Out-of-Order Execution as a Cross-VM Side-Channel and Other Applications

Sophia d'Antoine
Trail of Bits
roots@sophia.re

Jeremy Blackthorne
Rensselaer Polytechnic Institute
whitej12@rpi.edu

Bülent Yener
Rensselaer Polytechnic Institute
yener@cs.rpi.edu

ABSTRACT

Given the rise in popularity of cloud computing and platform-as-a-service, vulnerabilities in systems which share hardware have become more attractive targets to malicious actors. One of the vulnerabilities inherent to these systems is the potential for side-channels, especially ones that violate the isolation between virtual machines..

In this paper, we introduce a novel side-channel which functions across virtual machines. The side-channel functions through the detection of out-of-order execution. We create a simple duplex channel as well as a broadcast channel. We discuss possible adversaries for the side-channel and propose further work to make the channel more secure, efficient and applicable in realistic scenarios. In addition, we consider seven possible malicious applications of this channel: theft of encryption keys, program identification, environmental keying, malicious triggers, determining virtual machine co-location, malicious data injection, and covert channels.

ACM Reference format:
Sophia d'Antoine, Jeremy Blackthorne, and Bülent Yener. 2017. Out-of-Order Execution as a Cross-VM Side-Channel and Other Applications. In *Proceedings of Reversing and Offensive-oriented Trends Symposium, Vienna, Austria, November 16–17, 2017 (ROOTS),* 11 pages.
https://doi.org/10.1145/3150376.3150380

1 INTRODUCTION

Cloud computing often involves the virtualization of multiple hosts on a single physical host. The cloud abstracts away the real hardware allowing users to instantly allocate any virtual hardware or share hardware with other users.

This approach offers major advantages for business. First and foremost, it is an efficient allocation of resources. Flexibility of resources is another big advantage. Both of these have been driving the adoption of cloud computing in business.

These advantages come at a cost though. Sharing hardware with untrusted parties incurs serious security risks. Multiple virtual machines (VMs) often share the same physical device with the expectation that VMs are isolated from each other. This can be achieved through memory control and other permissions, but there are ways to overcome this isolation. One way to violate isolation is to communicate through the shared physical characteristics of computing on the same machine.

Much existing work on cross-VM channels focus on the scenario of an attacker exfiltrating cryptographic keys [30][2]. Zhang et al. demonstrate an L1 instruction cache side-channel across VMs to extract an ElGamal decryption key across VMs [33] . Tromer et al. use cache access patterns to determine the use of AES keys within the process [25]. Another approach uses rare instructions to flush and reload the L3 cache to leak information about keys [10]. One recent work goes so far as locking and unlocking the memory bus for the entire computer to communicate across VMs [3]. These approaches use unusual instructions in contrived setups which are difficult to hide from any curious party.

In this work we present a novel channel between VMs based on a processor optimization technique called out-of-order execution. This optimization technique is usually transparent to the program, but can be detected in programs with multiple threads. The frequency of out-of-order execution increases when CPU load increases. This correlation can be used to send messages between VMs that are co-resident, i.e. reside on the same physical machine. We construct a channel between VMs running on Xen hypervisor. We analyze the signal and noise characteristics when adding additional VMs on the same physical machine and consider both passive and active adversaries. Finally, we consider seven malicious applications of this channel: theft of encryption keys, program identification, environmental keying, malicious triggers, determining VM co-location, malicious data injection, and side-channels.

2 OUT-OF-ORDER EXECUTION

2.1 Background

Out-of-order execution, also known as dynamic scheduling, is a direct result of processor optimization[23]. To increase processing power, modern CPU architectures implement multi-staged pipelining, allowing for simultaneous execution of multiple instructions. Ideally, this occurs every clock cycle at full capacity, however conditions arise which degrade the overall performance time of the machine. These degrading conditions are called hazards. One such hazard is caused by an instruction which requires a great deal of cycles followed by another instruction which requires its output[8, 24]. For example, take loads and stores to main memory which both require many more cycles than an arithmetic operation. If the information used in either instruction is necessary for future operations, the processor creates a bubble to avoid a potential hazard which results in computational errors[23]. This bubble is a delay in the instruction pipeline until the hazard has passed.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ROOTS, November 16–17, 2017, Vienna, Austria
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5321-2/17/11…$15.00
https://doi.org/10.1145/3150376.3150380

Processor optimization fills in the resulting pipeline bubbles with instructions that have been determined not to depend on the current pending ones. This is called out-of-order execution, when instruction execution order in the processor is not the order listed in the program.

All hazards resolved by this method result in a pipeline order which is determined by the processor to be executed without hazards. However the processor does return the output to the higher processes in the order that it was given, ideally with the logically correct computation results[24].

At least one type of out-of-order execution is observable by the program itself and is known as memory reordering [15]. Memory reordering can occur in several ways in the x86 memory model [9], the focus of experiments in this paper. Every processor instruction set architecture has a memory model which guarantees when loads and stores will occur on the processor. Memory reordering, although mostly unheard of outside of low-level programming communities, is actually easy to demonstrate [19].

2.2 Our Channel

We use a multi-threaded program that facilitates memory reordering. Certain scenarios, in which the processor reorders the instructions, have a computation result that is not expected. Take for example two threads, one with initial values $X = 0$ and $r1 = 0$, the other with initial values $Y = 0$ and $r2 = 0$. When the program executes, $X = Y = 1$ and a swapping occurs where $r1 = Y$ and $r2 = X$. Logically, the expected final values of $r1$ and $r2$ should be respectively $0, 1$ or $1, 0$ depending on which thread executes fastest, alternatively in the case of syncing threads, $1, 1$ may also be expected. However, if the thread instructions are executed out-of-order, where $r1$ and $r2$ are set before the values of Y and X, then the final values of $r1$ and $r2$ will be $0, 0$. A diagram of the actual measurement process is seen in Figure 1.

This output can be exploited as an information leak, revealing the processor's behavior, possibly caused by other cycle intensive programs. The two threaded program described above will react to these external environment changes in the form of having an increased likelihood of returning the out-of-order $(0, 0)$ pairs. Iterating through this model many times returns an average frequency of out-of-order executions inside a certain reference frame. Comparing this frequency against a baseline frequency exposes valuable system information of all processes running on a certain set of shared cores. In the rest of this paper we explore the use of this information as a channel through which two programs can communicate.

2.3 Channel Design

We initially sought to construct a very simple channel to act as a proof-of-concept before creating an optimized channel that would be functional in a realistic environment. This channel is meant as a proof-of-concept for the contrived testing environment where there are only a few virtual machines running.

The channel is comprised of a rudimentary sender and receiver. Each of these is specifically tailored to either create or measure the amount of out-of-order executions. Most of the testing is done to guarantee that our channel can indeed generate bits. Once that is

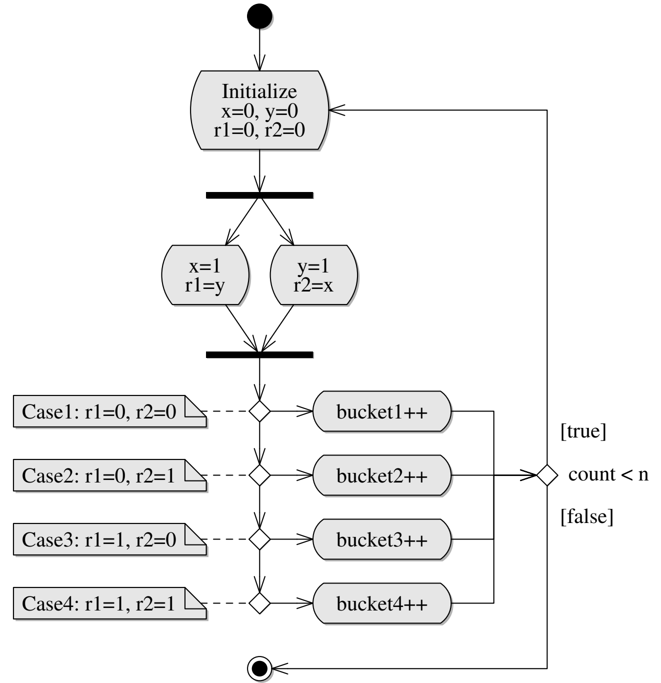

Figure 1: A diagram of a program with two threads which swap values. The variables x and y are memory locations, and r1 and r2 are registers. The result of (r1 = 0, r2 = 0) can only occur when out-of-order execution occurs.

the case, we then also create a channel for sending small bit strings by assigning one period of activity followed by one period of inactivity to be a 0 bit. Two contiguous periods of activity represent the 1 bit.

2.4 Shared Channel

There are two main cases for enhancing the way this channel works. It can either be used as a broadcast or multicast channel by sending a single message to many receivers. Also, multiple senders can be used in tandem to create amplified out-of-order execution frequencies which would give the receivers much more predictable thresholds ; therefore decreasing the span of a single time frame f_i needed to avoid false bits and increasing the bits per second rate of the channel.

One important factor to note is that background noise generated by idle or non participant virtual machines can be easily mitigated since this will only change the amplitude of the gaussian zero centered white noise causing an upwards shift in all average out-of-order execution frequencies without causing much distortion. Our signal will still be just as high above the noise as when the noise is distributed around zero.

2.5 Broadcast Channel

This channel could be very effective as a broadcast channel since all receivers distributed on the virtual machines co-located on the same hardware will get the same message as they will all read the

same frequency of out-of-order executions, possibly with a slight shift in amplitude of out-of-order executions between each virtual machine. This however eliminates the chance of having a multi-cast channel. If unintended receivers are aware of how the channel works and are monitoring the out-of-order executions then there is no way to only send the message to certain receivers and not others.

2.6 Receiver

A time frame f is a set in which a program iterates a set number of times over code. At the end of each iteration variables are checked to determine if out-of-order execution has occurred. This is then added to a summation, denoted by k_r, which represents the total number of out-of-order executions which occurred after time frame f_i is complete. The summed total is then compared to a threshold value, θ, in order to determine which bit the Sender was transmitting. The set number of opportunities, which compose a single time frame, can be increased in order to have a larger window in which to collect out-of-order executions. This results in a higher likelihood for success in receiving the correct bit. Alternatively, this number can be decreased in order to allow for a fast receipt of the final sum of order executions. This results in a quicker reading of the sent bit, while decreasing the accuracy.

The channel must have a preset time period in which an artifact is measured multiple times from the hardware. The average of these measurements can then be mapped to a single bit signal. The receiver or sender is constructed as a program which has a set number of iterations over the reception or transmission code used to take a single measurement from the hardware medium. Each iteration is one measurement and the number of iterations is the number of measurements averaged together. The time it takes for this number of iterations is a single time frame, f_i, used to measure a single bit.

The set number of iterations can be dynamically or statically increased in order to have a larger window in which to collect hardware measurements. This results in a higher likelihood for success in receiving the correct average measurement which maps to the correct bit.

Alternatively, this number can be decreased in order to allow for a fast receipt of a bit which decreases accuracy. This relationship holds true for n time frames in series used to collect an n bit message. In Figure 2, a series of time frames in series form a bit stream.

$$f_i \ = \ time\ frame\ to\ measure\ a\ bit$$

As there is one bit sent per time frame, the bandwidth of the channel, represented as b_i, is inversely correctly to f_i.

$$b_i \ = \ \frac{1}{f_i}$$

The time frame f_i is dependent on the constraints of the hardware medium across which the channel is built. This is an artifact of the time a single hardware measurement takes to collect by the process. For example, hardware media located further from the processor will most likely have longer minimum time frames as it takes longer for a single query to physically reach the component.

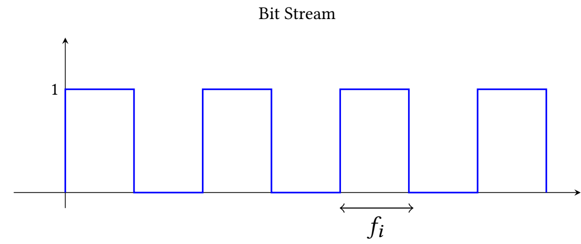

Figure 2: Example Bit Stream, each 1 represents a time frame in which there were more out-of-order executions than the baseline reading. 0 represents the opposite.

3 EXPERIMENTAL METHODOLOGY

3.1 Physical Setup

We setup our lab to mimic the configuration you will find on many cloud services. We ran the latest version of Citrix's Xenserver hypervisor on a machine with 8 processors and 24GBs of RAM. This simulates a smaller scale cloud for experimentation. The VM's were kept very simple. They were all running windows 7 and only had core programs, our sender, and our receiver.

3.2 Experiments

The experiments below were each done twice. Once with the Sender mentioned above where the out-of-order execution could be more finely controlled. The second time with a script doing simple computation in a Cygwin bash shell. This second Sender is implemented to show that the channel could be actualized using a more discreet method that would be less likely to be noticed in a real world scenario. The idea of using common user applications to trigger certain levels of out-of-order execution when run in specific ways has implications for both discrete Senders as well as identification by the Receiver of the processes peripheral virtual machines are running. Each experimental run consisted of both the receiver transmitting ten thousand opportunities and the sender listening for ten thousand opportunities. The only processes open were the virtual machines unless specified, as is the case in which additional background noise is added.

3.3 Testing Hypothesis

From initial testing results and observations we determined several interesting behaviors unique to this channel. Our experiments used three main overarching variables. The number of senders, receivers, and idle virtual machines. The receiver can be modified but for the purposes of showing basic emergent patterns and behaviors of the channel we chose to fix common parameters and only modify the number of VM's acting as one of our main variables. By fixing two of the variables and only changing one we note three main characteristics of this channel.

Firstly, as the number of senders is increased, the channel strength is amplified and the number of out-of-order executions increase. Second, increasing the number of receivers results in similar messages but with different amplitudes. The receivers themselves by nature of how they are detecting out-of-order executions also create an amplification on the channel, although not to the same extent as the sender. Finally we note that increasing the number of idle VM's doesn't have a significant impact on the channel other than a slight change in base noise, and a slight increase in the amplitude.

4 APPLICATIONS

We construct a side-channel which sends and receives information by exploiting out-of-order execution. This side-channel is deployed across virtual machine instances that reside on a Xen hypervisor and are co-located. Additionally, the environment contains four benign virtual machines idling on the system to mimic a live cloud computing environment. All virtual machines share the central processing unit.

We then construct seven different attacks as listed in Section 4.2.4 across this side-channel using the same out-of-order execution sending or receiving processes.

4.1 Theft of Encryption Keys

The first set of attacks are theft of cryptographic keys. Applications of this set are classified as being an exfiltrating side-channel attack which relies exclusively on a receiving application. The intended attack leaks the secret key of an encryption algorithm.

In literature, the use of a hardware side-channel to leak private keys is widely used to attest to the precision as well as the threat level of the side-channel. These include attacks against running encryption and decryption processes as well as a spectrum of algorithms including AES, ElGamal, DES, and RSA [17, 18, 26, 33].

Specifically, we attempt to demonstrate this attack in a simple lab setting with one active client and one malicious virtual machine. This removes the variable element of noise from the proof-of-concept attack.

Additionally, we target a simple XOR encryption algorithm inside a victim process. The client implementation uses C++ and a randomly generated encryption key. Each byte is randomly chosen between a range of ten and a hundred.

The attack begins immediately after the client virtual instance launches its encryption function and ends after. During this time frame, the receiver inside the malicious virtual machine records out-of-order execution patterns from the shared central processing unit. This is done in using the protocol discussed in the first few subsections of Section 4. The bit pattern which is recorded is then a function of the XOR operations executed by the victim's encryption process.

The encryption process was run 100 times, re-encrypting the same basic string of length 64 filled with ASCII 'A's. Every other set of eight bytes are XOR-ed using a randomly generated byte, each XOR uses 7000 small XORs of the same number for the purpose of testing the proof-of-concept. The seed for this random factor was provided by the standard C++ *rand*() function.

The reason we chose to only XOR every other set of eight bytes was to create an obvious fluctuation between central processor contention. The purpose of this was to generate binary activity, either encryption activity or none, by the encryption proof-of-concept on the CPU in order to reliably receive executed operations in the malicious virtual machine. Future work may include the application of intelligent algorithms to the current, simplistic receiver in order to parse and identify leaked CPU behavior induced by higher order encryption algorithms. The receiving application eavesdrops from the co-located, malicious virtual machine and runs the out-of-order execution recording process outlined earlier in this section.

The receiver implemented for this attack was able to reliably identify the different XOR blocks and non-XOR blocks of eight bytes, or sixteen bits, which were executed by the targeted encrypting process. However, there was a lack of granularity in the received number of out-of-order executions per time frame which prohibited us from mapping specific levels of out-of-order executions to the values randomly used in the byte-XOR. Instead, each block of out-of-order executions were declared either a '1' or a '0'. A '1' refers to values received in a time frame which may be mapped, with a degree of certainty, to a XOR operation being executed by the victim. A '0' implies no XOR was taking place. The recorded result of this attack may be seen in Figure 3.

It is apparent that the blocks of out-of-order execution containing bit strings of '1' are mappable to the byte blocks which were XOR-ed, the XOR-ed bytes are represented by a single byte, 'B'. The four encrypted blocks of eight bytes each, shown above, took the receiver 4.9525 seconds on average to leak across the central processing unit with a standard deviation of 0.15606 seconds.

Using the eight byte block method to create clear time frames of encryption, the receiver was able to map blocks of active-XOR and nonactive encryption with an accuracy of 85.9%. This accuracy is high enough to confidently map the periods of high and low operations in our chosen encryption algorithm.

The success of this attack lies in the ability for the malicious virtual machine to leak active behavior from the co-resident process. This may be seen as an attack on both the privacy aspect of transparent behavior by a client in cloud computing environments. Also, this attack highlights the possibility of a simplistic, but successful attempt to learn the victim's encryption algorithm used by a process.

Future work on this topic includes learning algorithms as well as general improvements on the reception channel to achieve increased precision rates. Additionally, this would allow an attacker to better connect different out-of-order execution patterns with complex encryption schemes as well as specific numeric values being used in them.

4.2 Active Program Identification

Using this side-channel, an adversary can eavesdrop on concurrent processes. Attacks of this type can uniquely identify co-active

Starting Bytes:

AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA

Encrypted Bytes:

AAAAAAAA BBBBBBBB AAAAAAAA BBBBBBBB

Bits Leaked:

0000000000000001 1011011011111101 0000000000000010 1111111101011101

Figure 3: The encrypted string compared to the leaked string.

applications as well as some more prominent functions. In this attack, the processes eavesdrops on system behavior using the channel defined above. Recording specific, repeated out-of-order execution patterns allows an attacker to map behavior to specific process identifiers.

We ran this attack one hundred times. Each duration lasted for 32 time frames, or roughly 3 seconds. During these runs, five co-located virtual machines were actively running. The targeted virtual machine was running instances of YouTube inside Google Chrome.

For our proof-of-concept, we sought to eavesdrop on this VM and confidently identify, with a high degree of certainty, what application, if any, was being run.

For each period of reception, the malicious application would record a bit stream of length 32. The pattern of bits averaged over several runs was then used to classify the co-active process as either being a high or low generator of out-of-order execution. From there, the average bit pattern could then be mapped to a prerecorded pattern of a known active Chrome session stored inside the malicious application binary.

The average time of each run was 3.13294 seconds, assuming the program was recording a bit stream of length 32. There were five co-located VM's sharing the central processing unit with the one virtual machine actively running the victim process.

The standard deviation of this experiment was 0.14234 seconds. The success rate of mapping active, unknown applications to one of two sets , either out-of-order execution generators or not, was 100%. However given more system noise, such as the numerous applications which would be co-resident in a highly active cloud server, the addition of higher order algorithms need to be applied to parse out identifying information from a system.

The lab environment contained six virtual machines running on a single Xen server. Under these conditions, the specific identification of a running instance of Chrome, as opposed to other programs artificially used to generate noise, was successful on a, average of 93%. This is significantly high enough to reliably identify a client running video instances inside this browser process.

The success of this basic attack carries implications on both a privacy and information security level as well as on a systems level. When concurrent processes continually leak data across virtual machines, the privacy of a user's activity may be called into question.

Future work on this topic includes further testing and averaging to create a larger database of patterns mapped to their associated processes, i.e. Safari, Firefox, or IE, under different system loads, i.e. while 1 virtual machine is running or 10.

The possibility for a mapping is shown to exist through our preliminary work. Given the possible precision an attack could achieve, identification of specific program execution by a client is detectable. An example of this precise identification may be an attack confidently identifying a user's physical input into a running program.

Bits Leaked from System Baseline Activity:

...0000 **000000000 000000000** 00000000 **000000000** 000000000...

Bits Leaked from Client Running Chrome:

...0100 **010101011 010101011** 01010010 **010101011** 010010010...

Figure 4: The bit stream leaked through the receiver showing the repeating pattern associated with running videos in Chrome.

If Figure 4, the results of this described attack may be observed in two different segments of the bits leaked from the CPU's out-of-order execution. In this receiver, each bit represents 100,000 individual out-of-order execution checks average together in order to reduce the affect of noise on the final bit stream. Each bit of this continual stream was recorded, on average, in 0.18806 seconds. As can be seen, while the victim was not running any programs, the bit stream was entirely '0'; however, after opening Chrome to play a video, the bit stream stabilized into the pattern shown above. This specific test was repeated 100 times in order to positively identify a mapping between the targeted application and out-of-order execution patterns exfiltrated from the system.

4.3 Environmental Keying

Environmental keying attacks rely exclusively on the receiving side-channel application. These channels are defined as using only a reception process to record out-of-order execution patterns as a bit stream. This stream represents the environment in which the malicious virtual machine resides[21].

For our specific implementation of an application from this attack, we chose to implement a simple environmental keying malicious program. The attack contains two distinct phases. The first is a malicious virtual machine which runs an environmental keying side-channel program to generate a unique key used to identify the system. The second stage uses a program located on a victim virtual machine that contains an encrypted payload. This application uses a replicated receiver to record system out-of-order execution patterns. If the patterns recorded match the targeted pattern identified by the malicious host, the malware decrypts its payload.

The crux of this attack lies in the generation of a unique environmental key which identifies the targeted environment. This allows a malicious application to gain location-awareness in order to expose its malicious behavior only when located on the proper virtual machine [21, 31, 32].

For our simplified implementation of this attack scenario, we set up 6 virtual machines on a Xen server with one machine categorizes as the malicious host and another as the victim. The receiver on the host VM receives a bit stream, the unique identifier,

using the out-of-order execution receiver discussed earlier in this section.

A malicious application with an encrypted payload and the unique ID may then be dropped onto the target VM. This malicious process immediately begins the duplicated receiving process to eavesdrop on the central processing unit behavior and ID the environment. If the identifiers match, it unpacks the payload and executes.

Host identity-based encryption may also be possible using this attack setup assuming the unique identifying string can be 100% recovered by the malicious process running inside the targeted virtual machine. This may require future work in channel optimization and averaging out system noise. We show that a unique identifier can be recovered by 83% which allows the application to decide if the identifier it records and the one pre-recorded by the host malicious virtual machine are close enough to confidently assert that the environment is the right one and execute accordingly.

The host virtual machine ran the out-of-order execution receiver to collect a key of length 32 bits, this averaged out to 27.29 seconds, or 3.41 seconds per 4 bit segments and a standard deviation of 0.064 seconds. An example of a unique environment key can be see in figure 4.

This key was then encoded into the deployed malware containing the encrypted dropper which was then installed on the targeted virtual machine. Once started, the application began the out-of-order execution receiver to record the same length bit stream as the host receiver captured. This 32 bit sequence was compared to the encoded bit stream representing the environment in which the malware should unpack.

This process was executed for 100 trials to compute an average percent similarity between the environment the malware was in and the expected environment identifier. Under the contrived circumstances of this laboratory setting, we found that the malware recorded an environmental identifier which correctly matched its environment to the one represented by the host's encoding identifier with 96.87% accuracy. This matching was deemed sufficiently high to identify the targeted system.

Future work on this subset of malicious applications can build from the use an out-of-order execution side-channel to identify unique environment keys. This work will include creating intelligent algorithms to better record individual bits based on the frequency of out-of-order executions. The goal would be to generate a receiver which can guarantee a repeatable reception of a specific bit stream. Once the key can be guaranteed, it may be used in the actual encryption/decryption on the payload.

In the current status of the attack, the received environment key can be deemed similar enough to correctly identify the system. This allows the receiver to be used as a binary decider. If the bit stream eavesdropped off the CPU is close by a given threshold to the original recorded by the host, the environment is positively identified and the malware executes. This inherently poses a threat to the privacy and security of virtual machines stored in the cloud and leaks valuable location information. As this attack was successful, future development on applications in this category also show potential to successful exfiltration.

4.4 Signal Trigger of Process

This attack model requires a transmitting as well as a reception process which are located on two distinct co-resident virtual machines. Additionally, it requires a pre-arranged time frame as both processes must have overlapping active periods for the success of the attack.

Either the sending process transmits continually waiting for the receiver read the signal. Or the receiving processes idles until it reads a one time signal.

Both methods rely on the use of message transmission across our constructed channel, to exploit forced variance in the out-of-order executions off of the CPU. The channel processes use the same algorithms outlined earlier in this section and used by all attacks described in this paper.

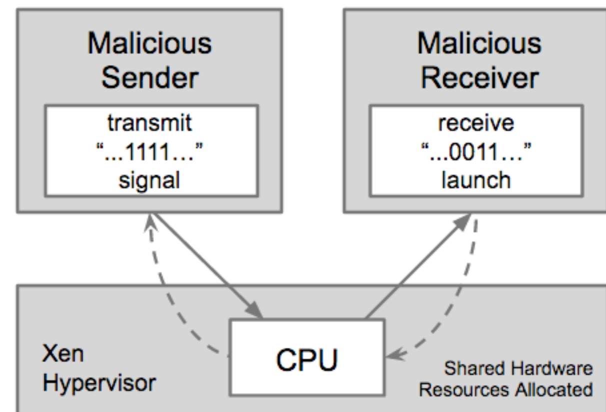

Figure 5: Use of the central processor in synced, concurrent time frames allows the transmission of a signal between two colluding applications in real time.

In figure 5, the use of the continually operating receiver can be seen. The receiver reads out-of-order execution patterns from the shared central processing unit in pre-arranged time frames. At the start, the transmitting program transmits the signal "111.." as a bit stream. It forces high levels of out-of-order executions repeatedly for several time frames. Each time frame represents a single bit. The receiver detects the high bit stream and launches its intended attack.

The resulting length of time necessary to run four bits in this described attack one hundred times repeatedly in the same environment is 1.79025 seconds with a standard deviation of 0.07816.

The environment of this laboratory system contains six idling virtual machines of which only two are active. One is the malicious host containing the sending process and other contains the reception process which is continually listening for the beacon signal. Additionally, we assume the co-location of the two interactivity virtual machines could be verified prior to the attack. The ability for an accurate time frame to be calculated from inside different virtual machines is also assumed.

Experimentation with this attack using multiple transmitting processes from inside different virtual machines to increase the

frequency of the forced out-of-order executions seen across the shared central processing unit by the targeted receiving process, resulted in degradation of the signal's precision.

Initially, increasing the number of senders did improve the broadcast signal's bandwidth. However, using the maximum number of machines virtually allocated on one physical server increased the noise to an amplitude higher than the out-of-order execution signal. This meant that the precision gained through multiple senders increasing the bandwidth was minimized by the noise of the system, forcing miss-reads in the receiver and failing the attack. Further experimentation is needed to test the limits of increasing signal strength through introducing additional concurrent senders versus increasing system load and noise levels.

Building more complex attacks off of this simple triggering signal requires little effort on the part of the advisory. This advisory to wrap the receiver in a obfuscating program with an arbitrary payload to execute upon receiving the signal.

Our basic attack model implemented to transmit a signal between two colluding parties co-located on shared hardware realizes a basic proof-of-concept channel attack. The implications of this simplistic, exfiltration vector span across violations of both unauthorized data access as well as active interference with the target's private virtual machine.

4.5 Adversarial Data Injection and Alteration

A final type of adversary, which can be built on this out-of-order execution side-channel, is destroying the integrity of the instructions being used to compute values in the victim process. It is classified as an infiltrating side-channel attack. This adversary can be built using the the transmitting methods described in this paper. The purpose of this attack is to force contention of central processing unit resources as well as pointedly alter the order of critical instructions used for computations by the target thread.

In literature, the contention of any resource which negatively affects the targeted user is often referred to as a Denial-of-Service attack (DoS) [7, 16]. This elemental intrusion of the user's environment does not require the least level of precision compared to other attacks discussed in this paper.

Compared to other adversarial models, this attack requires the greatest, consistent signal amplitude in order to significantly impede the CPU computations for the co-active processes. The difficulty with this interference comes from the hypervisor's resource scheduler and optimizations which attempt to decrease the constant load caused by the transmitter.

To consistently force out-of-order executions in the processor, the transmitter must use larger time frames, f_i. This allows the transmitter to execute more out-of-order execution generating assembly code to account for the few instructions which are optimized out of the time frame, f_i, by the hypervisor.

The effect of large time frames is an increased execution time for the attack. We implement a specific attack which attempts to interfere with the target's computations through increasing the out-of-order executions in the processor.

After a certain threshold level of these executions, the processor returns invalid or reordered values to the target process, thereby meeting our requirements for a denial-of-service attack. In our scenario, the service required by the target process is processor execution of specifically ordered instructions to result in precise values. Additionally, invalid return values have more interesting application in the art of exploitation; however using it effectively will require future research and greater precision.

The predicted increase in the minimum duration needed to successfully execute a DoS attack is seen in our implementation, against an isolated victim process running in four consecutive time frames, f_i. The average run time of 2.21538 seconds is measured from one hundred tests run on a Xen server with 6 virtual machines. The standard deviation these runs is 0.11023 seconds.

These results imply that the increase in bandwidth of the transmitted signal effects the precision of each run and generating higher variance in minimum time frame durations needed to interfere with the victim process. Additionally, the increase in signal strength from using multiple sending processes added noise to the out-of-order executions read in each time frame.

Combined, the decrease in precision from the larger f_i and the noise from the larger number of sending processes used to increase signal strength adversely affected the intended binary transmission. The attack operated successfully with the use of one to four virtual machines, operating at a threshold above the generated noise and variance. However, the attack failed under five virtual machines operating the transmission process. 5 virtual machines used to send a broadcast signal to clog the processor is the limit of the signal strength for the size of the laboratory Xen environment.

The attack success was measured in value miscalculation as computed for the victim process. On average, the successful runs of attack caused a 50% value loss in the computation of the targeted operations. The target process ran a while loop which read from an array and, in two threads, multiplied it by a constant value, storing the results back in the same index. This array could then be compared to the expected values pre-computed at the end. Out-of-order values meant that the processor did not obey program order, showing that the adversary was successful for the time frame of that array index's computation.

Example Target Process Array Before Calculations
[7, 4, 0, 9, 2, 8, 5, 7, 0, 9, 8, 7, 1, 2, 9, 4, 8, 5, 7, 3, 0, 2, 8]
Target Process Array After Multiplication with 5
[35, 20, 0, **9**, 10, **8,5**, 35, 0, 45, **8,7,1**, 10, 45, 20, **8**, 25, **7**, 15, 0, **2,8**]
Target Process Array Expected Calculations
[35, 20, 0, 45, 10, 40, 25, 35, 0, 45, 40, 35, 5, 10, 45, 20, 40, 25, 35, 15 0, 10, 40]

Figure 6: The array values before and after computation.

Figure 6 shows values of the array which were adversely affect during the computation due to the out-of-order executions forced by the malicious transmitting process. The success of this specific attack was 43.43% based on the number of stores in the array which were reordered to occur prior to the multiplication instruction. Additional testing to determine limitations of this attack on larger scale cloud environments will help. Increased noise tightens the

boundaries of malicious applications which fall under this category.

4.6 Determine VM Co-Location

One fundamental requirement to create a side-channel is establishing co-location of the virtual machines. These virtual machines must share one or more hardware components. This requirement is discussed in Section 1 and 2.

This application exploits out-of-order execution on the central processing unit to create a side-channel. It then verifies co-location with another colluding, malicious virtual machine with a threshold degree of certainty.

For the experimental setup, shown in figure 7, we hosted six virtual machines on a Xen Server with one selected as the malicious host receiver. This virtual machine attempts to verify its physical location. From the remaining virtual instances, we chose at random another to alternate between acting as a colluding virtual machine. If the malicious host VM determined co-location during a period that the chosen VM was colluding, a success was recorded. If it determined co-location during a period that the chosen VM was benign, a failure was determined and vice versa.

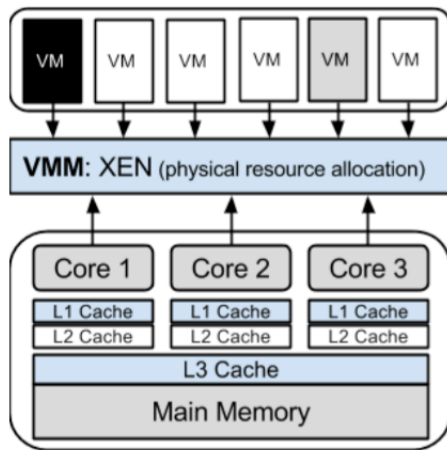

Figure 7: The black virtual machine represents the malicious host and the gray virtual machine represents the alternate continual transmitter.

The chosen, colluding VM continuously transmits a signal composed of time frames, f_i, which is read once by the malicious host VM, started on the Xen server.

Once the receiving process finishes this one time read of off the central processing unit, it makes a binary decision, comparing the read activity levels to a pre-determined threshold value. Based on the lack of noise in the simplistic environment used for our implementation, the threshold can be set closer to the expected results.

Running this scenario two hundred times interspersed with cases where co-location should be detected and should not, the overall percentage of correct co-location detection was 97% under the assumption of no concurrent, active processes that would significantly impact the noise threshold of the channel.

This level of successful identifications was in part a result of the increased length of each time frame, allowing to better average out any false positive readings. However, this did impact the overall time necessary for the attack, bringing the minimum duration that the adversary needs to record the system for four time frames to an average of 3.13295 seconds. The standard deviation of this measurement between all experimental results was 0.2171 seconds. These results make this successful attack the longest of the seven categories explored in our work. Further research may be done using varied testing environments to better test the boundaries of this attack at larger scales.

Based on our initial survey of the cloud computing environment, there are two distinguishing variables to explore. First, the increased levels of or variance in noise from surrounding processes. Additionally, the partial processor co-location where a virtual instance is allocated time on processors belonging to two or more separate cores on the same server. Both factors listed are sufficient to interfere with the success of an attack. Also, they are common enough to be present in the majority of live cloud computing systems.

The one time reception of an unique signal which is transmitted by a continuous sending process classifies this attack as operational across a channel. This attack creates an information leakage between virtual machines which should otherwise be operating in isolated segments of the hardware. The infiltration of the shared hardware system by the transmitter allows the colluding process to leave an artifact in a region of the server where the user is otherwise not privileged to access. The success of this attack can then be seen as a violation of privacy, an unauthorized escalation of privileges, as well as a physical exploitation of the processor pipeline.

4.7 Covert Channel

The final category of attacks requires the continual operation of both a transmitting process and a receiver process. This generates a communication between two colluding applications located in separate, co-located virtual machines.

For our experimentation, we used the same environment setup as with the previous attack implementations. This includes the Xen server and six virtual machine instances which share all physical cores available on the server. Additionally, we assumed that there is a pre-arranged start and stop agreed on between both processes. We assume there is a pre-established time frame duration in which a single bit is measured. Testing under these conditions, the variable of noise was included through either active or inactive, coresident virtual machines.

In the implementation of this attack, the two malicious hosts each contain both side-channel processes, one sender and one receiver. The processes are alternated between to generate bi-way, binary communication. A single test run included four bits transmitted and received by both parties. A success was measured when more than two of the four bits recorded matched those that were sent.

One virtual machine was designated as sending first and listening second, the other virtual machine took the opposite role. The receiver finished recording the system after four time frames to acquire the entire binary message. Following this, the transmitter

residing on the same virtual machine began sending the designated four bits. The duration of these two stages make up the length of the communication attempted.

In order to maintain an average percentage of correctly received bits, each time frame, f_i, was found to be 0.95 seconds given only the four idling virtual machines in the experimental settings. This number may change depending on specific system variables. The time frame duration raised the time to 3.80 seconds for a single four bit message to be sent. This attack is the longest of the seven. The standard deviation on one hundred tests was 0.13 seconds, the third highest variance of the seven attacks.

Overall, the minimum time needed to generate a successful attack, where more than half the total number of transmitted bits are correctly received, increased. Further research into optimization algorithms and communication protocols would undoubtedly decrease this time. For instance, an example communications algorithm may be implemented such that a single bit is transmitted three times in a row and the receiver takes the most common bit as the intended message.

One element of the communication channel is that it transmits messages via a broadcast signal. All shared processor activity may be received by any number of eavesdropping virtual machines provided they use an identical receivers and a synchronized time frame. Therefore, communicating parties using the hardware side-channel cannot be certain that co-located processes are unaware of the transmissions.

However, the processor side-channel may be considered to provide covert communications given the obscurity of the medium over which the message is sent. The hardware processor provides this covertness for our out-of-order execution channel as the majority of communication monitoring efforts target dynamic observation of active network traffic. While this attack was implemented to transmit and receive between a single malicious host and a single malicious client application, there is potential for further attack development.

The broadcast nature of the physical side-channel may be exploited in order to intentionally communicate with multiple virtual machines containing a reception and transmission application. Using multiple virtual machines colluding with a central malicious host VM, a botnet may be generated which resides on a single physical server as outlined in Figure 8.

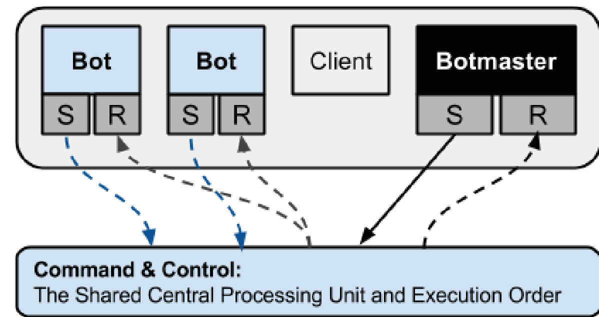

Figure 8: A botnet which uses n bots that receive commands and transmit responses to the Central Authority. Our CPU side-channel acts as the C&C relay.

5 DETECTION OF ATTACKS

In our experiments, we assume that the data stored inside a virtual machine is opaque to the hypervisor. Therefore, all that may be used to detect side-channel exploitation is the dynamic interaction between a virtual machine's processes and the surrounding shared hardware.

Recording full system activity over a period of time generates records of distinct resource consumption patterns. The hypervisor may then match these records with known resource consumption habits that are not permitted. Advancements in machine learning will further enhance the effectiveness of these techniques in side-channel detection. Such areas for improvement include pattern matching and hypervisors which dynamically learn new malicious patterns.

Additionally, some typical malware detection techniques may also be applied for the detection of side-channels [12, 22]. Examples of techniques from this subset include monitoring system calls, recording resource queries, and prohibiting repeated behaviors on which side-channels depend. These techniques are implemented at the hypervisor or host level. When such queries occur, the intelligent hypervisor may decide whether the call is blacklisted, whitelisted, or suspicious. Detection of communication behavior across the hardware not only discloses the presence of a hardware side-channel exploit, but also uncovers what malicious transmission is sent and received given the broadcast property of the signal.

On average, protection methods which prohibit virtual machines access to shared hardware resources are most effective [11, 13, 27]. However, such methods reduce the intended performance of the cloud system.

Given the strong parallelism between network communication and resource based side-channels, the application of signature, anomaly, and pattern based detection techniques should be further explored.

6 RELATED WORK

Side-channels can be categorized into three classes [33]: time-driven channels, access-driven channels, and trace-driven channels.

Time-driven channels communicate through the execution time of a program by measuring how it is affected by it's input like a cryptographic key. Osvik et al. show a time-driven channel in which they trigger an encryption, evict the cache, and then trigger a second encryption and time it [17]. Ranjith et al. assess covert channels across VMs based on network timing [20].

Our channel is a cache-based, access-driven channel and is a direct extension to [4][5]. Access-driven channels operate through an attacker competing with the target program for some resource, and the availability of that resource is affected specifically by the input to the target program. Zhang et al. demonstrate an L1 instruction cache side-channel across VMs running on Xen with a symmetric multiprocessing system [33]. They apply their technique in extracting an ElGamal decryption key across VMs. Percival shows

that cache sharing between threads on a simultaneous multithreading CPU can be used as a medium for a covert channel [18] and demonstrates the channel against OpenSSL. Wang et al. use the contention of shared functional units among SMT processors and look at cache usage on STM CPUs [28] . Saltaformaggio et al. establish a channel through locking and unlocking the memory bus and focus on the construction of a hypervior-based defense to detect memory locking side-channels [3]. Tromer et al. use cache access patterns to determine process behavior, specially the use of AES keys within the process [25]. Wu aet al. use the memory bus as a covert channel [29]. Ristenpart et al. focus on establishing co-residency between VMs in the cloud [21], and then applies the prime+probe cache technique from [25][17][14] to establish a covert channel. Irazoqui et al. use flush+reload on the L3 cache to leak information about keys used in OpenSSL [10]. Xu et al. assess the ideal models and experiments of existing L2 cache cross-vm covert channels [30].

Trace-driven channels continuously measure some property of the system which is affected by input, i.e. power consumption. The authors in [1] present an efficient trace-driven cache attack in which they measure the costs of the analysis phase against the number of traces needed. The number of cache misses is taken from the entire targeted machine. Another method of recording full machine cache misses through electronic emanations is presented by [6].

7 CONCLUSION

This channel is currently not optimized and is only capable of sending small bit strings. This does however already provide for a few useful scenarios; sending keys, sending triggers, sending handshakes and authentication packets or hashes and similar tasks. To optimize this channel we can add in error correcting algorithms. These methods would allow for better thresholds since the parity bits would correct for misread bits. While this would make the channel more reliable it would also require more periods per bit. This channel should also be able to adapt as a multi-cast channel when there are any number of idle or noisy but non adversarial virtual machines in play. For this either we can have an adaptive threshold or use data network methods such as measuring for the variance of the noise. Since our spikes should have significance, this would help parse out what is and is not noise, although it would not thwart an adversary who is sending signals to interfere with ours. This channel can also add in additional layers of security which are standard now in communication theory. Currently it is a covert channel because of how hard to detect the premise is, but if an adversary did know this channel was potentially being used we would need actual cryptographic layers of security to thwart their attacks.

REFERENCES

[1] Onur Acıiçmez and Çetin Koç. 2006. Trace-driven cache attacks on AES (short paper). *Information and Communications Security* (2006), 112–121.

[2] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Fine grain Cross-VM Attacks on Xen and VMware are possible! Cryptology ePrint Archive, Report 2014/248. (2014).

[3] Xiangyu Zhang Brendan Saltaformaggio, Dongyan Xu. 2013. BusMonitor: A Hypervisor-Based Solution for Memory Bus Covert Channels. *EuroSec'13* (2013).

[4] Sophia D'Antoine. 2015. Exploiting Processor Side Channels to Enable Cross VM Malicious Code Execution. In *Recon*.

[5] Sophia D'Antoine. 2015. Exploiting Processor Side Channels to Enable Cross VM Malicious Code Execution. In *Blackhat*.

[6] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. 2001. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems—CHES 2001*. Springer, 251–261.

[7] D. Grunwald and S. Ghiasi. 2002. Microarchitectural denial of service: insuring microarchitectural fairness. In *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*. 409–418.

[8] W. Hwu and Y. N. Patt. 1986. HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality. *SIGARCH Comput. Archit. News* 14, 2 (May 1986), 297–306.

[9] Intel 2010. *IntelÆ 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*. Intel, Santa Clara, CA, USA.

[10] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a minute! A fast, Cross-VM attack on AES. Cryptology ePrint Archive, Report 2014/435. (2014). http://eprint.iacr.org/.

[11] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium (Security'12)*. USENIX Association, Berkeley, CA, USA, 11–11.

[12] I. Kyte, P. Zavarsky, D. Lindskog, and R. Ruhl. 2012. Enhanced side-channel analysis method to detect hardware virtualization based rootkits. In *Internet Security (WorldCIS), 2012 World Congress on*. 192–201.

[13] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 118–129.

[14] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, and Carlo Alberto Boano. 2017. Hello from the other side: SSH over robust cache covert channels in the cloud. (2017).

[15] Paul E. McKenney. 2010. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA. http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html

[16] Thomas Moscibroda and Onur Mutlu. 2007. Memory Performance Attacks: Denial of Memory Service in Multi-core Systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium (SS'07)*. USENIX Association, Berkeley, CA, USA, Article 18, 18 pages.

[17] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2005. Cache Attacks and Countermeasures: the Case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006*. Springer-Verlag, 1–20.

[18] Colin Percival. 2005. Cache missing for fun and profit. In *Proc. of BSDCan 2005*.

[19] Jeff Preshing. 2012. Memory Reordering Caught in the Act. (2012).

[20] P. Ranjith, Chandran Priya, and Kaleeswaran Shalini. 2012. On covert channels between virtual machines. *Journal in Computer Virology* 8, 3 (2012), 85–97.

[21] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*. ACM, New York, NY, USA, 199–212.

[22] Mohamed Saher and Jayendra Pathak. 2014. Malware and exploit campaign detection system and method. (Sept. 10 2014). US Patent App. 14/482,696.

[23] James E. Smith and Andrew R. Pleszkun. 1985. Implementation of Precise Interrupts in Pipelined Processors. *SIGARCH Comput. Archit. News* 13, 3 (June 1985), 36–44.

[24] R. M. Tomasulo. 1967. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM J. Res. Dev.* 11, 1 (Jan. 1967), 25–33.

[25] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptol.* 23, 2 (Jan. 2010), 37–71.

[26] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, and Maki Shigeri. 2003. Cryptanalysis of DES implemented on computers with cache. In *Proc. of CHES 2003, Springer LNCS*. Springer-Verlag, 62–76.

[27] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. 2014. Scheduler-based Defenses against Cross-VM Side-channels. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 687–702.

[28] Zhenghong Wang and Ruby B. Lee. 2006. Covert and Side Channels due to Processor Architecture. 473–482. http://www.acsac.org/2006/papers/127.pdf

[29] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium (Security'12)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=2362793.2362802

[30] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. 2011. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop (CCSW '11)*. ACM, New York, NY, USA, 29–40.

[31] Lin JC Wang JF Zhang XJ. Yu S, Gui XL. 2013. Detecting VMs Co-residency in the cloud: using cache-based side channel attacks. *Electronics and Electrical Engineering*. 19, 5 (2013), 73–78.
[32] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. 2011. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP '11)*. IEEE Computer Society, Washington, DC, USA, 313–328.
[33] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 305–316.