EXPLOITING PROCESSOR SIDE CHANNELS TO ENABLE CROSS VM MALICIOUS CODE EXECUTION

By

Sophia M. D'Antoine

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Approved by the Examining Committee:

Professor Bülent Yener, Thesis Adviser

Professor Boleslaw Szymanski

Professor David Spooner

Rensselaer Polytechnic Institute
Troy, New York

April 2015
(For Graduation May 2015)

CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to thank Professor Yener for advising me through my master's thesis. Jeremy Blackthorne for introducing me to this area of research. Dan Guido and Alex Sotirov from Trail of Bits for providing me technical resources.

ABSTRACT

Given the rise in popularity of cloud computing and platform-as-a-service, vulnerabilities inherent to systems which share hardware resources will become increasingly attractive targets to malicious software authors.

This thesis first classifies the possible mediums for hardware side channel construction. Then we construct potential adversarial models associated with each. Additionally, a novel side channel is described and implemented across the central processing unit using out of order execution.

Finally, this thesis constructs seven adversarial applications, one from each adversarial model. These applications are deployed across a novel side channel to prove existence of each exploit. We then analyze successful detection and mitigation techniques of the side channel attacks.

1. INTRODUCTION

Modern cloud computing services employ the use of shared hardware between different virtual machine instances in order to increase efficiency and decrease cost. However, when hardware resources, such as the cache [1, 2, 3], are dynamically allocated to different virtual machine processes, a single virtual machine can continuously measure artifacts from its virtual allocation of a single hardware component and note changes it did not cause.

This allows the resident of one virtual machine instance to detect changes in the system caused by another virtual machine located on the same physical hardware. This information leakage allows an adversary to construct a side channel attack used to either record coresident behavior or communicate with a process purposefully generating these system changes [3, 4].

In covert side channels, a receiving software process is used to measure information from the unintentionally accessible hardware component. In the same way, a sending, software processes, located on a different virtual machine, provides this information across the hardware resource. In this way, the privacy provided by virtually allocation is bypassed.

The first section of this thesis presents a classification of the different shared hardware resources which may be exploited by a virtual machine process. We attempt to organize the different characteristics of each hardware medium to acquire the transmitting features used in a side channel construction.

Using these features, we analyze and compare the varied aspects of side channels built across different mediums to extract successful and limiting features.

Using these shared hardware mediums, this thesis then presents seven primary malicious attack models which may be applied across side channels built on each outlined hardware mediums. The distinctions between each are made according to the methods in which the processes attempts to attack co-resident virtual machines over a side channel, with the intent to extract or alter system information. Together, the malicious attack models encompass all malicious channel behaviors. Under each model, specific malicious processes can be categorized.

Each adversarial model is then analyzed for its potential across different hardware side channel architectures, presenting a formalization based on the transmission and reception functions of the channel. Cross applying the set of malicious program models on the set of hardware mediums, outlined in the first section, results in all pairs of attack vectors. We then analyze each pair for its applicability in real world cloud computing environments.

Finally, we implement a subset of the malicious program model, hardware medium pairs, targeting all pairs which use the central processing unit as the exploited hardware medium.

Using the processor, a novel side channel is constructed. Specifically, out of order execution is exploited to transmit and receive signals. We explore the use of this information as a channel through which malicious programs can operate covertly. Provided a sender,receiver, and message encoding schema we can create an effective attack vector with calculable success.

We discuss each malicious program implemented across the processor side channel in order to acquire insight into the scope of a hypothetical adversary on a cloud computing infrastructure.

2. HARDWARE MEDIUMS IN CLOUD COMPUTING INFRASTRUCTURE

Hardware side channels in cloud computing infrastructures rely on shared physical resources as the medium through which information is transferred. To do so, different variables unique to the targeted hardware component may be queried, measured or altered by the side channel processes residing on a virtual machine instance.

As the information gathered is specific to processes using the queried hardware component, a malicious virtual machine instance may only leak information from other co-resident virtual machines instantiated on the same shared hardware. Any information gained about a targeted, coresident VM may be polluted with noise caused by any number of benign coresident VM's.

This section attempts to first detail the hardware architecture found in cloud centers, categorize the hardware mediums found in these infrastructures, and cursory explain how side channel processes use them to transmit or receive information.

2.1 Standard Cloud Computing Architecture

In the industry standard for cloud computing infrastructure, a large physical hardware system is virtually allocated across a multi-core platform. This virtual allocated of hardware resources increases the efficiency of large scale computation for infrastructure as a service (Iaas).

As a result, multiple virtually allocated machines share processor time and memory space on a mutually held physical machine. For the sake of this thesis, the assumed architecture of the physical machine will belong to Intels Xeon family.

Depicted in Figure 2.1, is the physical organization of the hardware components which comprise a standard cloud computing server. Previous research done in the field of cloud security and hardware side channels have focused on this model as it best represents the largest market share for Infrastructure-as-a-Service (IaaS) [5, 6].

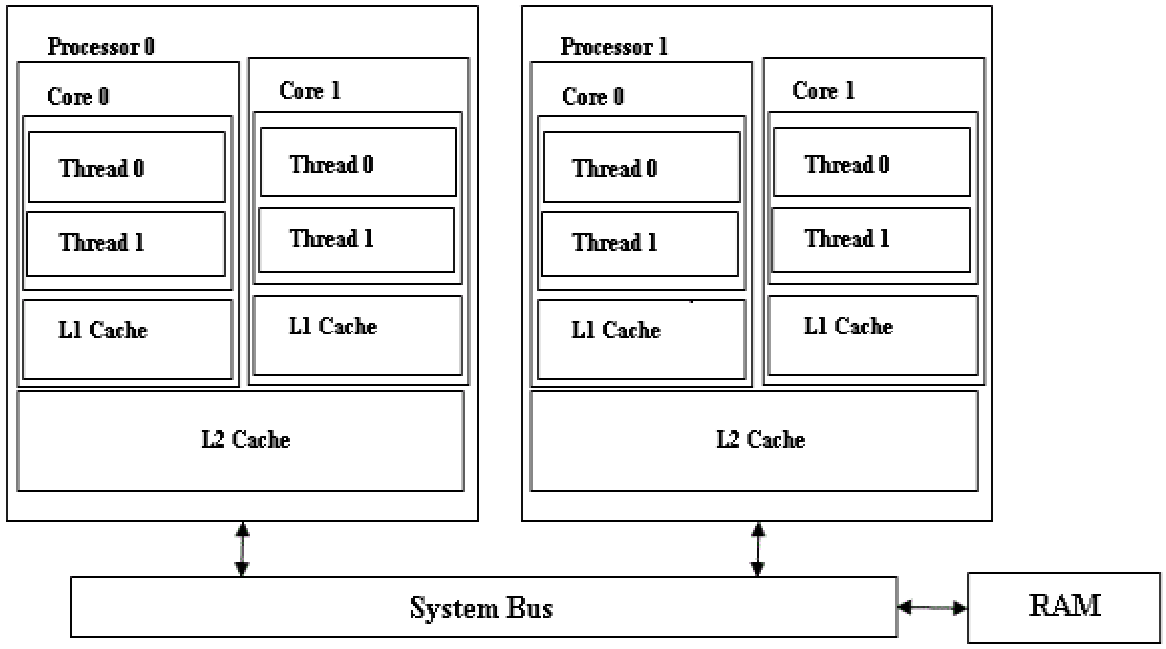


Figure 2.1: The hardware organization of the xeon family multi-core processors

Specifically, this hardware model includes a shared L2 cache across multiple cores and, in some units, an larger L3 cache shared across all processing units (CPUs).

In modern, multi-core systems an increased number of multi-tiered caches are added to overall reduce latencies incurred by time consuming queries to main memory. Such latencies are measurable on the order of several hundred nanoseconds. On average using this structure, for each memory access query, the processor saves time on a scale of two to three orders of magnitude.

Additionally, the multiple cores and processors allow for multi-threaded applications to be executed simultaneously. The diagrams depicted in Figure 2.2 represents how the organization of cache levels for multi-core processing units can vary across hardware provider; the Intel Core Duo, Xeon will be the reference point for this thesis's discussion of security and exploitable side channels.

The Intel Core Duo, Xeon processor contains an even share of shared and unshared memory. Shared memory between processors leads to contention of cache tiers used to store data. For programs running multiple threads with algorithmically complex operations across multiple cores, these shared cache tiers are optimal. Programs which do not gain much efficiency from shared cache include those which run simultaneous operations on exclusive sets of data.

In cloud computing environments, each virtual machine instance of a customers machine on the physical hardware unit is managed through a hypervisor. This hypervisor acts as a resource scheduler to dynamically allocate the performance equivalent of a single, full processing core (CPU) for the duration requested. This allocation may span across several cores and cache units on the same server if other co-located processes require more of a resource.

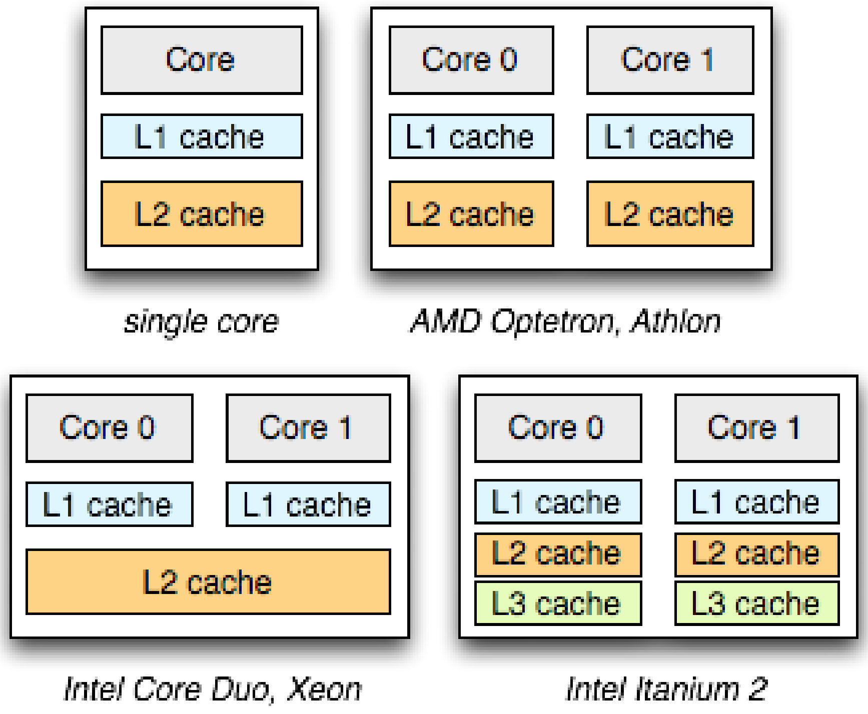


Figure 2.2: The differences in cache architecture between industry multi-core processing units

Multiple virtual machine instances allocated simultaneously are not guaranteed to be co-located on the same shared server. The notion of co-location is the primary element in the success of any hardware based, side channel attack vector [7, 4, 8]. Different hardware components of a server, applicable to side channel attacks, shared virtually and modifiable by the user, are discussed further in the section below.

2.2 Properties of Shared Hardware Resources Used in Side Channel Attacks

The first required property for extracting information from across a cloud computing side channel is location. The third part provider must provision the computing resources for the malicious and targeted virtual machine on the same physical unit. The adversarial instance must reside on the same server as the target instance in order to extract target information from the hardware component. For this reason, the most thorough security protection against side channel attacks implemented over a trusted third-party cloud infrastructure is the complete physical isolation of all equipment and hardware for the guaranteed instances. This thesis assumes that there is a single, unique hypervisor on each server to dynamically divide resources between virtual machines. For attacks targeting cloud computing, the use of the Xen hypervisor duplicates the cloud environment for all side channel research discussed below.

A product of commercial cloud infrastructures, is the randomization of virtual machine placement adding an additional layer of security. There is only a percentage of chance that the target instance and the malicious instance are placed on the save physical server containing the necessary multi-core microprocessors. To construct a fully operational side channel attack, we first determine if the controlled virtual machine is co-resident with the target. If not, a new instance may be created and tested. This technique is repeated until co-residence is determined. The important function of this technique is the ability for a virtual machine to realize it has achieved co-residency.

The research done by RSA labs and University of North Carolina Chapel [7] introduced the HomeAlone system which allows a virtual instance to verify its exclusive residency on a physical machine. There technique may be used to determine if a virtual machine is co-residing with another and thus susceptible to a side channel attack, specifically through exploitation of the shared L2 cache.

For their defensive purposes, a virtual machine verifies that there is no foreign instances residing on the hardware. This machine pre-arranges for the ́friendlyínstances to withhold from cache usage in a pre-determined time frame. During this time, the defensive machine measures an artifact of the shared L2 cache. Any unexpected activity divulges the presence of an unwanted resident. There are rare limitations to this model, such as an opponent which can withhold cache usage during the period of measurement. However, the general application of this technique, using a variety of heuristics, is highly effective.

Another method used to establish co-residency, a consumer is allowed to purchase a virtual instance on demand and the cloud service provider multiplexes many instances across shared hardware [4]. In this work, various side effects of the multiplexing system, such as networking speed, proximity and addresses, are exploited to verify co-residency on the same server with a high measurable accuracy. This work found that purchasing an instance immediately after the target, in the same geographical region, highly increased the probability of sharing resources.

In figure 2.4, a representation can be seen of the co-residency of n virtual instances in a cloud computing server. Also depicted is the allocation, based on client operations, of physical resources by virtualized slices of memory and processor time. The various shared and un-shared cache levels are not shown in this simplified diagram.

The consideration of these two methods does not exclude other potential methods to determine co-residency but merely highlights two broad techniques, measuring hardware and network heuristics. Once the co-location of the adversarial and the target virtual machine instances can be verified, the side channel attacks discussed in the subsequent sections can be constructed.

The second property, required for implementing a side channel across virtual machines, is shared memory and cores, multiplexing. In cloud environments however this is enabled by default as it is a more profitable setup, allowing the provider to meet the unique constraints of the virtual machines computation. The Xen hypervisor must be set to share time-slices of the available CPUs between all virtual guest instances on the server, potentially overbooking the amount of virtual CPUs. As this is the current standard in cloud computing infrastructure, for the scope of this thesis, the Xen architectural model will be assumed.

A feature of this model is the dynamic interplay between memory retrieval, caching and instruction execution slices of the hardware components. When these resources are shared, a single instance can now dynamically affect and react to the surrounding environment; an environment which contains other virtual machine instances.

The third property required for implementing successful, virtual machine side channels is that the channel activity should have a low delta from any benign user activity.

While this property is not a requirement for a functioning channel, it indicates towards the deployability of the side channel attack across a live cloud computing infrastructure. A successful channel will use operations that are both non-invasive and non-unique. The invasiveness can be measured in excess resources used outside of any normal operating levels. For example, a system which conveys a message between two virtual machines by filling the entire cache would be considered highly invasive. The reality of this channel on a real system communicating without detection is improbable in that the other co-located machines, deprived of any space in the cache, would catch the attention of the service provider. Limitations on cache use would effectively mitigate the side channel.

The ideal channel must also be non-unique. For example, a channel which relies on routinely using a certain cache address or spiking the processor's load demonstrates behavior which is highly anonymous. The ideal channel, to avoid rapid detection and hypervisor resource access rules, uses a commonly accessed hardware medium. A component which supports more active and sporadic behavior allows the channel activity and resource accesses to become statistically insignificant.

If these three primary system properties are met for any chosen hardware component and side channel attack, then it has the potential for successfully relaying information when deployed across a real world cloud computing system.

2.3 Analysis of Transmission and Reception Techniques used across Side Channels

Each physical component which is shared between virtual machines is potentially exploitable for use in a side channel, with varying degrees of success. In order to better analyze the possibilities, we consider all hardware components of a server.

This section discusses each hardware component in order to assess what feature provides the transmission and reception mechanisms. These features, specific to the medium, make up part of the side channelś characteristics, effecting its potential. In figure 2.5, the major components of the larger computational environment are shown. These are the different units which will be referenced in this thesis as distinct transmitting mediums.

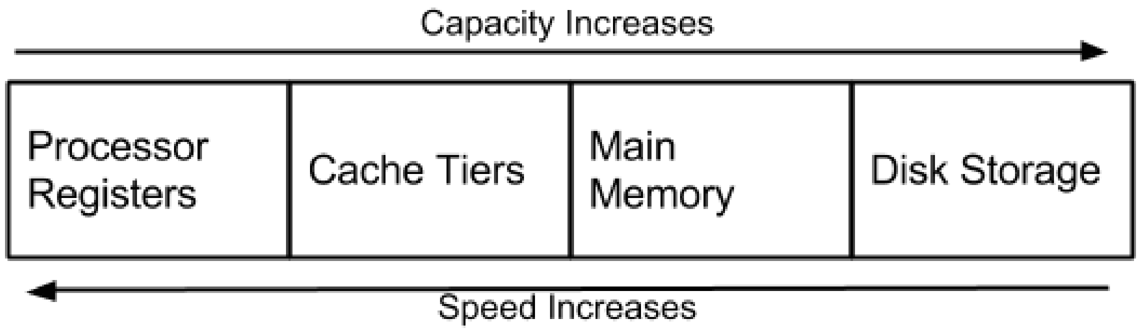


Figure 2.3: Memory hierarchy of related hardware components

Figure 2.3 shows a standard memory hierarchy of four hardware units used in memory management. They are shared between the co-residing virtual machine instances. As speed increases, capacity decreases. The majority of side channels exploit this inherent speed characteristic.

A single virtual machine measuring data retrieval times can discover on which component of the hierarchy its information is located. This capability is in fact not meant to be hidden and is legitimately used by many programs to optimize operations. However, when hardware is shared, it may become a mechanism to determine location of not only personal data but also the behavior of the co-residing virtual instances.

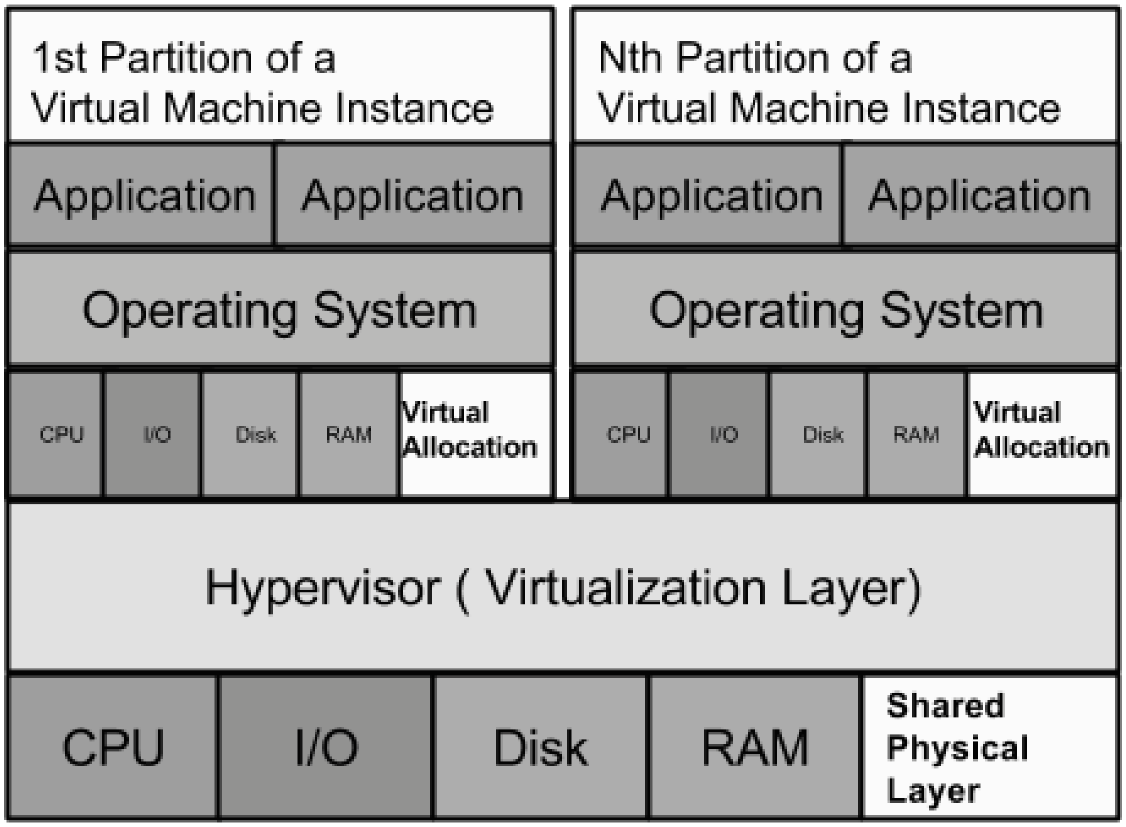


Figure 2.4: Physical resource allocation between virtual instances on a cloud server

2.3.1 Central and Graphics Processing Units as a Side Channel Medium

One of the shared resources, used in side channel attacks, is the processors. This may include the central processing unit, the graphics processing unit, or both as long as the graphical processor is allocated for the purpose of general purpose computing.

The use of the graphics processor to increase computation speed makes its use as a potential side channel in cloud computing even more relevant. Specifically, the graphics processor unit (GPU) has a massively paralleled architecture consisting of smaller, more efficient cores designed for handling multiple tasks simultaneously making it better fit to handle high loads. Using this processor over the central processing unit makes sense in multi-cored, parallel computing.

An information leak from the use of GPGPU (General Purpose GPU computing) is exploited to leak an AES key through research by Roma Tre University, [9], which exploits the resource managers necessary for GPGPU. The resource managers, CUDA (by NVIDIA) and OpenCL (by AMD), are used to interface with the graphics processor and to direct its use for computation. These platforms generate system artifacts in speed and memory which may be leveraged as a side channel by an attacker. Specifically, the CUDA architecture provides the system with shared memory, global memory, and registers available to the executing processes. This leads to cross virtual machine contention of these resources, resulting in information leakage between instances.

The central processing unit has been widely researched as a vector for side channel attacks in cloud infrastructure. The same methods to detect changes in system time and memory can be used for sending signals across the CPU as with the GPU.

A side channel over the central processing unit is developed in research by Princeton University [10] where the CPUs functional units or FUs, are exploited through a simultaneous multi-threaded process (SMT). Additionally, basic timing attacks are shown across shared processors[11] .

A single central processing unit has a set of functional units which are to be dynamically allocated to each process running as a SMT thread every cycle. The basis of the covert channel lies in these shared functional units. One process intentionally alters its use of the functional units in predetermined time intervals in order to interfere with other processes. When this resource contention occurs, a malicious process is able to discern artifacts from the environment.

The implementation of this side channel relies on measured time, the time it takes for a process to be allocated the appropriate functional units. Execution time will either slow or speed up depending on the state of the allocation. This makes measured time the transmitting vector for signaling between the isolated virtual machines. A proposed security defense against this channel is a selective partitioning solution in which resource sharing is minimized, adding noise and overhead to the system.

For both the graphical and central processor, time, as an effective measurement of resource allocation, enables information leakages across shared hardware. Side Channels developed using this technique are termed covert timing channels and may be constructed across hardware mediums besides the processors.

2.3.2 The Cache Tiers as a Side Channel Medium

There are three tiers of cache common in cloud computing servers, the L1, L2, and L3, each with varied memory space and distance from the the processor. We recognize that Amazon cloud services now offer premium C4 instances which run on Intels latest Xeon E5-2666 v3, Haswell, processors.

This new server provides an additional fourth tier of cache memory, L4. However, for the purposes of this thesis, all specific discussion of cache will address the common architecture only including L1, L2 L3.

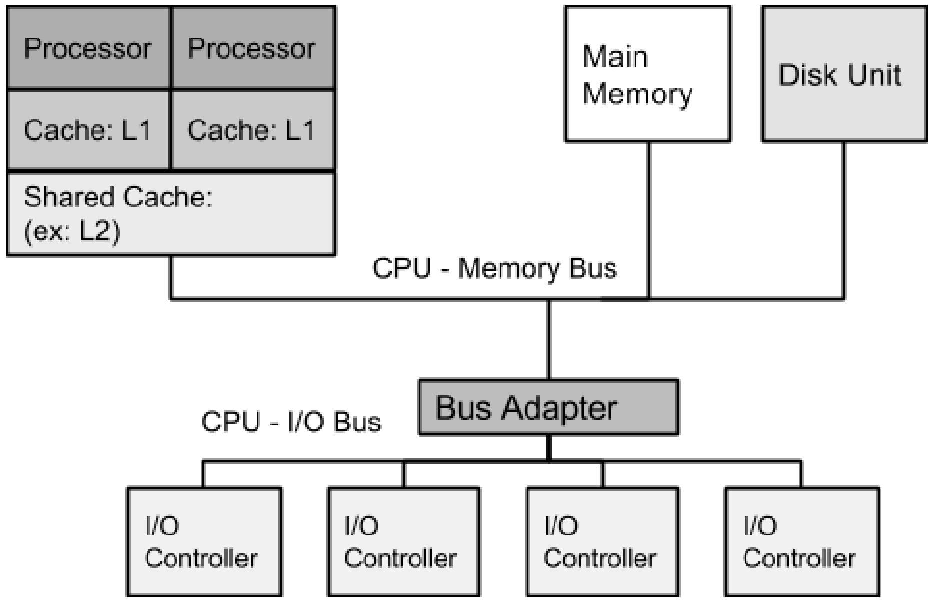


Figure 2.5: Hardware component organization on a multiprocessor server

As these tiers only differ in size and location, cache based side channel techniques developed against specific tiers of cache apply to all tiers with varied speeds.

In general, cache provides high speed memory access for the processing unit as an intermediary stage before much slower memory requests to main memory are required. The different tiers are labeled according to speed, L1 being the fastest and closest to the processor, and LN being the slowest and farthest away.

L1 provides the quickest and most expensive memory access, as it uses static RAM or SRAM cells. Additionally, L1 cache tiers are tied to a single core on standard multi-core processors as can be seen in Figure 2.5.

The L2 cache tier has become private, dynamically allocated memory, focused on a single processor and its set of cores. Similar to the dynamic partitioning and architecture of the L2, the L3 cache tier acts as a shared pool of memory common to all processing units on a system-on-a-chip (SoC). Higher quantities of L3 cache, and L2 cache, provide faster shared memory for virtual machines and multi-threaded (SMT) applications.

An adversary can take advantage of these shared resources through contention, memory probing, or preemption in order to leak valuable system information with the intention of forming a side channel.

While L1 cache is tied to a single core, virtualized allocation of a single core between processes or machine instances means the L1 cache is as well. This dynamic sharing becomes a mechanism to measure system artifacts.

Specifically, the L1 cache is vulnerable to a prime-probe technique. This was used to construct a covert timing channel to exploit the L1 cache and extract an ElGamal decryption key from the co-resident, target virtual instance [12]. A simple program can be implemented which fills a section of the cache associated with a given offset on each page, a technique called priming. A listening, malicious virtual machine would then query this cache section of the L1 cache page, a technique called probe. The time it takes to querying the cache acts as the transmitted signal.

The addition of different methods, such as preemption of the resource scheduler, add speed and accuracy to the channel. However, for all variations of side channel methods, time remains the fundamental measuring unit of a single bit.

Constructing side channels across the L2 cache vary in technique and precision, yet they depend on a prime-probe or preemption technique and use time to distinguish system changes.

A variant on the prime-probe transmitting technique, forcing cache misses can be used by a malicious virtual instance across the L2 cache tier to leak useful environment information [3].

A malicious process may exploit the translation lookaside buffer, TLB, and its limited mapping into the L2 cache. This TLB contains only a fraction of the addresses needed to decrease the speed of translating virtual addresses. Accessing all pages known to be in the L2 cache forces a TLB miss. This L2 side channel has a lower bandwidth than a comparable prime-probe channel across the L1 cache given the difference in proximity to the processor. This utilization of timed memory queries across the L2 cache is identical to timing channels across all cache tiers. The L3 cache offers shared memory pages between processors and dynamically resized virtual cache allocations. These shared memory pages creates an attack vector to record useful environment artifacts [13].

One such constructed channel relies on pages, shared on the L3 cache tier, and successfully leaked the private key from the GnuPG implementation of RSA [13]. A malicious virtual instance flushes a line of memory from the cache hierarchy and waits a set time frame. Then the malicious process queries the line and times how long the response takes.

If the victim instance queried the line, then the attacker now knows exactly what data the victim was accessing. This channels bandwidth relies on the granularity of the chosen time frame, sacrificing accuracy for speed.

For cache tiers, L1, L2, and L3, the measurement of time, specifically memory access time, is the key reception mechanism. Optimizations of the query algorithm may increase bandwidth, but essentially they still rely on querying time or values.

2.3.3 The I/O, Memory and Other System Buses as a Side Channel Medium

The system buses are necessary for high speed data transfers between different memory or computational units. There are several different times of buses, as can be seen below in Figure 2.6. Each bus transports different types of data via its bus line and connects the appropriate units.

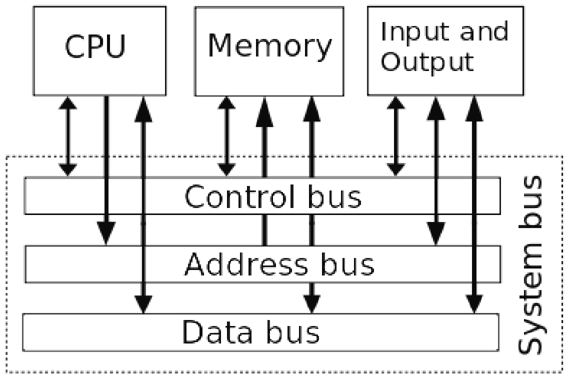


Figure 2.6: Primary system buses and their functionality

These buses are required for physical communication between computing units and are shared across all virtual machine instances on the server. A side channel can be built using any of the shared buses to leak system behavior.

For example, a side channel may be constructed using memory bus contention [14]. This channel measures possible access to the memory bus as the unit to record a binary signal. To force a signal, atomic instructions from the x86 instruction set may be used, by a transmitting application, to block uncached data access through the memory bus.

Effectiveness of this channel relies on locking all processors out of an essential hardware component, the memory bus. This greatly increases the possibility for detection in contrast to the timing based side channel used over other transmission mediums. As many processes accessing the memory bus increase contention, the resulting channel is subject to frequent interference and a great level of environment noise.

2.3.4 The Main Memory and the Dynamic RAM as a Side Channel Medium

The main memory is physically located much farther from the processors than the cache resulting in a larger access latency. However, a side channel may be developed with moderate success across different segments of the main memory.

Measurement of a pre-determined data segment in shared main memory reliably leaks environment information. The mechanism used to force contention of this shared data segment varies between specific implementations of the channel.

A side channel may be constructed to target memory paging one systems supporting SMT. Paging occurs as a result of a process requiring more memory than what is available. The system scheduler then pages these processes and corresponding data between main memory and the disk. A transmitting process can force paging by filling the main memory. A listening process can measure the shared address space allocated by the system. This measurement leaks the intentional memory use of the coresiding process which is then mapped to a binary signal.

A side channel can be implemented to exploit main memory on cloud servers, specifically the dynamic random access memory or DRAM [15]. A malicious virtual process may measure a value from a memory address on the DRAM. This value may or may not be accessed immediately, leaking whether or not that specific memory address was used by another process. The success or failure of a query may then be mapped to a binary signal. In channels built across this hardware unit, memory contention is used as the transmitting mechanism.

2.3.5 The Hard Disk, Including the Disk Drive and Virtual RAM as a Side Channel Medium

Co-residing virtual instances effectively share physical hard disk space while being virtually isolated. These virtual machines effectively force or monitor hard disk space contention to transmit a signal. By measuring file read times, a process can record whether or not the file was operated on by another process and form a successful side timing channel. This method may be used for transmitting and receiving and is a variation of the prime-probe technique [4].

Added optimizations to timing channel, such as symbol and frame synchronization between processes, resulted in a one thousand times increased the bandwidth [16].

The basis of this channel relies on rapidly accessing files in a known locations on the hard disk. This forces file contention with other processes accessing them. A simple side channel is constructed from this contention. A mutli-threaded, transmitting process quickly accesses a chosen set of files while the receiver tries to access them with varied read times. The physical disk drive segments, once filled, cannot be as easily re-purposed as the temporary memory units. Additionally, any disk read times will be slower inherently than read times of the other memory units as it is physically farther from the processor. These two characteristics of the disk drive results in a lower potential bandwidth.

Exploitation of any physically shared, but virtually allocated resource can lead to successfully measuring information from a computation environment. When these measured changes are caused intentionally by a co-residing virtual instance, reception of a transmitted message is possible and the construction of a side channel.

2.4 Classification of Hardware Units and the Transmitting Methods Used Across

Each hardware unit's functionality results in specific constraints specific which affect how a side channel can be constructed across it. The data, found in Figure 2.7, outlines the mechanisms through which transmission and reception of data occurs.

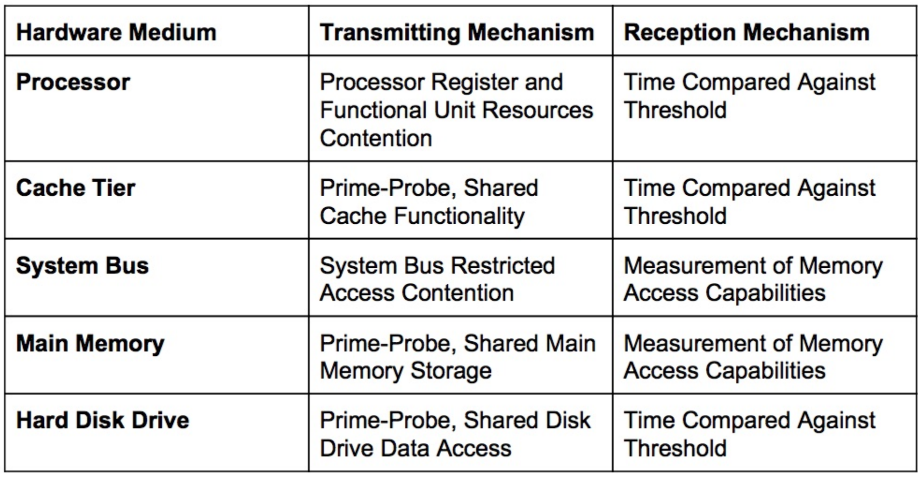
The category, Transmitting Mechanism, describes a side channels technique to force a behavior in a given hardware medium. Under this category, there are two primary techniques, resource contention and prime-probe. Resource contention occurs with the forced contention of a hardware units shared functionality or storage capacity. When the transmitter controls a segment of a unit's resource, the receiver may measure limited access to that resource. Prime-probe occurs when a transmitter manipulates data stored in a segment of the resource.

The category, Reception Mechanism, describes a side channels reception technique to measure behavior from a given hardware medium. Under this category, there are two primary techniques, measuring time and memory access. Measuring time to record a signal relies on an arranged time frame in which the measurement will be taken as well as an expected measurement value. This means that any noise in the system which increases latency in resource access will negatively effect the signal. Measuring memory access to record a signal has less susceptibility to system noise. A receiving process can than access this memory segment and analyze the received data.

Cloud computing server architecture shares physical resources between virtualized instances which include five major, distinct hardware units listed in Figure 2.7.

Each of these units may act as mediums across which co-resident, virtual machines construct a side channel using a transmitting and receiving mechanism to communicate. Unique programming implementations of transmitting and receiving mechanisms may differ for each physical unit. The table in Figure 2.7 lists which mechanisms apply across different physical units.

This section provides an organization of possible side channel hardware mediums as well as fundamental transmitting and receiving mechanisms which may be applied in specific malicious applications.



Hardware Medium	Transmitting Mechanism	Reception Mechanism
Processor	Processor Register and Functional Unit Resources Contention	Time Compared Against Threshold
Cache Tier	Prime-Probe, Shared Cache Functionality	Time Compared Against Threshold
System Bus	System Bus Restricted Access Contention	Measurement of Memory Access Capabilities
Main Memory	Prime-Probe, Shared Main Memory Storage	Measurement of Memory Access Capabilities
Hard Disk Drive	Prime-Probe, Shared Disk Drive Data Access	Time Compared Against Threshold

Figure 2.7: Mechanisms used to transmit and receive across different hardware units

3. MALICIOUS APPLICATIONS ACROSS THE SIDE CHANNEL MODEL

With the increasing number of side channel attacks developed across the different hardware components shared in cloud environments [17, 18, 19], this section attempts to classify potential attacks specifically tailored to exploit these side channels. The prerequisite for any side channel is the ability to transmit and receive information by exploiting specific hardware components - such as the cache, the processors, etc. The attack models described must reduce down to a transmission and a reception mechanism.

A "transmitter" alters a exploited hardware component in a repeatable way, generating an artifact which the receiver measures from the same medium. For example, to generate a signal across a cache based side channel, a transmitter may alter the data available in the cache. A "receiver" can then query a targeted location and meaningfully compared results from this query to expected ones.

This simplified model of a transmission and a reception process reduces the possible set of malicious functions which could be carried out across a chosen side channel. This section focuses on categorizations for malware types contained in this possible set. While prior research will be used to exemplify the categorizations made, this section will not pursue a detailed analysis of all malware types, but only those types relevant to side channels. With a typology of malicious behaviors which exploit side channels, it is possible to further analyze documented attack vectors as well as potential novel attacks.

This section assumes a symmetric multiprocessing system, a modern virtual machine manager, a exploitable hardware side channel and an optimized algorithm to maximize bandwidth across the channel. The categories described are abstracted from specific side channel implementations so as to be applicable to a larger set of processes.

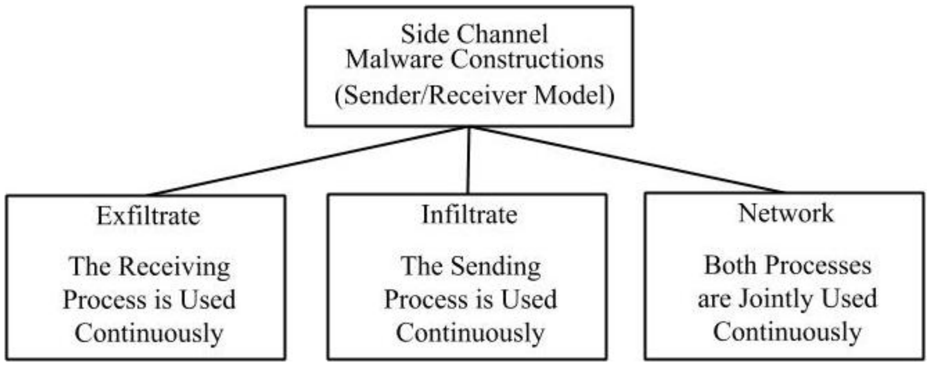

Side Channel
Malware Constructions
(Sender/Receiver Model)
Exfiltrate
The Receiving Process is Used Continuously
Infiltrate
The Sending Process is Used Continuously
Network
Both Processes are Jointly Used Continuously

Figure 3.1: The three permutations of the two processes used in side channel constructions

3.1 Characteristics of Malicious Side Channel Use

This section presents three classes to organize the malicious behavior which is possible across side channels. These categorizations are meant to capture all types of malicious applications, both traditional and novel, which rely on a hardware based side channel.

The "exfiltrate" and "infiltrate" categories, seen in Figure 3.1, have a distinct one way property. This means that either the sending or the receiving process of the side channel is under continual operation in the system environment. The remaining process is not the focus of the application and is either never used or used only once, independent of the surrounding system activity.

The “network” category has a bi-way property and relies on continually operating sending and receiving processes in order to effect or record system activity.

The first category, exfiltrate, refers to malicious applications which are constructed with the intent of exfiltrating system data or information. Processes in this category rely on the receiving process, where information is received through reaction to changes in the environment. Continuous operation of these reactive applications allow the adversary to effectively record the system changes over a given time period. When there is no transmitting process running, the receiving processes is reacting only to the artifacts of the targeted co-resident processes.

This record of shared hardware activity over a targeted medium can be mapped to known patterns, exfiltrating co-resident operations and information. For example, a coresident virtual machine running encryption operations will require cache resources in a pattern mappable to the encryption operations. The co-resident, receiving process setup to operate over a cache based side channel will react in a pattern similar to those of the victims encryption and can subsequently exfiltrate the encryption key [3, 20].

This category also encompasses malicious application functions which include a minor role for a transmission process. In these situations, the transmitter pre-agrees on a time frame with the receiver and is co-resident. The transmitter affects the hardware medium, creating an artifact so that the receiver can record a unique, coordinated signal. However, as the transmitter has no method of noting system responses, it cannot adjust its broadcast. This gives a one time property to the transmitting process.

The infiltrate category contains applications in which the main operating process is the transmitter and not the receiver. These applications continually transmit activity or data into the shared hardware. The effects of these operations are then seen in the reactions that the non-colluding, targeted virtual machines have in response to the altered environment.

This category also encompasses malicious application functions which include a minor role for a receiving process. The reception process operates once and without any knowledge before or after of the transmitter. One example of a side channel attack, used to determine if one malicious virtual machine is colocated with another, relies on a continual transmitter and a one way receiver [4, 7].

The third category, network, jointly operates both the reception and transmission processes continuously. A common example of a malicious application modeled in this way is the construction of a communication channel. Two colluding, co-residing virtual machines can covertly communicate to one another. Each of the communicant applications have both a transmission and reception function which operate in a known time frame. This joint sender-transmitter programs may either affect the environment to convey a message or react to the environment to receive a message [14].

These three main categories shown in figure 3.1 attempt to organize the different primitives from which an adversary may model a malicious side channel applications. The distinctions between each category are based on different combinations of transmitting, receiving behavior. These distinctions ultimately determine the possible functionality of the malicious application.

3.2 Exfiltration Applications

The ”exfiltrate” category encompasses side channel applications which leak system information. Malicious programs exhibiting exfiltration behavior continually operate a receiving process. In some cases a transmission process is used once to meaningfully alter the state of the shared hardware.

3.2.1 Continuously Active Receiver, No Transmitter

A hardware side channel application which contains a continuously active reception process, and no transmitter, will react to the changes in the targeted hardware component that it exploits. Exfiltration applications record system state over time. This record of system information may be analyzed to understand the activity or state of the coresident virtual machine. A set of common malicious application types which may utilize this single reception structure is presented.

Cryptographic key theft is the most common application documented in literature and targets a hardware side channel using a single reception process. Malicious programs of this type use a receiver to record the changes in the targeted hardware component, most popularly hardware cache. Patterns recorded may be mapped to known patterns which result from different encryption operations, such as multiplication, to leak information about the key. Accounting for noise in the recorded pattern increases the likelihood of successfully retrieving a cryptographic key [21, 12].

Activity logging refers to the monitoring and recording of co-residing virtual machine behavior. A specific demonstration of this type of malicious functionality is keylogging [22]. When built across a side channel, the receiving process acquires artifacts from the user activity in the coresident virtual machines.

Pre-acquired measurements of the affects a particular user activity has to the targeted medium allow the recording adversary to exfiltrate sensitive information about user behavior, such as keystrokes.

Malicious applications using a continuous receiver may also employ environmental keying across a side channel [4, 7]. Comparison between an expected record with the actual record of measurements taken from the targeted hardware environment, allows the malicious application to key the virtual machine's allocated hardware and decide how to execute. Environmental keying has many uses, such as enabling hardware specific execution, avoiding emulation or other security programs which by running on the same hardware, change the measurements taken from the environment.

3.2.2 Continuously Active Receiver, One Way Transmitter

A transmitting process is added to the receiving application model presented above. The addition of this transmitter, located in a coresident, colluding virtual machine, provides the functionality in the attack model to broadcast a signal through a shared hardware component. The receiver reads this pre-arranged signal and decides to take actions accordingly. As the transmitter cannot receive information from the system, it cannot adjust its transmission in response to external factors. We present a malicious application type which uses this single receiver, one way transmitter structure.

A transmitting application may act as a trigger for a receiving application through transmitting a broadcast signal by altering information on a shared hardware medium. The signal must be unique so that other, coresident applications may not generate processes which benignly alter the hardware medium in the same pattern. A receiving processes located on a colluding virtual machine will continuously record information from the targeted component until it receives this prearranged signal. After the triggering pattern is received, the application can then launch additional functions, such as process initialization or masking to dynamically avoid detection or hide intended information.

3.3 Infiltration Applications

The "infiltrate" category encompasses side channel applications which cause activity in or inject data into the targeted hardware component. Malicious programs exhibiting infiltration behavior continually operate a transmitting process. In some cases a reception process is used once to meaningfully alter the state of the shared hardware.

3.3.1 Continuously Active Transmitter, No Receiver

A hardware side channel application which uses a continuously active transmission process, and no transmitter will generate disturbances in the targeted hardware component. This application will influence how the system allocates resources between coresident virtual machines as well as what data is queriable from different hardware components, affecting the performance of the coresident processes. Depending on the function of the malicious application, the injected information may be tailored to target specific performance changes.

"Hardware denial of service" refers to the performance impact that the malicious transmitting process causes in the shared hardware component. Applications in this category use a transmitter to alter the data stored in a hardware component or to force prioritization [9, 10]. The specific function of the transmitting application may rely on a specific pattern of operations or data injected into the hardware medium, either to exploit the scheduling algorithm or to overwrite specific data stored in temporary memory. Accounting for noise caused by coresident operations and prioritization algorithms used by the scheduler increases the likelihood of successfully modifying performance of coresident processes.

3.3.2 Continuously Active Transmitter, One Way Receiver

In this category, a one time receiving process is added to the malicious application model which relies on a continuous transmission process. The addition of this receiver, located in a coresident, colluding virtual machine, provides a reception functionality in the attack model. The transmitter beacons continuously, leaving an artifact in a targeted hardware medium which may be read by the reception process. As the receiving process has no method for sending a response, it may only read the continuous signal from the system in the time frame that the transmitter is active. We present a malicious application type which uses this single transmitter, one way receiver structure.

This transmission and reception model may be exploited to determine virtual machine colocation. Applications of this type use a transmitting process located on a virtual machine residing on the targeted shared hardware. The receiving process can then move between different, allocated virtual machines at random, recording activity of a hardware medium from each one. When the receiving process records the prearranged signal, it can then assume it is collocated with the virtual machine running the transmitting process [4, 7].

3.4 Network Applications

The “network” category defines side channel applications which both transmit and receive signals across a targeted hardware component. Malicious programs exhibiting communicating behavior continually operate both a receiving and transmitting process, much like modern network endpoints.

3.4.1 Continuously Active Transmitter and Receiver

The implementation of a basic communication network across a hardware side channel requires that both communicating applications, running in collocated virtual machines, have joint transmission and reception processes. By avoiding traditional network-based detection methods, communicating side channels built across shared hardware have stronger security properties including covert transmission and increased privacy.

Command and control functionality relies on the bi-way communication possible through the use of multiple, coresident virtual machine applications. Applications of this type agree on a time frame and on a single, authority application. Specifically, botnets operate in this way, where each botnet node contains a joint transmission and reception process and communicates with the collocated, authority node [19]. Some functions of this authority node include choosing on important signals, time frames or protocol variables.

3.5 Summary of Three Architecture Models

Three categories are introduced to describe the malicious side channel programs which are structured with the intent to either exfiltrate, infiltrate or communicate across a shared hardware medium.

Different pairings of the transmitting and receiving processes in an application form the distinguishing factor used to categorize potential adversarial models. Under each category, specific application functionality and behavior types form a basis for sub-categorization. While these groupings do not attempt to exhaust existing attack models, they attempt to provide a view into potential malicious actions across a side channel using the constraints of the transmission and reception pairing. We hope this typology will initiate further discussion on the potential for traditional malware functionality to be applied to modern, cloud based, side channels.

3.6 Summary of Channel Architecture Models

– Malicious Attack Type:

- Exfiltration (Receiver Only)
  - $M_1 \in$ Cryptographic Key Theft
  - $M_2 \in$ Activity Monitoring
  - $M_3 \in$ Environmental Keying
  - $M_4 \in$ Triggered by Broadcast Signal
- Infiltration (Sender Only)
  - $M_5 \in$ Resource Denial of Service
  - $M_6 \in$ Determine VM Coresidency
- Network (Receiver & Sender)
  - $M_7 \in$ Command & Control Communication

4. FORMALIZATION OF THE HARDWARE SIDE CHANNEL MODEL

We formalize the two distinguishing factors of side channel construction, the hardware exploited and the processes used, to create a tuple representing distinct channel implementations. Each channel implementation represents an attack vector across which specific malicious applications may be deployed.

4.1 Models to Represent Hardware Side Channels

The first distinguishing factor is the specific hardware resource used to transmit and receive environment information. This shared hardware component is exploited by the sending and receiving processes.

Second is the three specific implementation architectures discussed in Section 4 - sending process only, receiving process only, or use of both processes.

Each of the three architectures is associated with specific attack models, also discussed in Section 4. They are listed according to which of the processes the attack model uses. For example, a communicating attack, M_7, requires both processes.

Distinguishing Factor 1

Hardware Side Channel Mediums:

- Processor
  - central
  - graphical
- Cache Tiers
  - L1
  - L2
  - L3
- System Bus
- Main Memory
- Hard Disk Storage

Distinguishing Factor 2

Channel Architecture S/R Models

– Malicious Attack Types Applicable:

- C_1 : Exfiltration (Receiver Only)
  - $M_1 \in$ Cryptographic Key Theft
  - $M_2 \in$ Activity Monitoring
  - $M_3 \in$ Environmental Keying
  - $M_4 \in$ Triggered by Broadcast Signal
- C_2 :Infiltration (Sender Only)
  - $M_5 \in$ Resource Denial of Service
  - $M_6 \in$ Determine VM Coresidency
- C_3 :Network (Receiver & Sender)
  - $M_7 \in$ Command-and-Control Communication Network

4.2 Tuple Representing a Hardware Side Channel

We take all possible cross-sections of Category 1 and Category 2. Each cross section contains a single element from each. This forms a set of tuples. Each tuple includes a single hardware medium and channel architecture S/R Model pair. This set provides a complete framework within which different channel attacks can be grouped.

A tuple represents a specific side channel across which a subset of the malicious attack models can be applied. Each application has specific characteristics and upper bounds for success.
C = Channel Construction =
{ Hardware Medium, Channel Architecture }

$C_1 = CPU, Receiver$

$C_2 = CPU, Sender$

$C_3 = CPU, Sender\&Receiver$

Each of the Malicious Attack Models can be applied across the channel constructions which implement the associated Channel Architecture S/R Model it is listed beneath above. While the attack models may be attempted under different channel architectures, success is hindered by the lack of the necessary sending or receiving processes.

For a fully functioning attack, a specific piece of malware that falls under the applied Channel Architecture Attack Model, represented by M_x, runs across the given C_x.

$$M_x \text{ x } C_x$$

Specific characteristics and properties are associated with each C_x running software, M_x. Implementation in the following section focuses on channels C_1, C_2, and C_3 which use the processor as the hardware medium and represent the three possible permutations of the sending and receiving processes. We then quantify different artifacts from each M_x associated with each of the three channel possibilities.

4.3 Channel Construction

The channel constructions C_x must have a pre-set time period in which an artifact is measured multiple times from the hardware. The average of these measurements can than 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 4.1, 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 C_x, 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, H_x, across which the C_x is built. This is an artifact of the time a single hardware measurement takes to collect by the process. For example, hardware mediums located further from the processor are most likely mediums which have longer minimum time frames as it takes longer for a single query to physically reach the component.

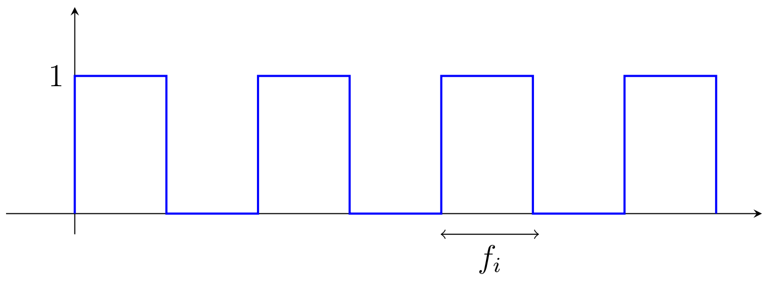


Figure 4.1: A bit stream, with each time frame and its average measurement representing a single bit

There are three channel constructions per each H_x, each which are implemented using one of the channel architectures. This set of C_x will have the same value for an optimal b_i and therefore of f_i given that these values are functions of the hardware component.

The reason for this invariability is that the transmission and reception processes iterate over code that is tailored to interact with a specific hardware medium. The code iterations do not depend on whether the processes are being used alone or jointly.

We show that the optimal f_i and b_i do not vary with channels of the same hardware medium. We do this by choosing to use the three channel constructions, represented in this thesis by C_1, C_2, and C_3 defined above, which are implemented using the central processor (CPU) hardware medium, H_{cpu}. Across these channels, variable malware types will be implemented to show what attack vectors are possible across the three different architecture models.

5. HARDWARE SIDE CHANNEL IMPLEMENTATION

We implement the three possible channel constructions, C_1, C_2, and C_3, across the central processing unit, H_{cpu}. To do this, we create a novel side channel by exploiting a function necessary in modern processor optimization: out-of-order-execution.

5.1 Out of Order Execution

In constructing the three channel types, out-of-order-execution must be exploited. To exploit this behavior, we create an algorithm which reliably forces out-of-order-execution occurrences across all shared CPU's. This implies that the algorithm itself must not overload the processor and get optimized off of the shared hardware.

Also known as dynamic scheduling, out of order execution is a direct result of processor optimization [23]. To increase processing power, modern computer architecture implements multi-staged pipelining, allowing for simultaneous execution of multiple instructions. Ideally, this occurs every clock cycle at full capacity, however hazards arise which degrade the overall performance time of the machine. One such hazard would be a delay caused by an instruction set which requires a great deal of cycles and there are other instructions require its output [24, 25].

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.

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 sent to it from higher level processes.

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 [25].

5.2 Exploiting Out of Order Execution

Certain reordering scenarios result in computations that are 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 this process can be illustrated as seen in Figure 5.1

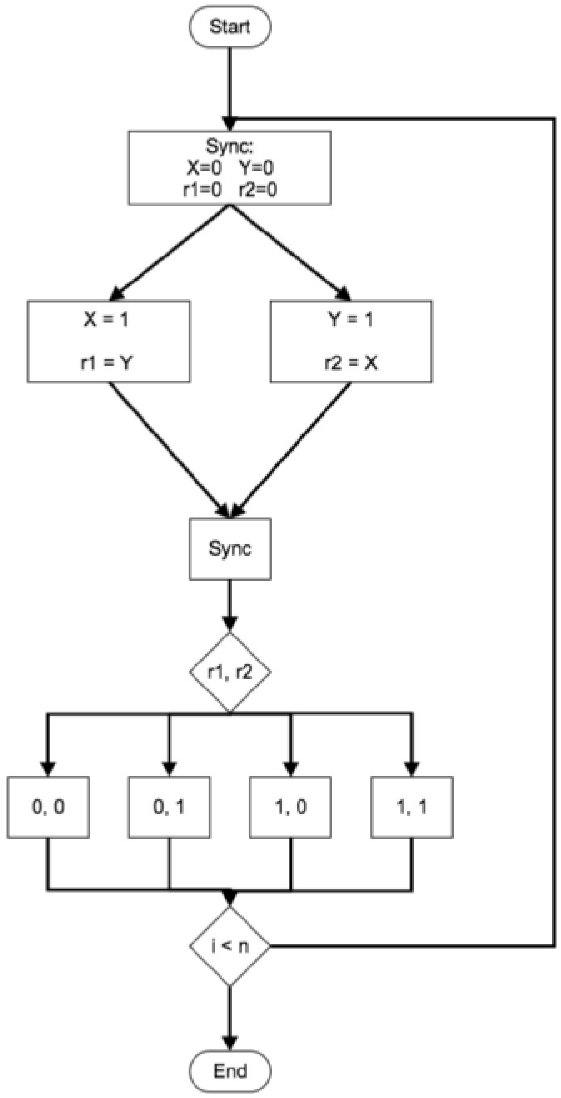


Figure 5.1: Two threads depicting possible results of the swap

The illogical output, $(0, 0)$ can be exploited as an unintended leaking of processor behavior. A program will record external environment changes by measuring the frequency of the four different outputs.

Iterating through this computation many times returns an average frequency of out-of-order-execution outputs divided by total number of outputs recorded, or number of iterations. Comparing this frequency against a baseline frequency exposes valuable system information of all processes running on a certain set of shared cores.

5.3 Transmitting and Receiving Processes

We construct the three channel architectures: exfiltrating, infiltrating, and network as separate side channel. To do this, a pair of transmitting and receiving processes exploit the shared central processing unit. The transmitter must force out-of-order-execution to occur and the receiver must record these occurrences.

The receiver is constructed using the method described above to record out-of-order-execution occurrences. A loop in constructed for each time frame which iterates over a single measuring function. This function contains the two threads used to record one of four operation results. After the loop is complete, the sum of out-of-order-execution results is divided by the total number of iterations to get a percentage. This percentage is compared against a baseline percent of out-of-order-executions to determine if the sending process is transmitting a high bit. The absence of a high bit received in a single time frame implies a low bit. Pseudo code representative of this process used to retrieve a single bit signal is seen in Algorithm 1.

The transmitting process forces the shared central processing unit to execute the operations of the sending processes two threads out of order in order to transmit

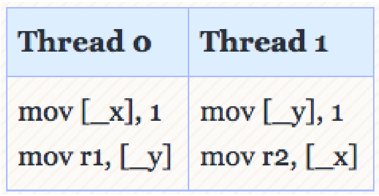
Algorithm 1 Receiver Pseudo-Code

```
procedure ReceiveBit
    $n \leftarrow$ # of iterations in time frame $f_i$
    $sum \leftarrow 0$                                        ▷ The summation of OoOE
    for $k = 0,\ k{+}{+},$ while $k < n$ do
        $X \leftarrow 0$                                      ▷ The initial variable values
        $r_1 \leftarrow 0$
        $Y \leftarrow 0$
        $r_2 \leftarrow 0$
        parallel $Z \leftarrow (X \parallel Y), r_n \leftarrow (r_1 \parallel r_2)$ do
            $Z \leftarrow 1$                                  ▷ Two threads setting variables in parallel
            $r_n \leftarrow \neg Z$
        end parallel
        if $r_1 \equiv 0$ AND $r_2 \equiv 0$ then
            $sum{+}= 1$                                       ▷ The (0,0) case implies OoOE occurred
        end if
    end for
    if $sum \div n \geq$ threshold % then
        return 1                                              ▷ A high bit is received
    end if
    return 0                                                  ▷ A low bit is received
end procedure
```

a single high bit. To send a low bit, the transmitting process simply refrains from operating, allowing the processor to execute the receiving threads in one of the three expected orders. The construction of this transmitter relies on a shared time frame, f_i, which is representative of the time it takes the receiver to complete n iterations in the ReceiveBit procedure, see Algorithm 1.

During this time frame, the transmitting process may repeatedly execute out-of-order-execution inducing assembly instructions. These are memory fence instructions, in x86, mfence, lfence, and sfence, used to force the processor to complete the time intensive transmitting process loads before the loads of the receiver.

This means that the processor optimizes the receiver operations by preemptively loading variable values needed to be stored before these variables are altered. In Figure 5.2, the second move instruction in both threads requires this targeted load of the variable value.



Thread 0	Thread 1
mov [_x], 1 mov r1, [_y]	mov [_y], 1 mov r2, [_x]

Figure 5.2: The store instructions in both receiver threads; the second move requires a reorderable load in the processor

In transmitting a high bit, the sending process alters the order of the receiver's thread instructions in the processor as can be seen in Algorithm 2. In transmitting a low bit, no interfering instructions are executed. Either operation happens continuously during the given time frame.

Algorithm 2 Processor Delayed Stores, Preemptive Loads

```
1: procedure ProcessorReorder                     ▷ assuming all variables set to 0
2:     parallel $Z \leftarrow (X \parallel Y), r_n \leftarrow (r_1 \parallel r_2)$ do
3:         load [$\_Z$]                ▷ loading value, storing in subsequent variable
4:         store [$\_r_n$]
5:         load 1
6:         store [$\_Z$]
7:     end parallel
8: end procedure
```

5.4 Construction of Seven Attacks

We construct a simplistic out-of-order-execution side channel. We use this to deploy an application from each of the seven attack models. The current CPU side channel construction is tailored to the contrived testing environment where there are only a few virtual machines running.

Using the requirements of a successful covert channel discussed earlier in this thesis, we present Figure 5.3 highlighting a complete attack vector. These stages include determining co-residency, a requirement for any hardware based exploitation. Next, the physical interference or observation of a specific hardware unit, below the L1 cache is listed. Then noise reducing functions applied to the received data to average away noise and other environment variables. And finally, the malware can eavesdrop from inside the receiving process based on information leaked from co-resident virtual machines.

For the sake of our implementation, we assumed that co-residency is pre-determined. Also, that a satisfactory noise canceling algorithm was used. These assumptions were implemented by reducing the total number of running virtual machines to 6, all of which were instantiated on a single Xen server using software configurations which reflect those of the Amazon Cloud Computing service[26].

The implemented side channel by adversarial virtual machines is comprised of a single sender and receiver as described above in Section 4.2. Attack models which require only an active receiver to operate fall under the exfiltration category described previously in this thesis. Similar categorization of attack models as either infiltration or network programs holds true for models requiring only a sender or both processes.

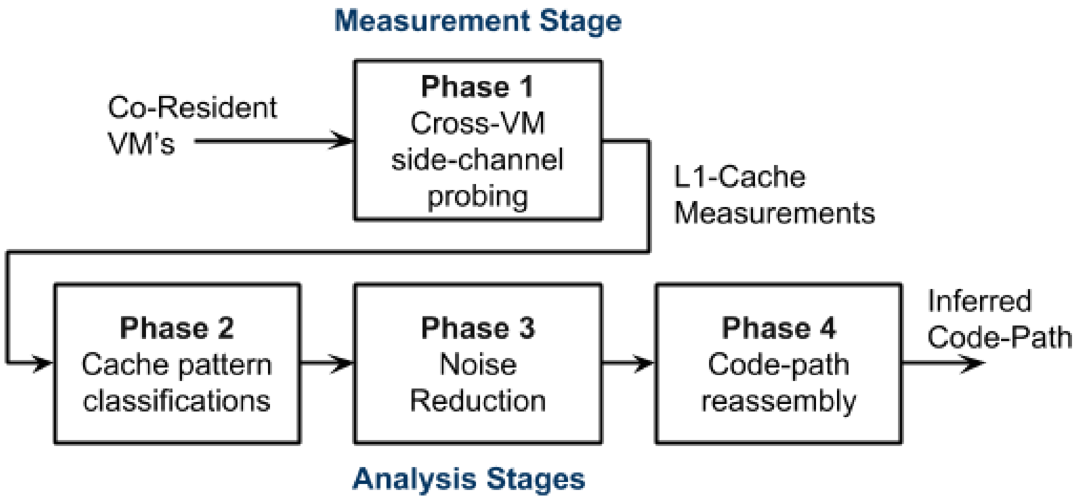


Figure 5.3: A complete diagram of the different stages of a deployed attack in a live, cloud computing environment

For the sake of this thesis, 7 different attacks, one M_x from each malicious attack type found in Section 3.6, is implemented to test for time frame, f_i, applicability to different architecture models, success limitations, susceptibility to noise, and detectability. A listing of the specific attack objectives, ordered by the malicious attack type they belong to, can be seen below.

Implemented Attacks

$\boldsymbol{M}_1$ = encryption key theft

$\boldsymbol{M}_2$ = detection of co-located running program

$\boldsymbol{M}_3$ = capturing unique environment ids

$\boldsymbol{M}_4$ = malicious process triggered

$\boldsymbol{M}_5$ = interfere with coresident CPU usage

$\boldsymbol{M}_6$ = colluding VM detection

$\boldsymbol{M}_7$ = botnet communication

Where M_{1-4} require only a receiving program, M_{5-6} require only a sending program, and M_7 requires both a sender and a receiver. The metrics used to assess each implementation are applied uniformly across all process and are further detailed in the following subsection.

5.5 Metrics

5.5.1 Success

The success of applying a malicious application, M_x, across chosen channel, C_x, can be measured. First, through the ability for the malware to function using only the processes of the channel. This will be measured by flagging the malicious application as being either C_1, C_2 or C_3 compatible. This means that it relies on that specific sender or receiver process to operate.

Additionally we record whether more than a single bit is needed to a received or transmitted signal in order to make it useful to the malware's functionality. This will give insight into the overhead necessary to complete the attack. We measure this metric against an optimal bandwidth, b_i, and time frame, f_i.

Applicable Channels	C_1, C_2, C_3
Minimum Bits Required for Success of Attack	1-bit / 1 Process Alteration

5.5.2 Efficiency

The efficiency of the malicious side channel attack will be measured. Specifically, we measure the speed and capacity of the malware.On a larger scale, the possibility of repeating the same malicious attack is measured in the number of repetitions possible until degradation of the channel. This gives insight into the malware's persistence and potential scope. An efficient attack will not vary under continual use.

Resilience	# repetitions before degradation

5.5.3 Detectability

A major component of the malicious attack model is avoiding detection. We measure the level at which it can avoid observation or any unwanted identification. To test the covertness of the channel, the potential for defensive mechanisms applied by the server host assessed. Specifically, we measure the possibility for an intelligent scheduler or hypervisor to detect unwarranted hardware behavior.

Finally, we look at several detection techniques used in malware detection and apply it to malware across hardware side channels. These include, malicious file signature recognition, detection of the anomalies generated by malicious hardware activity, and finally monitoring resource elements.

Intelligent Hypervisor	% resource use visibility
Susceptibility to Detection Techniques	(listed below)

Specific Detection Techniques Used:

- Signature Based
- Anomaly Based
  - Specification Based
  - Pattern Recognition
- Protected Resource Ownership

Signature based detection is, by definition, a detection system based on known signatures of malicious activity. If a process signature is seen on the system matching one of the known signatures, the system can respond.

This applies best to systems with access to all static programs inside a virtual machine. As cloud hypervisors can only access active resource use and allocation, this thesis assumes the client's static data is private, the benefits of implementing signature based detection is minimal.

Anomaly based detection is, by definition, a detection system targeting computer intrusions and anomalous activity by monitoring system activity patterns and classifying it as either acceptable or deviating from what is standard. Under this umbrella falls specification based detection which relies on specifications that describe the intended behavior and resource usage of a virtual machine to identify anomalies at runtime. Pattern recognition, another anomaly based method, detects undesired patterns in hardware resource usage to identify side channel like behavior and take action accordingly. As these two techniques directly react to active changes in system activity, it has the most potential for defense against side channel attacks in the cloud.

Protected resource ownership refers to locking out untrusted users or third party virtual machine from using a hardware resource either at all, or while another, protected virtual machine is operating on that resource. This inherently decreases cloud computing efficiencies achieved through sharing hardware and is not a viable solution to cloud computing infrastructure. However restricting virtual machine resource consumption two completely isolate processes effectively mitigates side channel attacks.

5.6 Description of Implemented Attacks and Results

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 collocated. 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.

5.6.1 M_1 Theft of Encryption Key

The first set of attacks are termed Cryptographic Key Theft as discussed in Section 3. Applications of this set are classified as being an exfiltrating side channel attacks which rely exclusively on a receiving application, channel C_1. We implement M_1 an application belonging to this set. 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 [20, 12, 27, 3].

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 is 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 one hundred 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 seven thousand 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. 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 5.6.1.

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 85.9375%. This accuracy is significantly high enough to confidently map the periods of high and low operations in our chosen encryption algorithm. The summary of these findings may be found in figure 5.10 at the end of this section.

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.

Starting Bytes:	AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA
Encrypted Bytes:	AAAAAAAA BBBBBBBB AAAAAAAA BBBBBBBB
Bits Leaked:	

0000000000000001 1011011011111101 0000000000000010 1111111101011101

Figure 5.4: The encrypted string compared to the leaked string

5.6.2 M_2 Active Program Identification

Applications built to eavesdrop on concurrent processes fall under the second attack category outlined in Section 3. Attacks from this group uniquely identify co-active applications. For this subset, we chosen to implement malware M_2. Specifically, this processes eavesdrops on system behavior using the channel defined above. Recording specific, repeated out-of-order-execution patterns allows M_2 to map behavior to specific process identifiers.

For the purposes of this thesis, our implementation of M_2 was constructed on channel type C_1. This is implied for the activity monitory category outlined in Section 3. We ran M_2 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 0.93%. This is significantly high enough to reliably identify a client running video instances inside this browser process. An overview of the results of this attack, M_2 can be found in Figure 5.10.

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 statistical 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 this thesis's 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 5.5: The bit stream leaked through the receiver showing the repeating pattern associated with running videos in chrome

If figure 5.6.2, 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.

5.6.3 M_3 Environment Keying

Environment identifying programs are the third attack category outlined in Section 3. They rely exclusively on the receiving side channel application. For this reason, they may be labeled as processes which rely on channel type C_2. 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 [4].

For our specific implementation of an application from this malicious process category, we chose to implement a simple environmental keying malware. The attack contains two distinct phases. The first is a malicious virtual machine runs an environmental keying side channel program to generate a unique key used to identify the system. The second stage uses a piece of malware 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 [4, 8, 7].

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. For the sake of this thesis, 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 27.2925 seconds, or 3.41156 seconds per 4 bit segments and a standard deviation of 0.06478 seconds. An example of an unique environment key can be see in figure 5.6.2.

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 trails 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 96.875% accuracy. This matching was deemed sufficiently high enough to confidently 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.

5.6.4 M_4 Signal Trigger of Process

Our implementation of M_4 belongs to the last exfiltrating attack model specified in Section 3. 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.

This attack model may use channel types C_1 or C_2 depending on which of the two processes is continually running. 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 thesis.

In figure 5.6, 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.

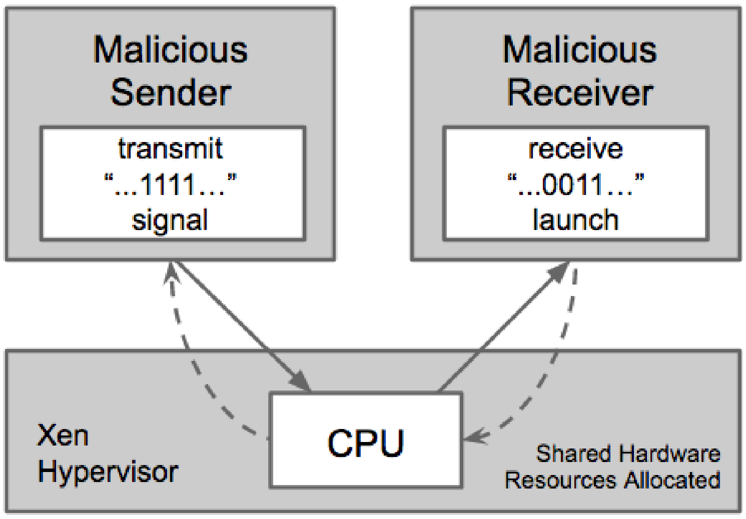

Malicious Sender
transmit "...1111..." signal
Malicious Receiver
receive "...0011..." launch
Xen Hypervisor
CPU
Shared Hardware Resources Allocated

Figure 5.6: Use of the central processor in synced, concurrent time frames allows the transmission of a signal between two colluding applications in real time

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 collocation 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.

5.6.5 M_5 CPU Resource Contention

The fifth set of malicious applications, from Section 3, is Resource Denial of Service. It is classified as an infiltrating side channel attack. This is defined by specific applications which rely exclusively on a transmitting application, channel construction two, C_2. We construct M_5 as a specific application belonging to this set. The purpose of this attack is to force contention of central processing unit resources for targeted processes.

In literature, the contention of any resource which negatively effects the targeted user is often referred to as a Denial-of-Service attack (DoS) [15, 28]. This elemental intrusion of the user's environment does not require the least level of precision compared to other attacks discussed in this thesis.

Compared to other adversarial models, this attack category, M_5, 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, M_5. For this thesis, we implement a specific, M_5, 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.

The predicted increase in the minimum duration needed to successfully execute this attack is see in our implementation, M_5, 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 M_5 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 M_5 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. Incorrect values meant that the thread order was incorrectly altered by the processor, 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

$$[35, 20, 0, \mathbf{9}, 10, \mathbf{8{,}5}, 35, 0, 45, \mathbf{8{,}7{,}1}, 10, 45, 20, \mathbf{8}, 25, \mathbf{7}, 15, 0, \mathbf{2{,}8}]$$

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 5.7: The array values before and after computation

Figure 5.7 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.

5.6.6 M_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.

From the malicious process category termed Determine VM Co-Location, which requires channel type C_2, we implement M_6. 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 5.8, 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.

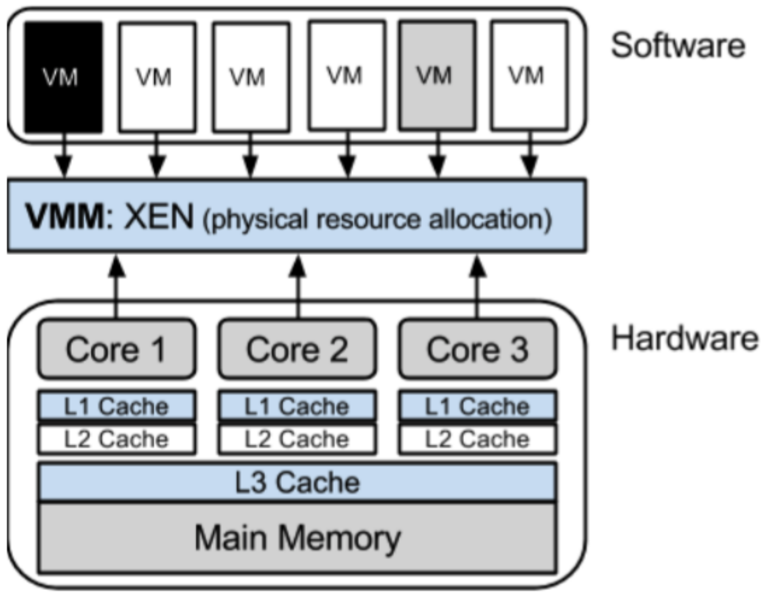


Figure 5.8: The black virtual machine represents the malicious host and the gray virtual machine represents the alternate continual transmitter

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, M_6, 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 implemented for this thesis. 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 by M_6. 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 architected C_2 as defined in Section 3. M_6 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 transmitor 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.

5.6.7 M_7 Bi-Way Communication

The final channel category of the three, C_3, requires the continual operation of both a transmitting process and a reception process. This generates a bi-way communication between two colluding applications located in separate, co-located virtual machines. The category termed Command-and-Control Communication Network contains all malicious applications which rely on this channel architecture. Our implementation of malicious application M_7 is a subset of this group, exploiting the environment using channel C_3.

Specifically, M_7 attempts to successfully create a covert channel for two malicious virtual machines. This allows the virtual machines to communicate without generating highly visible network traffic caused through normal mechanisms.

For our experimentation, we used the same environment setup as with the previous attack implementations. This includes the Xen server and six virtual 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, co-resident virtual machines.

In the implementation of M_7, 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 0.95107 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 3.80428 seconds for a single four bit message to be sent. This attack is the longest of the seven described for this thesis. The standard deviation on one hundred tests was 0.13538 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 M_7 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 5.9.

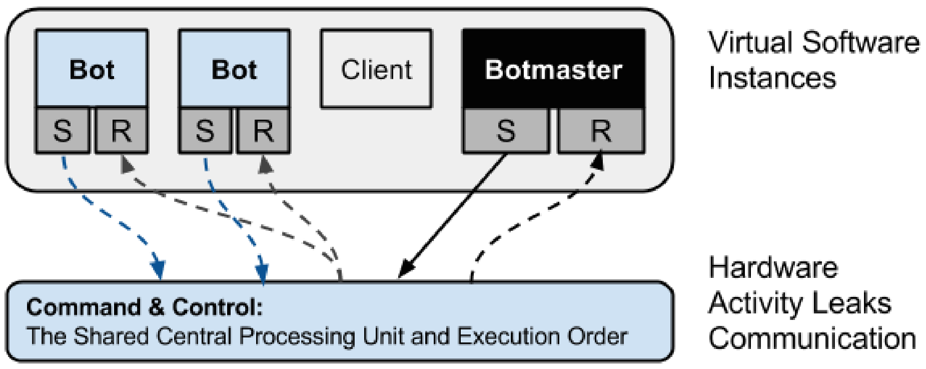
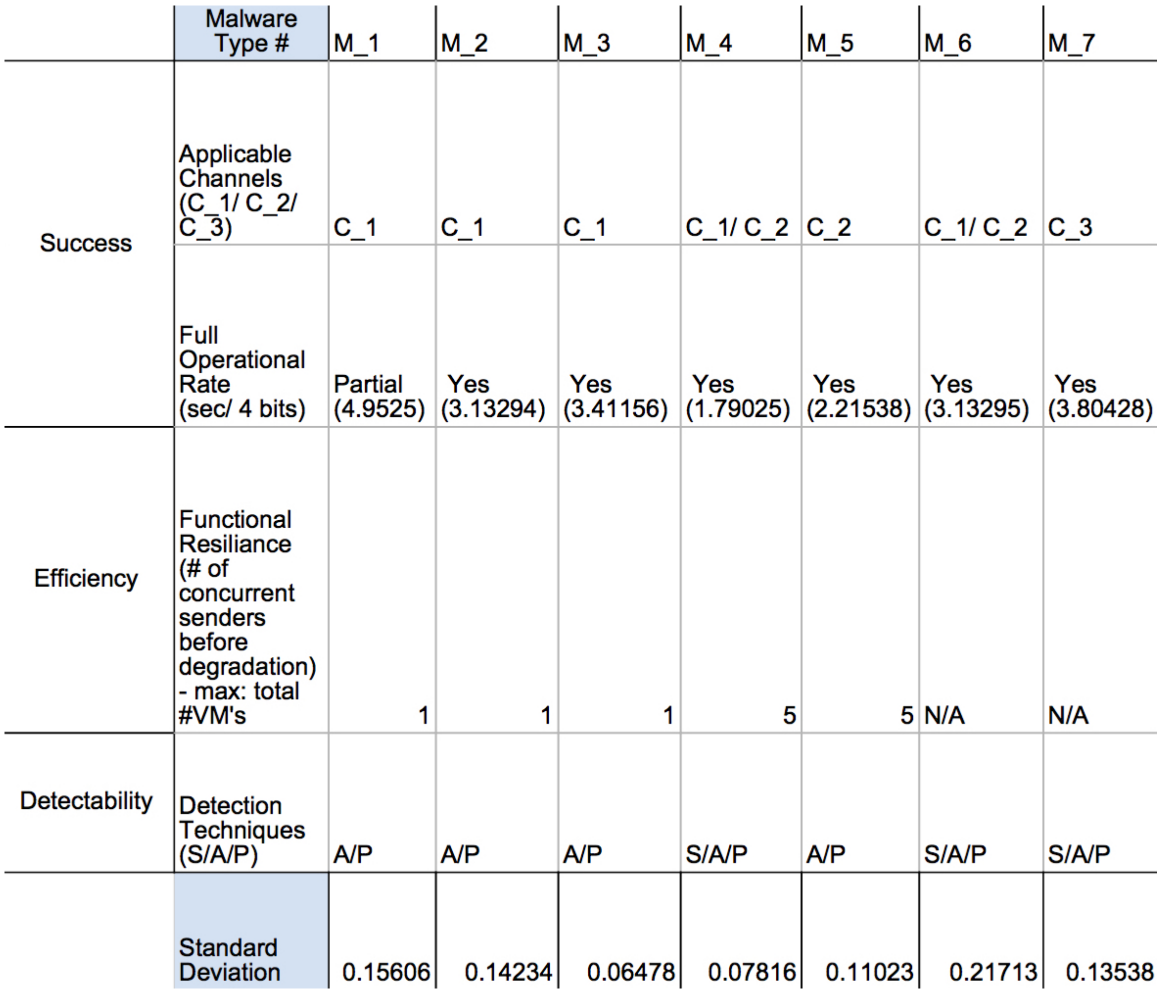

Bot
S
R
Bot
S
R
Client
Botmaster
S
R
Virtual Software Instances
Command & Control:
The Shared Central Processing Unit and Execution Order
Hardware Activity Leaks Communication

Figure 5.9: A botnet which uses n bots that receive commands and transmit responses to the central authority; our side channel acts as the relay

5.7 Summary of Implementation Measurements



	Malware Type #	M_1	M_2	M_3	M_4	M_5	M_6	M_7
Success	Applicable Channels (C_1/ C_2/ C_3)	C_1	C_1	C_1	C_1/ C_2	C_2	C_1/ C_2	C_3
	Full Operational Rate (sec/ 4 bits)	Partial (4.9525)	Yes (3.13294)	Yes (3.41156)	Yes (1.79025)	Yes (2.21538)	Yes (3.13295)	Yes (3.80428)
Efficiency	Functional Resiliance (# of concurrent senders before degradation) - max: total #VM's	1	1	1	5	5	N/A	N/A
Detectability	Detection Techniques (S/A/P)	A/P	A/P	A/P	S/A/P	A/P	S/A/P	S/A/P
	Standard Deviation	0.15606	0.14234	0.06478	0.07816	0.11023	0.21713	0.13538

Figure 5.10: A table listing the average deploy rate for each of the seven attacks implemented as well as the associated standard deviations

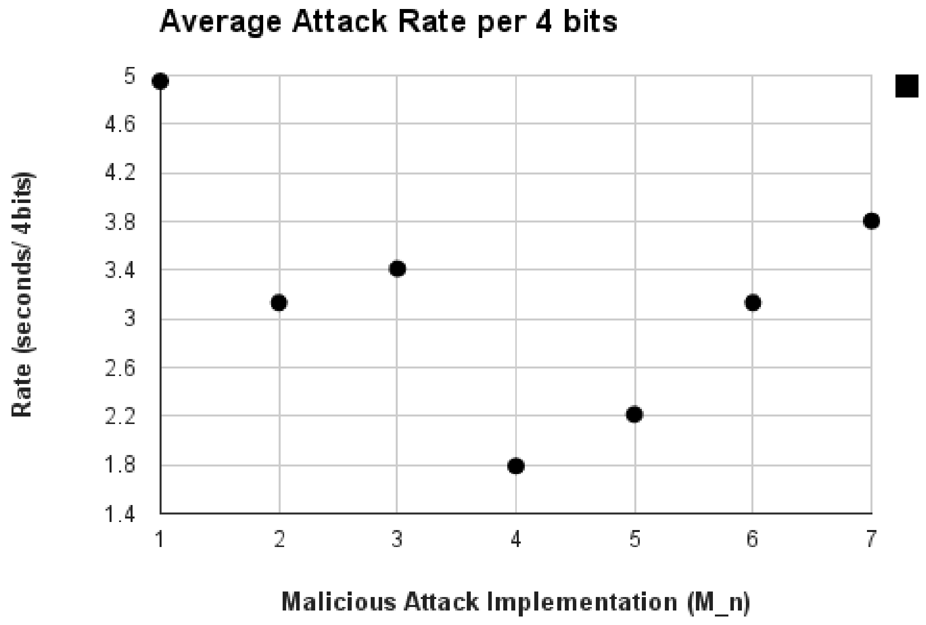


Figure 5.11: A chart visualizing the execution rate of each implementation and the minimum times necessary for each attack's success

6. APPLICATION AND POTENTIAL OF INTRUSION DETECTION TECHNIQUES

This thesis outlines seven attack categories deployed across variations of our out-of-order execution side channel. An application from each category is then implemented. Presented with these adversarial models, we attempt to highlight the major detection techniques for an intelligent hypervisory. Consider that each malicious application requires signal transfer, either exfiltrating or infiltrating, across the central processor. This use inherently mimics malware's use of a traditional network.

We target a set of defensive methods applicable to our channel construction. These are outlined in Figure 6.1.

In order to detect this malicious activity, a process residing on a host machine may be monitored in three ways [29]. Through dynamic analysis of their interaction with external processes and protected resources, termed Anomaly-Based. Through static identification of known malicious programs, termed Signature-Based. Through matching recorded activity patterns with known untrusted or restricted behaviors, termed Pattern-Based. A subset of these techniques may be applied to defend against applications which exploit hardware side channels.

6.1 Anomaly-Based Channel Malware Detection

Anomaly based detection is, by definition, a system for detecting computer intrusions and misuse [30]. This is done by monitoring system level activity and classifying it as either normal or anomalous.We apply this technique to hardware side channels.

A pattern of activity across the targeted hardware medium is instigated by the channel's transmission or reception processes. This pattern may be recorded or observed by the hypervisor. Using anomaly based detection methods, the hypervisor may recognize communication like activity, the repeated resource consumption, coming from a virtual machine instance.

Specification based system monitoring uses a dictionary of expected behaviors and resource consumption habits for each virtual machine [31, 32]. This gives the hypervisor insight into what should be considered anomalous behaviors. For example, a virtual instance registered as hosting a static website requires sporadic use of the processor. An intelligent hypervisor could associated scarce processor use with this machine. If the virtual machine begins requiring long periods of processor time for computation, suspicion would be raised.

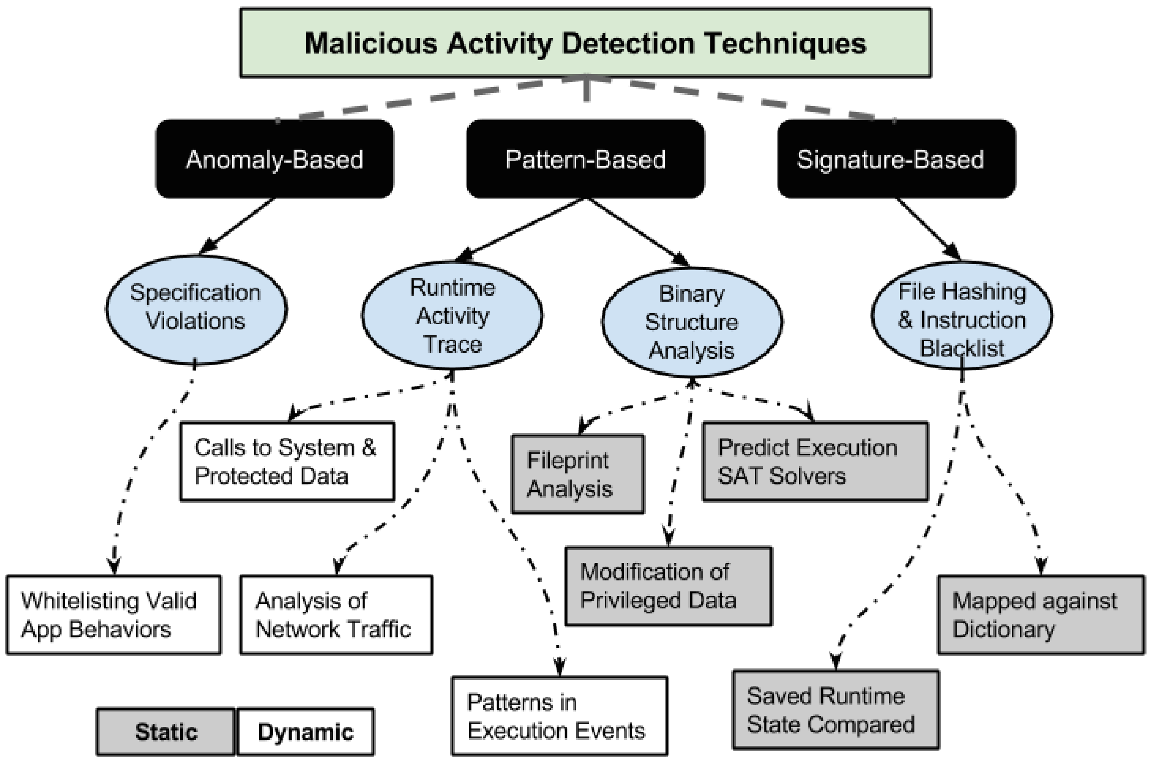

Malicious Activity Detection Techniques
Anomaly-Based
Pattern-Based
Signature-Based
Specification Violations
Runtime Activity Trace
Binary Structure Analysis
File Hashing & Instruction Blacklist
Calls to System & Protected Data
Fileprint Analysis
Predict Execution SAT Solvers
Modification of Privileged Data
Whitelisting Valid App Behaviors
Analysis of Network Traffic
Mapped against Dictionary
Saved Runtime State Compared
Patterns in Execution Events
Static
Dynamic

Figure 6.1: The set of detection techniques applicable to the seven attack categories deployed across the out-of-order-execution side channel

6.2 Signature-Based Channel Malware Detection

Signature based defensive methods is most often implemented by antivirus software [33]. It uses a dictionary of static signatures or hashes generated per file or segment of code. If an program or a subset of it can be hashed and found in the dictionary, the host determines the program to be malicious. The technique of blacklisting hashes can be applied to side channel attacks.

In this case, a particular block of shared memory resource used by a procress could be hashed. This helps determine if a known side channel process is residing in memory or if a process is filling segments of memory with blacklisted contents [34].

Alternatively, the memory used by one virtual instance may be hashed and later rehashed after other processes executed or time elapsed. This would provide insight into repeatedly emptied or repeatedly queried blocks of memory.

6.3 Pattern-Based Channel Malware Detection

Pattern-based detection methods monitor system behavior. The system may then recognize communication-like activity originating from one or more virtual machine instances. Two distinct method types stem from pattern based detection. Tracing runtime activity and static analysis of binary data [35].

Of the two, the latter cannot easily be applied to malicious side channel detecting as it would require full access to a user's programs residing in the virtual machine. We assume the hypervisor does not have read privileges for client data inside a virtual machine. Static Pattern-Based methods are out of scope for side channel detection.

Alternatively, analysis of runtime activity traces applies to the side channel attack model. Consider an intelligent hypervisor which records system activity of the hardware resource leveraged by the side channel to communicate. The cyclical transmission of bits by forcing a specific resource state inherently generates detectable patterns.

The hypervisor dynamically learns which patterns are safe and which indicate anomalous activity [30, 35]. If prior runtime traces are acquired from an example system, the hypervisor can compare recorded and expected patterns. Then positive identify malicious side channel communication can be made.

6.4 Inherent Strengths of Hardware Side Channels Against Detection

Hardware layer side channels possess numerous strengths over other types. These weaker implementations include side channels built using network artifacts, such as social media [36]. Shared hardware side channels, by nature of their construction, circumvent many defensive mechanisms usually applicable to these network or software level exploits.

In cloud computing infrastructures, the virtual machine's contents are opaque to the third party host. Therefore all static malware detection techniques are of little value. All that is available to the hypervisor is the interaction of the processes at runtime with the physical system. We present an intelligent hypervisor or host system to exploit these interactions.

The intelligent hypervisor records a user's expected resource consumption, runtime behaviors, and other identifiers. These properties allow restrictions to be set on resources and co-executing process interaction. These modifications also support the defensive methods discussed in the previous section. However, numerous changes to the current state of the hypervisor and its anonymous allocation of virtual machine resources are required.

Another common security measure, easily defeated by hardware side channel attacks, is the use of sandboxed environments. Sandboxes isolate executing programs from the live system and record behaviors [37]. They place a layer of virtualization between the side channel and the hardware medium used in the communication. This defensive mechanism, by its presence, disrupts the fundamental medium of the side channel.

In this sandbox, a reception process, querying the hardware component, is made aware that the noise levels or response timings have changed. The readings from the system will either not be as expected or fail generate meaningful signals. Once the target environment is abstracted away from the hypervisor, the signal being sent and received is adversely perceived by the side channel processes. The side channel application may then idle until the hardware readings indicate the removal of the sandbox. In this defensive techniques, observation affects the channel as well as the expected hardware responses and access capabilities.

Overall, the use of static software analysis or sandboxing by the host does not reliably defend against hardware based side channels. Furthermore, they are easily mitigated by a side channel's reception of unexpected hardware behaviors.

6.5 Potential of Detection Techniques against Side Channel Attacks

In our experiments, we assume that the data stored inside a virtual machine is opaque to the hypervisor. Static analysis of a user's binaries is not permitted as per cloud based privacy rules [26]. 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 and pattern matching will further enhance the effectiveness of these techniques in side channel detection.

Additionally, some typical malware detection techniques may also be applied for the detection of side channels [38, 37]. 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 [39, 40, 41]. 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. Effective techniques include dynamic and static methods. Dynamic methods monitor resource consumption patterns of individual processes. Static methods hash and analyze used memory spaces of the processes over time [42, 33].

Effective Detection Techniques:

- Signature Based
  - Analysis of memory block signatures over time
- Anomaly Based
  - Specification Based
    - Limiting system calls
  - Pattern Recognition
    - Resource use patterns
    - Repeated data manipulation
- Protected Resource Ownership
  - Isolate Virtual Machine Hardware
  - Blacklist resource access by concurrent processes

All three categories outlined in Figure 6.1 show strong potential for detection of side channel exploitation in the cloud. Techniques from each of these categories mitigate the potential vulnerabilities from sharing hardware. They do not interfere with the efficiencies gained through sharing resources by isolating resources or prohibiting co-location. Further research into these three techniques will better tailor the defensive methods to large scale cloud environments.

7. CONCLUSION

In this thesis, we establish a basis for classifying seven potential adversarial models through identifying side channel primitives. To do so, we assess the properties inherent to all side channel exploits across the spectrum of shared hardware components. Necessary factors include static time frames, co-location, and pre-arranged channel protocols. Assuming these factors, we classify our potential adversarial models by their functionality. To further divide these models, we create three distinct channel types - exfiltrating, infiltrating, or networking under which different attack models belong.

Using this classification, we create seven distinct malware models which encompass all possible malware implementations. Each model possesses unique requirements for sending and receiving processes. Additionally, each model instruments singular algorithms for achieving the desired attack with available data. We classify the models under one of the three channel architectures. This allows for architecture specific features and impediments to be taken into account during implementation.

In order to provide a proof of concept for our typologies, we create seven novel malicious applications, each of which are associated with an attack model. Each of the seven malware implementations exploit the primitives of our out-of-order-execution side channel. The environment used mimics that of a standard cloud running a Xen hypervisor and several benign virtual machines.

Assessing the above findings, we conclude that exploitation of out-of-order-execution across the CPU shows potential for attacking large scale cloud environment.

Additionally, shared hardware behavior can be recorded or altered, and subsequently mapped to specific functions. This lack of anonymity has implications for the future of user security in the cloud. Also, creating covert communication between multiple virtual machines has direct potential for a live environment. Further research in areas of all seven malware attack models would target optimization, noise reduction, and speed.

We outlined the three main categories for dynamic OoOE side channel detection which show promise for hypervisor-level security - Resource, Signature, and Anomaly. Dynamic security and resource consumption monitoring must be pursued in order to maintain the current level of both process anonymity and private data storage in Infrastructure-as-a-Service.

Ideally, this work will provide an effective means for evaluating preexisting and novel side channel attack vectors. Also, we present attacks which show the vulnerabilities present in modern cloud environments. Emphasizing these areas will focus future research into developing offensive side channel applications and innovating solutions to future cloud based security vulnerabilities.

LITERATURE CITED

[1] M. Godfrey and M. Zulkernine, "Preventing cache-based side-channel attacks in a cloud environment," *IEEE Trans. Cloud Comput*, vol. 2, pp. 395–408, Oct. 2014.

[2] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *IEEE/IFIP 41st Int. Conf. on Dependable Syst. and Networks Workshops*, pp. 194–199, June 2011.

[3] C. Percival, "Cache missing for fun and profit," in *Proc. of BSDCan*, 2005.

[4] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring inform. leakage in third-party compute clouds," in *Proc. of the 16th ACM Conf. on Comput. and Commun. Security*, CCS '09, (New York, NY, USA), pp. 199–212, ACM, 2009.

[5] K. Gurudutt, "Considerations in software design for multi-core multiprocessor architectures," May 2013. www.ibm.com/developerworks/aix/library/au-aix-multicore-multiprocessor, [Accessed Dec. 17, 2014].

[6] P. Brady, "Memory hierarchy," 2008. www.pixelbeat.org/docs/memory_hierarchy, [Accessed Jan. 2, 2015].

[7] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "Homealone: Co-residency detection in the cloud via side-channel anal.," in *IEEE Security Privacy*, SP '11, (Washington, DC, USA), pp. 313–328, IEEE Comput. Soc., 2011.

[8] S Yu, XL Gui, JC Lin, JF Wang, and XJ Zhang, "Detecting vms co-residency in the cloud: using cache-based side channel attacks," *Electron. and Elect. Eng.*, vol. 19, no. 5, p. 7378, 2013.

[9] R. Di Pietro, F. Lombardi, and A. Villani, "Cuda leaks: information leakage in gpu architectures," *arXiv preprint arXiv:1305.7383*, 2013.

[10] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Proc. of the 22nd Annu. Comput. Security Appl. Conf.*, ACSAC '06, (Washington, DC, USA), pp. 473–482, IEEE Comput. Soc., 2006.

[11] J. Chen and G. Venkataramani, "Cc-hunter: Uncovering covert timing channels on shared processor hardware," in *2014 47th Annu. IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*, pp. 216–228, Dec. 2014.

[12] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *ACM Conf. on Comput. and Commun. Security*, CCS '12, (New York, NY, USA), pp. 305–316, ACM, 2012.

[13] Y. Yarom and K. E. Falkner, "Flush+ reload: a high resolution, low noise, l3 cache side-channel attack.," *IACR Cryptology ePrint Archive*, vol. 2013, p. 448, 2013.

[14] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud," in *Proc. of the 21st USENIX Conf. on Security Symp.*, Security'12, (Berkeley, CA, USA), pp. 9–9, USENIX Assoc., 2012.

[15] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core syst," in *Proc. of 16th USENIX Security Symp. on USENIX Security Symp.*, SS'07, (Berkeley, CA, USA), pp. 18:1–18:18, USENIX Assoc., 2007.

[16] B. Lipinski, W. Mazurczyk, and K. Szczypiorski, "Improving hard disk contention-based covert channel in cloud computing environment," *CoRR*, vol. abs/1402.0239, 2014.

[17] Z. Tari, "Security and privacy in cloud computing," *IEEE Trans. Cloud Comput*, vol. 1, pp. 54–57, May 2014.

[18] D. Fernandes, L. Soares, J. Gomes, M. Freire, and P. Incio, "Security issues in cloud environments: a survey," *Int. J. of Inform. Security*, vol. 13, no. 2, pp. 113–170, 2014.

[19] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *Proc. of the 2014 ACM SIGSAC Conf. on Comput. and Commun. Security*, CCS '14, (New York, NY, USA), pp. 990–1003, ACM, 2014.

[20] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *The Cryptographers Track at the RSA Conf. on Topics in Cryptology*, pp. 1–20, Springer-Verlag, 2005.

[21] E. Tromer, D. Osvik, and A. Shamir, "Efficient cache attacks on AES, and countermeasures," *J. of Cryptology*, vol. 23, no. 1, pp. 37–71, 2010.

[22] A. Tannous, J. Trostle, M. N. Hassan, S. E. McLaughlin, and T. Jaeger, "New side channels targeted at passwords," in *Proc. of the 2008 Annu. Comput. Security Appl. Conf.*, ACSAC '08, (Washington, DC, USA), pp. 45–54, IEEE Comput. Soc., 2008.

[23] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," *SIGARCH Comput. Archit. News*, vol. 13, no. 3, pp. 36–44, 1985.

[24] W. Hwu and Y. N. Patt, "Hpsm, a high performance restricted data flow architecture having minimal functionality," *SIGARCH Comput. Archit. News*, vol. 14, pp. 297–306, May 1986.

[25] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Res. Dev.*, vol. 11, no. 1, pp. 25–33, 1967.

[26] Amazon, "Amazon ec2 instances." Online, Dec. 2014.

[27] Y. Tsunoo, T. Saito, T. Suzaki, and M. Shigeri, "Cryptanalysis of des implemented on computers with cache," in *Springer LNCS Proc. of CHES*, pp. 62–76, Springer-Verlag, 2003.

[28] D. Grunwald and S. Ghiasi, "Microarchitectural denial of service: insuring microarchitectural fairness," in *Proc. 35th Annu. IEEE/ACM Int. Symp. on Microarchitecture*, pp. 409–418, 2002.

[29] J. Wu, L. Ding, Y. Wu, N. Min-Allah, S. U. Khan, and Y. Wang, "C2detector: a covert channel detection framework in cloud computing," *Security and Communication Networks*, vol. 7, no. 3, pp. 544–557, 2014.

[30] M. Gander, M. Felderer, B. Katt, A. Tolbaru, R. Breu, and A. Moschitti, "Anomaly detection in the cloud: Detecting security incidents via mach. learning," in *Trustworthy Eternal Syst. via Evolving Software, Data and Knowledge* (A. Moschitti and B. Plank, eds.), vol. 379 of *Commun. in Comput. and Inform. Sci.*, pp. 103–116, Springer Berlin Heidelberg, 2013.

[31] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automat. detection and removal of control-flow side channel attacks," ICISC'05, (Berlin, Heidelberg), pp. 156–168, Springer-Verlag, 2006.

[32] C.-Y. Tseng, P. Balasubramanyam, C. Ko, R. Limprasittiporn, J. Rowe, and K. Levitt, “A specification-based intrusion detection syst. for aodv,” in *Proc. of the 1st ACM Workshop on Security of Ad Hoc and Sensor Networks*, SASN ’03, (New York, NY, USA), pp. 125–134, ACM, 2003.

[33] S. Naval, V. Laxmi, M. S. Gaur, and P. Vinod, “Spade: Signature based packer detection,” in *Proc. of the 1st Int. Conf. on Security of Internet of Things*, SecurIT ’12, (New York, NY, USA), pp. 96–101, ACM, 2012.

[34] L. Hélouët and A. Roumy, “Covert channel detection using information theory,” *arXiv preprint arXiv:1102.5586*, 2011.

[35] R. Lanotte, A. Maggiolo-Schettini, and A. Troina, “Time and probability-based inform. flow anal.,” *IEEE Trans. Softw. Eng.*, vol. 36, no. 5, pp. 719–734, 2010.

[36] N. Lawson, “Side-channel attacks on cryptographic software,” *IEEE Security Privacy*, vol. 7, pp. 65–68, Nov. 2009.

[37] M. Saher and J. Pathak, “Malware and exploit campaign detection sys. and method.” U.S. Patent 20150074810, March, 12, 2015.

[38] I. Kyte, P. Zavarsky, D. Lindskog, and R. Ruhl, “Enhanced side-channel analysis method to detect hardware virtualization based rootkits,” in *Internet Security (WorldCIS), 2012 World Congr. on*, pp. 192–201, June 2012.

[39] R. Martin, J. Demme, and S. Sethumadhavan, “Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks,” in *Proc. of the 39th Annu. Int. Symp. on Comput. Architecture*, ISCA ’12, (Washington, DC, USA), pp. 118–129, IEEE Comput. Soc., 2012.

[40] T. Kim, M. Peinado, and G. Mainar-Ruiz, “Stealthmem: Syst.-level protection against cache-based side channel attacks in the cloud,” in *Proc. of the 21st*

USENIX Conf. on Security Symp., Security'12, (Berkeley, CA, USA), pp. 11–11, USENIX Assoc., 2012.

[41] V. Varadarajan, T. Ristenpart, and M. Swift, "Scheduler-based defenses against cross-vm side-channels," in *23rd USENIX Security Symp. (USENIX Security 14)*, (San Diego, CA), pp. 687–702, USENIX Assoc., 2014.

[42] M. Milenković, A. Milenković, and E. Jovanov, "Using instruction block signatures to counter code injection attacks," *ACM SIGARCH Comput. Architecture News*, vol. 33, no. 1, pp. 108–117, 2005.