New cache designs for thwarting software cache-based side channel attacks. In: ACM SIGARCH Computer Architecture News, vol 35. ACM. pp 494–505
29. Kim T, Peinado M, Mainar-Ruiz G (2012) Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In: USENIX Security Symposium. USENIX. pp 189–204
30. Liu F, Ge Q, Yarom Y, Mckeen F, Rozas C, Heiser G, Lee RB (2016) Catalyst: Defeating last-level cache side channel attacks in cloud computing. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE. pp 406–418
31. Payer M (2016) HexPADS: a platform to detect "stealth" attacks. In: International Symposium on Engineering Secure Software and Systems. Springer. pp 138–154
32. Kong J, Aciiçmez O, Seifert JP, Zhou H (2009) Hardware-software integrated approaches to defend against software cache-based side channel attacks. In: IEEE 15th International Symposium on High Performance Computer Architecture. IEEE. pp 393–404