# Whispers in the Hyper-Space: High-Bandwidth and Reliable Covert Channel Attacks Inside the Cloud

Zhenyu Wu, *Member, IEEE*, Zhang Xu, and Haining Wang, *Senior Member, IEEE*

***Abstract*—Privacy and information security in general are major concerns that impede enterprise adaptation of shared or public cloud computing. Specifically, the concern of virtual machine (VM) physical co-residency stems from the threat that hostile tenants can leverage various forms of side channels (such as cache covert channels) to exfiltrate sensitive information of victims on the same physical system. However, on virtualized x86 systems, covert channel attacks have not yet proven to be practical, and thus the threat is widely considered a "potential risk." In this paper, we present a novel covert channel attack that is capable of high-bandwidth and reliable data transmission in the cloud. We first study the application of existing cache channel techniques in a virtualized environment and uncover their major insufficiency and difficulties. We then overcome these obstacles by: 1) redesigning a pure timing-based data transmission scheme, and 2) exploiting the memory bus as a high-bandwidth covert channel medium. We further design and implement a robust communication protocol and demonstrate realistic covert channel attacks on various virtualized x86 systems. Our experimental results show that covert channels do pose serious threats to information security in the cloud. Finally, we discuss our insights on covert channel mitigation in virtualized environments.**

***Index Terms*—Cloud, covert channel, network security.**

## I. Introduction

CLOUD vendors today are known to utilize virtualization heavily for consolidating workload and reducing management and operation cost. However, due to the relinquished control from data owners, data in the cloud are more susceptible to leakage by operator errors or theft attacks. Cloud vendors and users have used a number of defense mechanisms to prevent data leakage, ranging from network isolation to data encryption. Despite the efforts being paid on information safeguarding, there remain potential risks of data leakage, namely the covert channels in the cloud [1]–[5].

Covert channels exploit imperfections in the isolation of shared resources between two unrelated entities and enable communications between them via unintended channels, bypassing mandatory auditing and access controls placed on standard communication channels. Unlike a seemingly similar threat, side channel attacks [1], [6] that extrapolate information by observing an unknowing sender, covert channels transfer data between two collaborating parities. However, the additional requirement of an insider does not significantly reduce the usefulness of covert channels in data theft attacks. Data theft attacks using covert channels can be launched in two steps, namely *infiltration* and *exfiltration*. In the *infiltration* step, attackers may apply multiple infiltration attacks (such as buffer overflow [7], VM image pollution [8], [9], and various social engineering techniques [10], [11]) to place "insiders" in a victim and harvest secret information. In the *exfiltration* step, secret information is transported to the attacker via a covert channel, without leaving a trace on any security surveillance systems (e.g., firewalls, intrusion detection systems, network traffic logs, etc.)

While previous research has shown that covert channels on a nonvirtualized system can be constructed using a variety of shared media [12]–[16], to date there is no known practical (i.e., high-speed) exploit of covert channels on virtualized x86 systems. Exposing cloud computing to the threat of covert channel attacks, Ristenpart *et al.* [2] have implemented an L2 cache channel in Amazon EC2 [2], achieving a bandwidth of 0.2 bits per second (b/s), far less than the 1-b/s "acceptable" threshold suggested by the Trusted Computer System Evaluation Criteria (TCSEC, a.k.a. the "Orange Book") [17]. A subsequent measurement study of cache covert channels [4] has achieved slightly improved speeds—a theoretical channel capacity of 1.77 b/s.[1] Given such low reported channel capacities from previous research, it is widely believed that covert channel attacks could only do very limited harm in the cloud environment. Coupled with the fact that the cloud vendors impose nontrivial extra service charges for providing physical isolation, one might be tempted to disregard the concerns of covert channels as purely precautionary and choose the lower cost solutions instead.

In this paper, we show that the threat of covert channel attacks in the cloud is real and practical. We first study existing cache covert channel techniques and their applications in a virtualized environment. We reveal that these techniques are rendered ineffective by virtualization due to three major insufficiency and difficulties, namely, *addressing uncertainty*, *scheduling uncertainty*, and *cache physical limitations*. We tackle the addressing and scheduling uncertainty problems by designing a pure timing-based data transmission scheme with relaxed dependencies on precise cache line addressing and scheduling patterns. Then, we overcome the cache physical

Manuscript received November 25, 2012; revised September 06, 2013 and November 12, 2013; accepted January 16, 2014; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor Y.-C. Hu. Date of publication February 19, 2014; date of current version April 14, 2015.

Z. Wu is with NEC Laboratories America, Inc., Princeton, NJ 08540 USA (e-mail: adamwu@nec-labs.com).

Z. Xu and H. Wang are with the Department of Computer Science, College of William and Mary, Williamsburg, VA 23187 USA.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2014.2304439

[1]This value is derived from the results presented in the original paper—a bandwidth of 3.20 b/s with an error rate of 9.28%, by assuming a binary symmetric channel.

limitations by discovering a high-bandwidth memory bus covert channel, exploiting the atomic instructions and their induced cache-memory bus interactions on x86 platforms. Unlike cache channels, which are limited to a physical processor or a silicon package, the memory bus channel works system-wide, across physical processors, making it a very powerful channel for cross-VM covert data transmission.

We further demonstrate the real-world exploitability of the memory bus covert channel by designing a robust data transmission protocol and launching realistic attacks on our testbed server as well as in the Amazon EC2 cloud. We observe that the memory bus covert channel can achieve: 1) a bandwidth of over 700 b/s with extremely low error rate in a laboratory setup, and 2) a real-world transmission rate of over 100 b/s in the Amazon EC2 cloud. Our experimental results show that, contrary to previous research and common beliefs, covert channels are able to achieve high bandwidth and reliable transmission on today's x86 virtualization platforms.

The remainder of this paper is structured as follows. Section II surveys related work on covert channels. Section III describes our analysis of the reasons that existing cache covert channels are impractical in the cloud. Section IV details our exploration of building high-speed, reliable covert channels in a virtualized environment. Section V presents our evaluation of launching covert channel attacks using realistic setups. Section VI provides a renewed view of the threats of covert channels in the cloud and discusses plausible mitigation avenues. Section VII concludes this paper.

## II. Related Work

Covert channel is a well-known type of security attack in multiuser computer systems. Originated in 1972 by Lampson [12], the threats of covert channels are prevalently present in systems with shared resources, such as file system objects [12], virtual memory [15], network stacks and channels [13], [14], [18], processor caches [1], [16], input devices [19], etc. [17], [20].

Compared to other covert channel media, the processor cache is more attractive for exploitation because its high operation speed could yield high channel bandwidth and the low-level placement in the system hierarchy can bypass many high-level isolation mechanisms. Thus, cache-based covert channels have attracted serious attention in recent studies.

Percival [16] introduced a technique to construct interprocess high-bandwidth covert channels using the L1 and L2 caches and demonstrated a cryptographic key leakage attack through the L1 cache side channel. Wang and Lee [1] deepened the study of processor cache covert channels and pointed out that the insufficiency of software isolation in virtualization could lead to cache-based cross-VM covert channel attacks. Ristenpart *et al.* [2] further exposed cloud computing to covert channel attacks by demonstrating the feasibility of launching VM co-residency attacks and creating an L2 cache covert channel in the Amazon EC2 cloud. Zhang *et al.* [21] presented a cross-VM side channel attack using a combination of support vector machine (SVM) and hidden Markov model (HMM), extracting cryptographic keys by inferring cypher operations from cache timing observations. Xu *et al.* [4] conducted a follow-up measurement study of [2] on L2 cache covert channels in a virtualized environment. Based on their measurement results, they concluded that the harm of data exfiltration from cache covert channels is quite limited due to low achievable channel capacity.

In response to the discovery of cache covert channel attacks, a series of architectural solutions has been proposed to limit cache channels, including RPcache [1], PLcache [22], and Newcache [23]. RPcache and Newcache employ randomization to prevent data transmission by establishing a location-based coding scheme. PLcache, however, is based on enforcing resource isolation by cache partitioning.

One drawback of hardware-based solutions is their high adaptation cost and latency. With the goal of offering immediately deployable protection, HomeAlone [3] proactively detects the co-residence of unfriendly VMs. Leveraging the knowledge of existing cache covert channel techniques [2], [16], HomeAlone infers the presence of a malicious VM by acting like a covert channel receiver and observing cache timing anomalies caused by another receiver's activities. STEALTHMEM [24] presents a hypervisor-based side-channel defense framework, which reserves for each guest VM a small amount of memory that guarantees to always stay in the processor cache. Enlightened guest operating systems and applications can store sensitive information, such as cryptographic keys or cipher functions, in the cache-resident storage, and thus become immune to cache channel attacks.

The industry has taken a more pragmatic approach to mitigating covert channel threats. The Amazon EC2 cloud provides a featured service called dedicated instances [25], which ensures VMs belonging to each tenant of this service do not share physical hardware with any other cloud tenants' VMs. This service effectively eliminates various covert channels induced by the shared platform hardware, including cache covert channel. However, in order to enjoy this service, the cloud users have to pay a significant price premium.[2]

Of historical interest, the study of covert channels in virtualized systems is far from a brand new research topic—legacy research that pioneered this field dates back over 30 years. During the development of the VAX security kernel, a significant amount of effort has been paid to limit covert channels within the Virtual Machine Monitor (VMM). Hu [26], [27] and Gray [28], [29] have published a series of follow-up research on mitigating cache channels and bus contention channels, using timing noise injection and lattice scheduling techniques. However, this research field has lost its momentum until recently, probably due to the cancellation of the VAX security kernel project, as well as the lack of ubiquity of virtualized systems in the past.

## III. Struggles of the Classic Cache Channels

Existing cache covert channels (namely, the classic cache channels) employ variants of Percival's technique, which uses a hybrid timing and storage scheme to transmit information over a shared processor cache, as described in Algorithm 1.

The classic cache channels work very well on hyper-threaded systems, achieving transmission rates as high as hundreds of

[2]As of August 2012, each dedicated instance incurs a 23.5% higher per-hour cost than regular usage. In addition, there is a $10 fee per hour/user/region. Thus, for a user of 20 small instances, the overall cost of using dedicated instances is 6.12 times more than that of using regular instances.

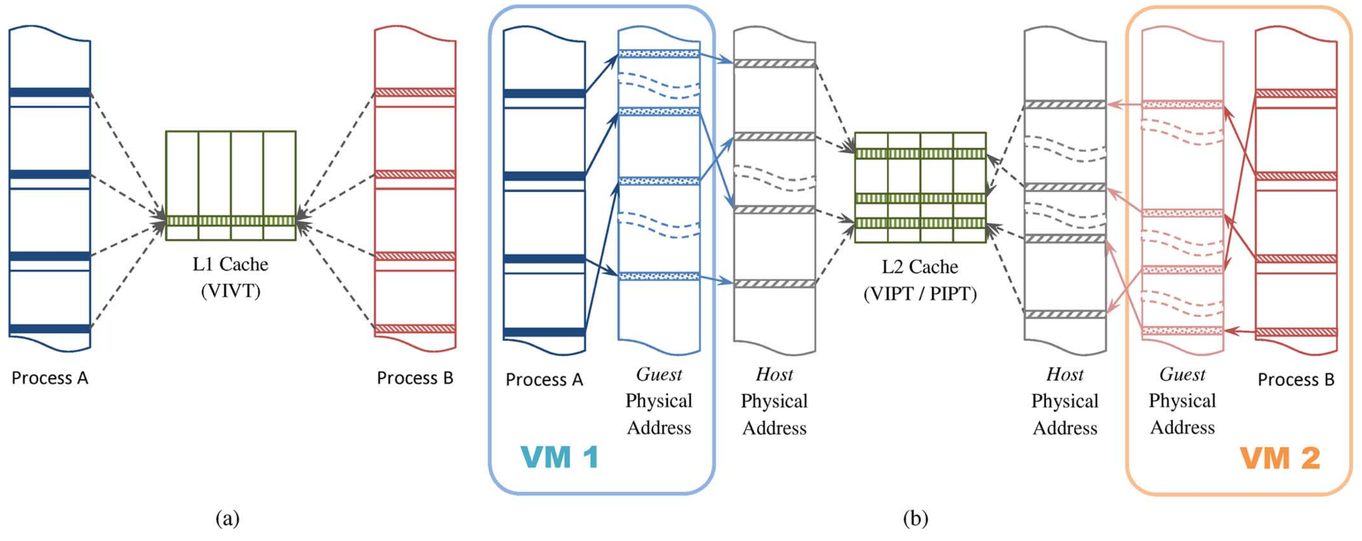


Fig. 1. Memory address to cache line mappings for (a) a VIVT L1 cache and (b) an L2 cache on virtualized systems.

**Algorithm 1** Classic Cache Channel Protocol

```
$Cache[N]$: A shared processor cache, conceptually divided into $N$ regions;
            Each cache region can be put in one of two states, cached or flushed.
$D_{Send}[N]$, $D_{Recv}[N]$: $N$ bit data to transmit and receive, respectively.

Sender Operations:
(Wait for receiver to initialize the cache)

Receiver Operations:
for $i := 0$ to $N-1$ do
    {Put $Cache[i]$ into the cached state}
    Access memory maps to $Cache[i]$;
end for
- - - - - - - - - - - - - - - - - - - -
Sender Operations:
for $i := 0$ to $N-1$ do
    if $D_{Send}[i] = 1$ then
        {Put $Cache[i]$ into the flushed state}
        Access memory maps to $Cache[i]$;
    end if
end for

Receiver Operations:
(Wait for sender to prepare the cache)
- - - - - - - - - - - - - - - - - - - -
Sender Operations:
(Wait for receiver to read the cache)

Receiver Operations:
for $i := 0$ to $N-1$ do
    Timed access memory maps to $Cache[i]$;
    {Detect the state of $Cache[i]$ by latency}
    if $AccessTime > Threshold$ then
        $D_{Recv}[i] := 1$; {$Cache[i]$ is flushed}
    else
        $D_{Recv}[i] := 0$; {$Cache[i]$ is cached}
    end if
end for
```

kilobytes per second [16]. However, when applied in today's virtualized environments, the achievable rates drop drastically, to only low single-digit bits per second [2], [4]. The multiple-orders-of-magnitude reduction in channel capacity clearly indicates that the classic cache channel techniques are no longer suitable for cross-VM data transmission. In particular, we found that on virtualized platforms, the data transmission scheme of a classic cache channel suffers three major obstacles—addressing uncertainty, scheduling uncertainty, and cache physical limitation.

## *A. Addressing Uncertainty*

Classic cache channels modulate data by the states of cache regions, and hence a key factor affecting channel bandwidth is the number of regions in a cache being divided. From information theory's perspective, a specific cache region pattern is equivalent to a transmitted symbol. The number of regions in a cache thus corresponds to the number of symbols in the alphabet set. The higher symbol count in an alphabet set, the more information can be passed per symbol.

On hyper-threaded Pentium 4 systems, for which classic cache channels are originally designed, the sender and receiver are executed on the same processor core, using the L1 cache as the transmission medium. The L1 cache on the Pentium 4 processor is addressed purely by virtual memory addresses, a technique called virtually indexed, virtually tagged (VIVT). With a VIVT cache, two processes can impact the same set of associative cache lines by performing memory operations with respect to the same virtual addresses in their address spaces, as illustrated in Fig. 1(a). This property enables processes to precisely control the status of the cache lines, and thus allows for fine division of the L1 cache, such as 32 regions in Percival's cache channel [16].

However, on today's production virtualization systems, hyper-threading is commonly disabled for security reasons (i.e., eliminating hyper-threading induced covert channels).

TABLE I
EXPERIMENTAL SYSTEM CONFIGURATIONS

| | System A | System B |
|---|---|---|
| CPU | Core2 Q8400, 2.66GHz, Caches: (size, set-associativity) L1D – 32KB, 8-way L2 – 2MB, 8-way | 2 * Xeon E5520, 2.26GHz, Caches: (size, set-associativity) L1D – 32KB, 8-way L2 – 256KB, 8-way L3 – 8MB, 16-way |
| Memory | DDR2 DIMM, 1621MHz | DDR3 FBDIMM, 2153MHz |

TABLE II
CACHE LATENCIES VERSUS ACCESS PATTERN LENGTHS

System A (L2 Associative Set Size = 256KB)

| Accesses | 1–6/7 | 8–24/32/64 | More than 64 |
|---|---|---|---|
| Latency | 8 cycles (const.) | +8 cyc./access | +48 cyc./access |

System B (L2 Associative Set Size = 32KB)

| Accesses | 1–18 | 19–64 | More than 96 |
|---|---|---|---|
| Latency | 4 cycles (const.) | +2 cyc./access | +33 cyc./access |

Therefore, the sender and receiver could only communicate by interleaving their executions. This scheduling pattern makes VIVT L1 caches unusable for covert communication because their contents are completely flushed at each context switch. Higher-level caches (e.g., the L2 cache) as well as L1 caches in modern processors (e.g., Intel Nehalem) involve physical memory addresses in their cache line addressing [i.e., physically indexed, physically tagged (PIPT) or virtually indexed, physically tagged (VIPT)], so that contents are preserved across context switches. These caches can be leveraged by classic cache channels.

We demonstrate the effect of physical addressing by conducting experiments on two testbed systems with configurations shown in Table I. We first calculate the *associative block* sizes of the L2 caches by dividing the total cache capacities over their corresponding set-associativity counts. Then, we measure the latencies of a repeating sequence of random memory accesses, with each access spaced multiple *associative blocks* apart. The repeating sequence length starts at one and is incremented by one for each measurement. As shown in Table II, both systems sustain small memory access latencies with up to 64 random accesses. However, if their L2 caches were VIVT, we would have observed large access latency increases beyond 8 random accesses since these caches are 8-way set associative.

In comparison to the VIVT L1 caches, the usability of VIPT/PIPT caches for classic cache channel transmission is much reduced. This is because a normal (i.e., unprivileged) process only has knowledge of its own virtual address space, which is usually mapped to nonlinear physical address spaces. As a result, normal processes cannot be completely certain of the cache line that a memory access would affect due to address translation. Server virtualization has further complicated the addressing uncertainty by adding another layer of indirection to memory addressing. As illustrated in Fig. 1(b), even the "physical memory" of a guest VM is virtualized, and access to it must be further translated. As a result, it is very difficult, if not impossible, for a process in a guest VM (especially for a full virtualization VM) to discover the actual physical memory addresses of a memory region.

Due to the addressing uncertainty, for classic covert channels on virtualized systems, the number of cache regions is reduced to a minimum of only two [2], [4].

## B. Scheduling Uncertainty

Classic cache channel data transmission depends on a cache pattern "round-trip"—the receiver completely resets the cache and correctly passes it to the sender, and the sender completely prepares the cache pattern and correctly passes it back to the receiver. Therefore, to successfully transmit one cache pattern, the sender and receiver must be strictly round-robin scheduled.

However, without special scheduling arrangements (i.e., collusion) from the hypervisor, such idealistic scheduling rarely happens. On production virtualized systems, the physical processors are usually heavily multisubscribed in order to increase utilization. In other words, each physical processing core serves more than one virtual processor from different VMs. As a result, there exist many scheduling patterns that prevent successful cache pattern "round trip," such as the following.

- *Channel not cleared for send*: The receiver is descheduled before it finishes resetting the cache.
- *Channel invalidated for send*: The receiver finishes resetting the cache, but another unrelated VM is scheduled to run immediately after.
- *Sending incomplete*: The sender is descheduled before it finishes preparing the cache.
- *Symbol destroyed*: The sender finishes preparing the cache, but another unrelated VM is scheduled to run immediately after.
- *Receiving incomplete*: The receiver is descheduled before it finishes reading the cache.
- *Channel access collision*: The sender and receiver are executed in parallel on processor cores that share cache.

Xu *et al.* [4] have clearly illustrated the problem of scheduling uncertainty in two of their measurements. First, in a laboratory setup, the error rate of their covert channel increases from near 1% to 20%–30% after adding just one nonparticipating VM with moderate workload. Second, in the Amazon EC2 cloud, they have discovered that only 10.5% of the cache measurements at the receiver side are valid for data transmission, due to the fact that the hypervisor's scheduling is different from the idealistic scheduling.

## C. Cache Physical Limitation

Besides the two uncertainties, classic cache channels also face an insurmountable limitation—the necessity of a *shared* and *stable* cache.

If the sender and receiver of classic cache channels are executed on processor cores that do not share any cache, obviously no communication could be established. On a multiprocessor system, it is quite common to have processor cores that do not share any cache since there is usually no shared cache between different physical processors. Smetimes even processor cores residing on the same physical processor do not share any cache, such as the Intel Core2 Quad processor, which contains two dual-core silicon packages with no shared cache in between.

Even if the sender and receiver could share a cache, external interferences can destabilize the shared cache. Modern multicore processors often include a large last-level cache (LLC)

**Algorithm 2** Timing-based Cache Channel Protocol

```
$CLines$: Several sets of associative cache lines picked by both the sender and the receiver;
          These cache lines can be put in one of two states, cached or flushed.
$D_{Send}[N]$, $D_{Receive}[N]$: $N$ bit data to transmit and receive, respectively.

Sender Operations:
for $i := 0$ to $N-1$ do
  if $D_{Send}[i] = 1$ then
    for an amount of time do
      {Put $CLines$ into the flushed state}
      Access memory maps to $CLines$;
    end for
  else
    {Leave $CLines$ in the cached state}
    Sleep of an amount of time;
  end if
end for

Receiver Operations:
for $i := 0$ to $N-1$ do
  for an amount of time do
    Timed access memory maps to $CLines$;
  end for
  {Detect the state of $CLines$ by latency}
  if $Mean(AccessTime) > Threshold$ then
    $D_{Receive}[i] := 1$; {$CLines$ is flushed}
  else
    $D_{Receive}[i] := 0$; {$CLines$ is cached}
  end if
end for
```

shared between all processor cores. To facilitate a simpler cache coherence protocol, the LLC usually employs an inclusive principle, which requires that all data contained in the lower-level caches must also exist in the LLC. In other words, when a cache line is evicted from the LLC, it must also be evicted from all the lower-level caches. Thus, any nonparticipating processes executing on those processor cores that share the LLC with the sender and receiver can interfere with the communication by indirectly evicting the data in the cache used for the covert channel. The more cores on a processor, the higher the interference.

Overall, virtualization induced changes to cache operations and process scheduling render the data transmission scheme of classic cache channels obsolete. First, the effectiveness of data modulation is severely reduced by addressing uncertainty. Second, the critical procedures of signal generation, delivery, and detection are frequently interrupted by less-than-ideal scheduling patterns. Finally, the fundamental requirement of stably shared cache is hard to satisfy as processors are having more cores.

## IV. Covert Channel in the Hyper-Space

In this section, we present our techniques to tackle the existing difficulties and develop a high-bandwidth, reliable covert channel on virtualized x86 systems. We first describe our redesigned, pure timing-based data transmission scheme, which overcomes the negative effects of addressing and scheduling uncertainties with a simplified design. After that, we detail our findings of a powerful covert channel medium, exploiting the atomic instructions and their induced cache-memory bus interactions on x86 platforms. Finally, we specify our designs of a high error-tolerance transmission protocol for cross-VM covert channels.

### A. Redesigning Data Transmission

We first question the reasoning behind using cache state patterns for data modulation. Originally, Percival [16] designed this transmission scheme mainly for the use of side-channel cryptographic key stealing on a hyper-threaded processor. In this specific usage context, the critical information of memory access patterns are reflected by the states of cache regions. Therefore, cache region-based data modulation is an important source of information. However, in a virtualized environment, the regions of the cache no longer carry useful information due to addressing uncertainty, making cache region-based data modulation a great source of interference.

We therefore redesign a data transmission scheme for the virtualized environment. Instead of using the cache region-based encoding scheme, we modulate data based on the state of cache lines over time, resulting in a pure timing-based transmission protocol, as described in Algorithm 2.

Besides removing cache region-based data modulation, the new transmission scheme also features a significant change in the scheduling requirement, i.e., signal generation and detection are performed instantaneously, instead of being interleaved. In other words, data are transmitted while the sender and receiver run in parallel. This requirement is more lenient than strict round-robin scheduling, especially with the trend of increasing number of cores on a physical processor, making two VMs more likely to run in parallel than interleaved.

We conduct a simple raw bandwidth estimation experiment to demonstrate the effectiveness of the new cache covert channel. In this experiment, interleaved bits of zeros and ones are transmitted, and the raw bandwidth of the channel can thus be estimated by manually counting the number of bits transmitted over a period of time.

We build the cache covert channel on our testbed System A. Based on our experimental observations in Section III-A, we select *CLines* as a set of 64 cache lines mapped by memory addresses following the pattern $M + X \cdot 256K$, where $M$ is a small constant and $X$ is a random positive integer. The sender puts these cache lines into the *flushed* state by accessing a sequence of *CLines*-mapping memory addresses. The receiver times the access latency of another sequence of *CLines*-mapping memory addresses. The length of the receivers access sequence should be smaller than, but not too far away from, the cache line set size, for example, 48.

As shown in Fig. 2, the $x$-value of each sample point is the observed memory access latency by the receiver, and the trend line is created by plotting the moving average of two samples. According to the measurement results, 39 bits can be transmitted over a period of 200 $\mu$s, yielding a raw bandwidth of over 190.4 kb/s, about five orders of magnitude higher than the previously studied cross-VM cache covert channels.

Having resolved the negative effects of addressing and scheduling uncertainties and achieved a high raw bandwidth, our new cache covert channel, however, still performs poorly on systems with nonparticipating workloads. We discover that the sender

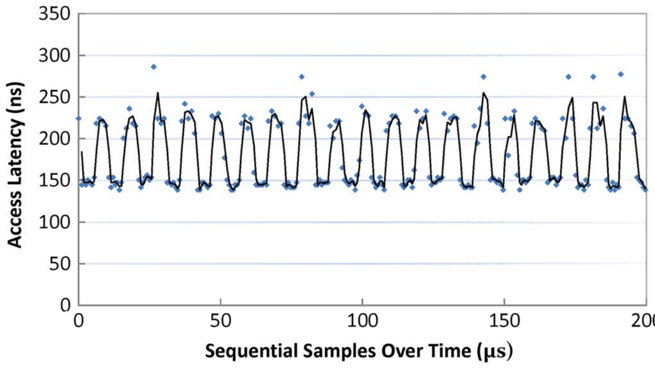


Fig. 2. Timing-based cache channel bandwidth test.

and receiver have difficulty in establishing a stable communication channel. The cause of instability is that the hypervisor frequently migrates the virtual processors across physical processor cores, which is also observed by Xu *et al.* [4]. The outgrowth of this behavior is that the sender and receiver frequently reside on processor cores that do not share any cache, making our cache channel run into the insurmountable cache physical limitation just like the classic cache channels.

## B. (Re)Discovering the Bus-Contention Channel

The prevalence of virtual processor core migration handicaps cache channels in cross-VM covert communication. In order to reliably establish covert channels across processor cores that do not share any cache, a commonly shared and exploitable resource is needed as the communication medium. The memory bus comes into our sight as we extend our scope beyond the processor cache.

*1) Background:* Interconnecting the processors and the system main memory, the memory bus is responsible for delivering data between these components. Because contention on the memory bus results in a system-wide observable effect of increased memory access latency, a covert channel can be created by programmatically triggering contention on the memory bus. Such a covert channel is called a bus contention channel.

The bus contention channels have long been studied as a potential security threat for virtual machines on the VAX VMM, on which a number of techniques have been developed [26], [28], [29] to effectively mitigate this threat. However, the x86 platforms we use today are significantly different from the VAX systems, and we suspect similar exploits can be found by probing previously unexplored techniques. Unsurprisingly, by carefully examining the memory-related operations of the x86 platform, we have discovered a bus-contention exploit using atomic instructions with exotic operands.

Atomic instructions are special x86 memory manipulation instructions, designed to facilitate multiprocessor synchronization, such as implementing mutexes and semaphores—the fundamental building blocks for parallel computation. Memory operations performed by atomic instructions (namely, atomic memory operations) are guaranteed to complete uninterrupted because accesses to the affected memory regions by other processors or devices are temporarily blocked from execution.

*2) Analysis:* Atomic memory operations, by their design, generate system-wide observable contentions in the target memory regions on which they operate. This particular feature of atomic memory operations caught our attention. Ideally, contention generated by an atomic memory operation is well bounded and is only evident when the affected memory region is accessed in parallel. Thus, atomic memory operations are not exploitable for cross-VM covert channels because VMs normally do not implicitly share physical memory. However, we have found out that the hardware implementations of atomic memory operations do not match the idealistic specification, and memory contentions caused by atomic memory operations could propagate much further than expected.

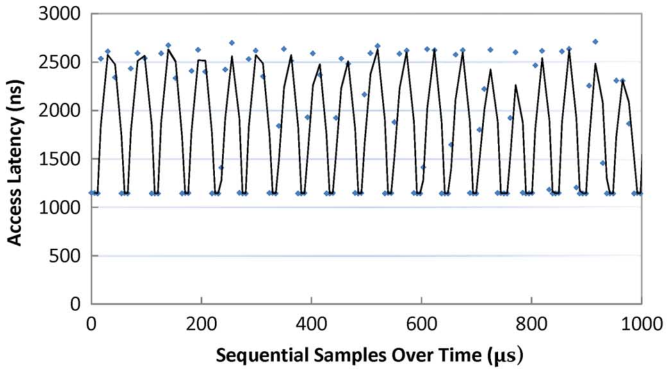
Early generations (before Pentium Pro) of x86 processors implement atomic memory operations by using bus lock, a dedicated hardware signal that provides exclusive access of the memory bus to the device who asserts it. While providing a very convenient means to implement atomic memory operations, the sledgehammer-like approach of locking the memory bus results in system-wide memory contention. In addition to being exploitable for covert channels, the bus-locking implementation of atomic memory operations also causes performance and scalability problems.

Modern generations (before Intel Nehalem and AMD K8/K10) of x86 processors improve the implementation of atomic memory operations by significantly reducing the likelihood of memory bus locking. In particular, when an atomic operation is performed on a memory region that can be entirely cached by a cache line, which is a very common case, the corresponding cache line is locked, without locking the memory bus [30]. However, atomic memory operations can still be exploited for covert channels because the triggering conditions for bus locking are not eliminated. Specifically, when atomic operations are performed on memory regions with an exotic[3] configuration—unaligned addresses that span two cache lines—atomicity cannot be ensured by cache line locking, and bus lock signals are thus asserted.

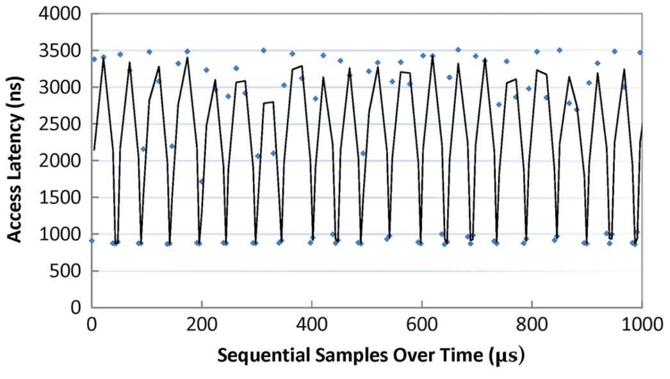
Remarkable architecture evolutions have taken place in the latest generations (Intel Nehalem and AMD K8/K10) of x86 processors, one of which is the removal of the shared memory bus. On these platforms, instead of having a unified central memory storage for the entire system, the main memory is divided into several pieces, each assigned to a processor as its local storage. While each processor has direct access to its local memory, it can also access memory assigned to other processors via a high-speed interprocessor link. This nonuniform memory access (NUMA) design eliminates the bottleneck of a single shared memory bus and, thus, greatly improves processor and memory scalability. As a side effect, the removal of the shared memory bus has seemingly invalidated memory bus covert channel techniques at their foundation. Interestingly, however, the exploit of atomic memory operation continues to work on the newer platforms, and the reason for this requires a bit more in-depth explanation.

On the latest x86 platforms, normal atomic memory operations (i.e., operating on memory regions that can be cached by a single cache line) are handled by the cache-line locking

[3]The word "exotic" here only means that it is very rare to encounter such an unaligned memory access in modern programs, due to automatic data field alignments by the compilers. However, manually generating such an access pattern is very easy.



(a)



(b)

Fig. 3. Timing-based memory bus channel bandwidth tests. (a) Intel Core2, Hyper-V, Windows guest VMs. (b) Intel Xeon (Nehalem), Xen, Linux guest VMs.

**Algorithm 3** Timing-based Memory Bus Channel Protocol

```
$M_{Exotic}$: An exotic configuration of a memory region that spans two cache lines.
$D_{Send}[N]$, $D_{Recv}[N]$: $N$ bit data to transmit and receive, respectively.

Sender Operations:
for $i := 0$ to $N-1$ do
    if $D_{Send}[i] = 1$ then
        for an amount of time do
            {Put memory bus into contended state}
            Perform atomic operation with $M_{Exotic}$;
        end for
    else
        {Leave memory bus in contention-free state}
        Sleep of an amount of time;
    end if
end for

Receiver Operations:
for $i := 0$ to $N-1$ do
    for an amount of time do
        Timed uncached memory access;
    end for
    {Detect the state of memory bus by latency}
    if $Mean(AccessTime) > Threshold$ then
        $D_{Recv}[i] := 1$; {Bus is contended}
    else
        $D_{Recv}[i] := 0$; {Bus is contention-free}
    end if
end for
```

mechanism similar to that of the previous-generation processors. However, for exotic atomic memory operations (i.e., operating on cache-line-crossing memory regions), because there is no shared memory bus to lock, the atomicity is achieved by a set of much more complex operations: All processors must coordinate and completely flush in-flight memory transactions that are previously issued. In a sense, exotic atomic memory operations are handled on the newer platform by "emulating" the bus locking behavior of the older platforms. As a result, the effect of memory access delay is still observable, despite the absence of the shared memory bus.

*3) Verification:* With the memory bus exploit, we can easily build a memory bus covert channel by adapting our timing-based cache transmission scheme with minor modifications, as shown in Algorithm 3.

Compared to Algorithm 2, there are only two differences in the memory bus channel protocol. First, we substitute the set of cache lines (*CLines*) with the memory bus as the transmission medium. Similar to the cache lines, the memory bus can also be put in two states, *contended* and *contention-free*, depending on whether exotic atomic memory operations are performed. Second, instead of trying to evict contents of the selected cache lines, the sender changes the memory bus status by performing exotic atomic memory operations. Correspondingly, the receiver must make uncached memory accesses to detect contentions.

We demonstrate the effectiveness of the memory bus channel by performing bandwidth estimation experiments, similar to the one in Section IV-A, on our testbed systems running different generations of platforms, hypervisors, and guest VMs. Specifically, System A uses an older shared memory bus platform and runs Hyper-V with Windows guest VMs, while System B utilizes the newer platform without a shared memory bus and runs Xen with Linux guest VMs. As Fig. 3 shows, the $x$-value of each sample point is the observed memory access latency by the receiver, and the trend lines are created by plotting the moving average of two samples. According to the measurement results, on both systems, 39 bits can be transmitted over a period of 1 ms, yielding a raw bandwidth of over 38 kb/s. Although an order of magnitude lower in bandwidth than our cache channel, the memory bus channel enjoys its unique advantage of working across different physical processors. Notably, the same covert channel implementation works on both systems, regardless of the guest operating systems, hypervisors, and hardware platform generations.

## C. Whispering Into the Hyper-Space

We have demonstrated that the memory bus channel is capable of achieving high-speed data transmission on virtualized systems. However, the preliminary protocol described in Algorithm 3 is prone to errors and failures in a realistic environment because the memory bus is a very noisy channel, especially on virtualized systems running many nonparticipating workloads.

Fig. 4 presents a realistic memory bus channel sample, taken using a pair of physically co-resident VMs in the Amazon EC2 cloud. From this figure, we can observe that both the "contention-free" and "contended" signals are subject to frequent interferences. The "contention-free" signals are intermittently disrupted by workloads of other nonparticipating VMs, causing the memory access latency to moderately raise above the baseline.

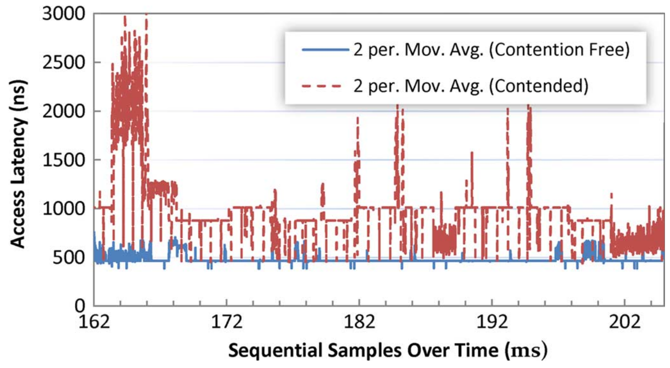


Fig. 4. Memory bus channel quality sample in EC2.

In contrast, the "contended" signals experience much heavier interferences, which originate from two sources: scheduling and nonparticipating workloads. The scheduling interference is responsible for the periodic drop of memory access latency. In particular, context switches temporarily deschedule the sender process from execution, thereby briefly relieving memory bus contention. The nonparticipating workloads executed *in parallel* with the sender process worsen memory bus contention and cause the spikes in the figure, while nonparticipating workloads executed *concurrently* with the sender process reduce memory bus contention and result in the dips in the figure. All these interferences can degrade the signal quality in the channel and make what the receiver observes different from what the sender intends to generate, which leads to *bit-flip* errors.

Besides the observable interferences shown in Fig. 4, there are also unobservable interferences, i.e., the scheduling interferences to the receiver, which can cause an entirely different phenomenon. When the receiver is descheduled from execution, there is no observer in the channel, and thus all data being sent is lost. To make matters worse, the receiver could not determine the amount of information being lost because the sender may also be descheduled during that time. As a result, the receiver suffers from *random erasure* errors.

Therefore, three important issues need to be addressed by the communication protocol in order to ensure reliable cross-VM communication: clock synchronization, receiving confirmation, and error correction.

*Clock Synchronization:* Since the sender and receiver belong to two independent VMs, scheduling differences between them tend to make the data transmission and detection procedures desynchronized, which can cause a significant problem to pure timing-based data modulation. We overcome clock desynchronization by using self-clocking coding—a commonly used technique in telecommunications. Here, we choose to transmit data bits using differential Manchester encoding, a standard network coding scheme [31].

*Receiving Confirmation:* The *random erasure* errors caused by receiver descheduling can make the transmitted data very discontinuous, significantly reducing its usefulness. To alleviate this problem, it is very important for the sender to be aware of whether the data it sent out has been received.

We avoid using message-based "send-and-ack," a commonly employed mechanism for solving this problem, because this mechanism requires the receiver to actively send data back to the sender, reversing the roles of sending and receiving, and thus subjects the acknowledgment sender (i.e., the data receiver) to the same problem. Instead, we leverage the system-wide effect of memory bus contention to achieve simultaneous data transmission and receiving confirmation. In particular, the sender infers the presence of receiver by observing increased memory access latencies generated by the receiver. The corresponding changes to the data transmission protocol include the following.

1) Instead of making uncached memory accesses, the receiver performs exotic atomic memory operations, just like the sender transmitting a one bit.
2) Instead of sleeping when transmitting a zero bit, the sender performs uncached memory accesses. In addition, the sender always times its memory accesses.
3) While the receiver is in execution, the sender should always observe high memory access latencies; otherwise, the sender can assume the data has been partially lost, and retry at a later time.

*Error Correction:* Due to interfering system workload and scheduling variations, *bit-flip* errors are expected to be common. Similar to resolving the receiving confirmation problem, we again avoid using acknowledgment-based mechanisms. Assuming only a one-way communication channel, we resolve the error correction problems by applying forward error correction (FEC) to the original data, before applying self-clocking coding. More specifically, we use the Reed–Solomon coding [32], a widely applied block FEC code with strong multibit error correction performance.

In addition, we strengthen the communication protocol's resilience to clock drifting and scheduling interruption by employing data framing. We break the data into segments of fixed-length bits and frame each segment with a start-and-stop pattern. The benefits of data framing are twofold. First, when the sender detects transmission interruption, instead of retransmitting the whole piece of data, only the affected data frame is retried. Second, some data will inevitably be lost during transmission. With data framing, the receiver can easily localize the erasure errors and handle them well through the Reed–Solomon coding.

The finalized protocol with all the improvements in place is presented in Algorithm 4.

## V. Evaluation

We evaluate the exploitability of memory bus covert channels by implementing the reliable Cross-VM communication protocol and demonstrate covert channel attacks on our in-house testbed server, as well as on the Amazon EC2 cloud.

### A. In-House Experiments

We launch covert channel attacks on our testbed System B, which is equipped with the latest-generation x86 platform (i.e., with no shared memory bus). The experimental setup is simple and realistic. We create two Linux VMs, namely VM-1 and VM-2, each with a single virtual processor and 512 MB of memory. The covert channel sender runs as an unprivileged user program on VM-1, while the covert channel receiver runs on VM-2, also as an unprivileged user program.

We first conduct a quick profiling to determine the optimal data frame size and error correction strength. We find out that a data frame size of 32 bits (including an 8-bit preamble) and

**Algorithm 4** Reliable Timing-based Memory Bus Channel Protocol

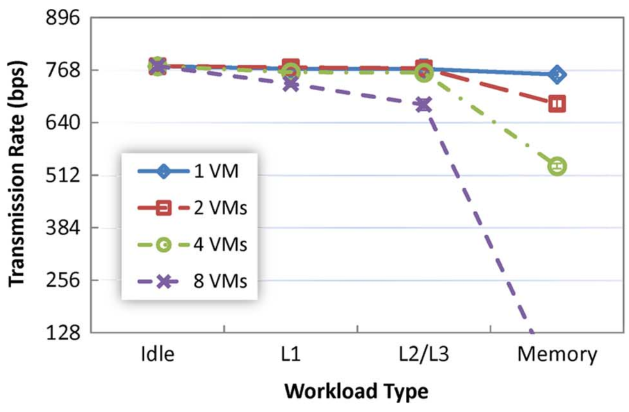
```
$M_{ExoticS}$, $M_{ExoticR}$: Exotic memory regions for the sender and the receiver, respectively.
$D_{Send}$, $D_{Recv}$: Data to transmit and receive, respectively.

Sender Prepares $D_{Send}$ by:
{$DM_{Send}$[]: Segmented encoded data to send}
$RS_{Send}$ := ReedSolomon$_{Encode}(D_{Send})$;
$FD_{Send}$[] := Break $RS_{Send}$ into segments;
$DM_{Send}$[] := DiffManchester$_{Encode}(FD_{Send}[])$;

Receiver Recovers $D_{Recv}$ by:
{$DM_{Recv}$[]: Segmented encoded data received}
$FD_{Recv}$[] := DiffManchester$_{Decode}(DM_{Recv}[])$;
$RS_{Recv}$ := Concatenate $FD_{Recv}$[];
$D_{Recv}$ := ReedSolomon$_{Decode}(RS_{Recv})$;
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Sending Encoded Data in a Frame:
{$Data$: A segment of encoded data to send}
{$FrmHead$, $FrmFoot$: Unique bit patterns
 signifying start and end of frame, respectively}
$Result := SendBits(FrmHead)$;
if $Result$ is not $Aborted$ then
  $Result := SendBits(Data)$;
  if $Result$ is not $Aborted$ then
    {Ignore error in sending footer}
    $SendBits(FrmFoot)$;
    return $Succeed$;
  end if
end if
return $Retry$;

Receiving Encoded Data in a Frame:
{$Data$: A segment of encoded data to receive}
Wait for frame header;
$Result := RecvBits(Data)$;
if $Result$ is $Aborted$ then
  return $Retry$;
end if
$Result$ := Match frame footer;
if $Result$ is not $Matched$ then
  {Clock synchronization error, discard $Data$}
  return $Erased$;
else
  return $Succeed$;
end if
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Sending a Block of Bits:
{$Block$: A block of bits to send}
{$Base_1$, $Base_0$: Mean contention-free access
 time for sending bit 1 and 0, respectively}
for each $Bit$ in $Block$ do
  if $Bit = 1$ then
    for an amount of time do
      Timed atomic operation with $M_{ExoticS}$;
    end for
    $Latency := Mean(AccessTime) - Base_1$;
  else
    for an amount of time do
      Timed uncached memory access;
    end for
    $Latency := Mean(AccessTime) - Base_0$;
  end if
  if $Latency < Threshold$ then
    {Receiver not running, abort}
    return $Aborted$;
  end if
end for
return $Succeed$;

Receiving a Block of Bits:
{$Block$: a block of bits to receive}
for each $Bit$ in $Block$ do
  for an amount of time do
    Timed atomic operation with $M_{ExoticR}$;
  end for
  {Detect the state of memory by latency}
  if $Mean(AccessTime) > Threshold$ then
    $Bit$ := 1; {Bus is contended}
  else
    $Bit$ := 0; {Bus is contention-free}
  end if
  {Detect sender de-schedule}
  if too many consecutive 0 or 1 bits then
    {Sender not running}
    Sleep for some time;
    {Sleep makes sender abort, then we abort}
    return $Aborted$;
  end if
end for
return $Succeed$;
```

a ratio of 4 parity symbols (bytes) per 4 data bytes works well. Effectively, each data frame transmits 8 bits of preamble, 12 bits of data, and 12 bits of parity, yielding an efficiency of 37.5%. In order to minimize the impact of burst errors, such as multiple frame losses, we group 48 data and parity bytes and randomly distribute them across 16 data frames using a linear congruential generator (LCG).

We then assess the capacity (i.e., bandwidth and error rate) of the covert channel by performing a series of data transmissions using these parameters. For each transmission, a 1-kB data block is sent from the sender to the receiver. With 50 repeated transmissions, we observe a stable transmission rate of 746.8±10.1 b/s. Data errors are observed, but at a very low rate of 0.09%.

We further evaluate the impact of covert channel performance by interfering workload, in particular, the workload on the memory subsystem, from nonparticipating VMs ("other VMs" for short). We define four levels of interferences, *idle*, *L1*, *L2/L3*, and *Memory*, listed in ascending order by the weight of impact to the memory subsystem. The *idle* interference is generated by spawning other VMs and leaving them idle. The *L1* interference is generated by running in the other VMs a program with a tight infinite loop, which only stresses the processor L1 cache due to the very small amount of memory involved in execution. Both *L2/L3* and *Memory* interferences are generated by running `cachebench` [33], a processor cache and memory benchmarking utility: For the *L2/L3* interference, the amount of memory access is limited to the size of the processor L3 cache; for the *Memory* interference, the amount of memory access is set to be slightly larger than the size of the processor L3 cache.

As shown in Fig. 5, we measure the bandwidth and error rate of the covert channel when it is subjected to each level of interferences generated by up to eight nonparticipating VMs. We observe that the covert channel is very resilient to *idle*, *L1*, and *L2/L3* interferences. More specifically, while these interferences do exert negative impacts on the covert channel (i.e., decreased bandwidths and increased error rates), the effects are minimal—except for the moderate decrease of bandwidth with eight VMs running *L2/L3* workload, the bandwidth and error rate reductions in all other cases are negligible. The robustness against cache-based interferences is well expected since



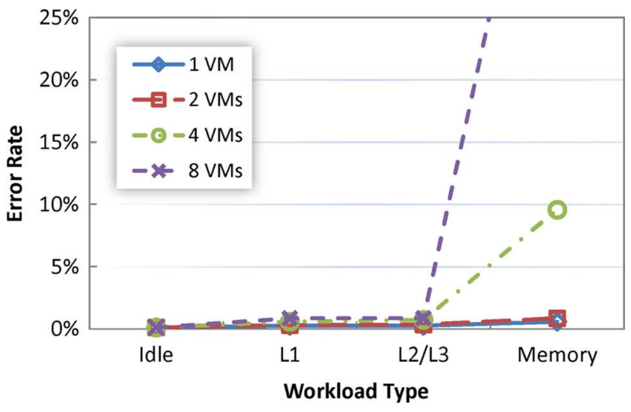


Fig. 5. Effects of nonparticipating workload on bandwidth and error rate.

the processor cache is not used as a medium for this covert channel. However, when subjected to *Memory* interferences, the covert channel performances degrades significantly with more than four VMs running nonparticipating workload. Especially, with eight VMs, no data could be transmitted without uncorrectable error (i.e., the error rate approaches 50% and the transmission rate drops to near zero). This dramatic reduction of performance is also well expected because the memory benchmark program inflicts extreme workload on the memory bus, thereby rendering this medium unusable for the covert channel. Because normal applications would rarely generate such an intense memory workload for an extended period of time, the memory bus covert channel is still practical in the real world.

### B. Amazon EC2 Experiments

We prepare the Amazon EC2 experiments by spawning physically co-hosted Linux VMs. Following instructions presented in [2] and [4], we uncover several pairs of physically co-hosted VM instances. Information disclosed in `/proc/cpuinfo` shows that the host servers use the shared-memory-bus platform, one generation older than our testbed server used in the previous experiment.

Similar to our in-house experiments, we first conduct a quick profiling to determine the optimal data frame size and error correction strength. Compared to our in-house system profiles, memory bus channels on Amazon EC2 VMs have a higher tendency of clock desynchronization. We compensate for this deficiency by reducing the data frame size to 24 bits. The error correction strength of 4 parity symbols per 4 data bytes still works well. The overall transmission efficiency thus becomes 33.3%.

We again perform a series of data transmissions and measure the bandwidth and error rates. For small EC2 instances, our covert channel achieves a stable performance of $300.32 \pm 38.79$ b/s, with an error rate of 0.50%. However, for micro EC2 instances, we observe more interesting results. As illustrated in Fig. 6, the covert channel performance exhibits three distinct stages as the experiments progress. During the initial 12–15 rounds of experiments, we measure a transmission rate of $343.5 \pm 66.1$ b/s, with an error rate of 0.39%. As we continue to repeat the measurements, the covert channel performance degrades. For the next five to eight rounds of experiments, the bandwidth slightly reduces, and the error rate slightly increases. Finally, if we continue with the experiments, the covert channel performance becomes very bad. While the bandwidth is still comparable to that of the best performance, the error rate becomes unacceptably high.

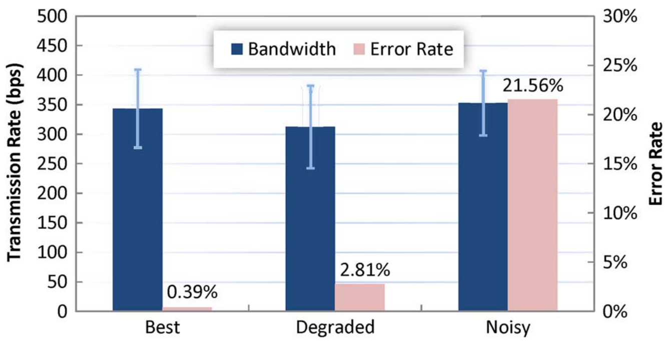


Fig. 6. Memory bus channel performances on EC2 micro instances.

By repeating this experiment, we uncover that the three-staged behavior can be repeatedly observed after leaving both VMs idle for a long period of time (e.g., 20 min to 1 h). We believe, according to the Amazon EC2 documentation [34], the cause of this behavior can be explained by the CPU level limiting for micro-instance VMs.[4] During our initial transmissions, both the sender and receiver VMs are allowed to run at full speed, and thus they are very likely to execute in parallel, resulting in good channel performance. Then, either the sender VM or receiver VM depletes its allotted processor resource and is throttled back, causing the channel performance to degrade. Soon after that, the other VM also consumes its quota and is throttled back. As a result, the communication channel is heavily interrupted due to this "unfriendly" scheduling pattern, which injects errors and random erasure beyond the correction capability of the FEC mechanism.

Because our communication protocol is designed to handle very unreliable channels, we further investigate how to maintain covert channel reliability when one or both parties are being throttled back. We are able to find a working solution by tuning only two parameters. First, we increase the ratio of parity bits

[4]Each EC2 micro instance is only allowed to use a small amount of CPU resource in a certain period of time. When a VM occupies a processor for a prolonged time span, it uses up all of its "processor quota," and the VM is put through a "throttled back" period, during which it is only allowed a small fraction of a processor's resource.

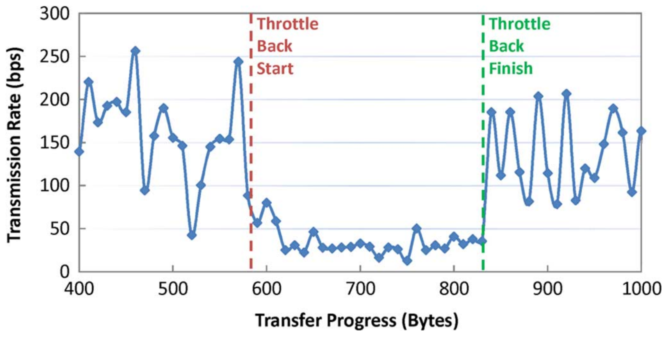


Fig. 7. Reliable transmission with adaptive rates on EC2 micro instances.

to 4 parity symbols per 2 data bytes. Although it reduces transmission efficiency by 11.1%, the error correction capability of our FEC is increased by 33.3%. Second, we reduce the transmission symbol rate by about 20%. By lengthening the duration of the receiving confirmation, we effectively increase the probability of discovering scheduling interruptions. After the parameter adjustments, we are able to achieve a transmission rate of 107.9±39.9 b/s, with an error rate of 0.75%, even through periods in which both the sender and the receivers are throttled back.

Fig. 7 depicts the adjusted communication protocol in action. During the first period of full-speed running, the transmission rate can be as high as over 250 b/s. However, when throttle back starts, the sender responds to frequent transmission failures with increased retries, allowing the receiver continue to receive and decode data without uncorrectable error. Correspondingly, the transmission rate drops to below 50 b/s. Finally, when the harsh scheduling condition is alleviated, the transmission rate is automatically restored. The capability of adaptively adjusting transmission rates according to channel conditions evidences the versatility of our reliable communication protocol.

## VI. Discussion

In this section, we first show a breakdown of impact to cover channel performance from our communication protocol features. Then, we reassess the threat of covert channel attacks based on our experimental results. Finally, we discuss possible means to mitigate the covert channel attacks in virtualized environments.

### A. Performance Impact

Our communication protocol design consists of three main features, clock synchronization, receiving confirmation, and error correction. Each feature contributes differently to the covert channel performance.

Clock synchronization plays a fundamental role in the communication protocol, without which we could not reliably transmit meaningful length of data. The bandwidth cost of clock synchronization is simple to derive. With differential Manchester encoding, each bit requires a transition edge to encode, and thus reduces the usable bandwidth by 50%.

Receiving confirmation ensures the information transmitted by the sender reaches the receiver. We observe the bandwidth impact of receiving confirmation in our EC2 small instance experiment. On average, the sender retransmits 5.27 times when sending each data frame: 4.11 retransmissions occur on the preamble byte, and 1.26 retransmissions occur on the first data byte. As a result, approximately 6.63 B of information are invalidated for every successful transmission of 3 B, yielding an empirical efficiency of 31.15%.

Error correction serves as a final measure to ensure the correctness of transmitted data. While the encoding efficient analysis of error correction is provided in Section V, we report its empirical contribution to channel reliability in our EC2 small instance experiment. We observe that on average one error correction is invoked for every 3.11 blocks of data, yielding a raw error rate of 32.15%. Compared to the data error rate of 0.5% after the error correction, the error correction has recovered 98.44% of the errors.

### B. Damage Assessment

We extrapolate the threat of the memory bus covert channel from three different aspects—achievable attacks, mitigation difficulties, and cross-platform applicability.

*1) Achievable Attacks:* Due to their very low channel capacities [2], [4], previous studies conclude that covert channels can only cause very limited harms in a virtualized environment. However, the experimental results of our covert channel lead us to a different conclusion that covert channels indeed pose realistic and serious threats to information security in the cloud.

With over 100 b/s high-speed and reliable transmission, covert channel attacks can be applied to a wide range of mass-data theft attacks. For example, a 100-B credit card data entry can be silently stolen in less than 30 s; a 1000-B private key file can be secretly transmitted under 3 min. Working continuously, over 1 MB of data, equivalent to tens of thousands of credit card entries or hundreds of private key files, can be trafficked every 24 h.

*2) Mitigation Difficulties:* In addition to high channel capacity, the memory bus covert channel has two other intriguing properties that make it difficult to be detected or prevented.

- *Stealthiness*: Because processor cache is not used as channel medium, the memory bus covert channel incurs negligible impact on cache performance, making it totally transparent to cache-based covert channel detection, such as HomeAlone [3].
- *"Future-proof"*: Our in-house experiment shows that even on a platform that is one generation ahead of Amazon EC2's systems, the memory bus covert channel continues to perform very well.

*3) Cross-Platform Applicability:* Due to hardware availability, we have only evaluated memory bus covert channels on the Intel x86 platforms. On one hand, we make an intuitive inference that similar covert channels can also be established on the AMD x86 platforms since they share compatible specifications on atomic instructions with the Intel x86 platforms. On the other hand, the atomic instruction exploits may not be applicable on platforms that use alternative semantics to guarantee operation atomicity. For example, MIPS and several other platforms use the load-linked/store-conditional paradigm, which does not result in high memory bus contention as atomic instructions do.

### C. Mitigation Techniques

The realistic threat of covert channel attacks calls for effective and practical countermeasures. We discuss several plausible mitigation approaches from three different perspectives—tenants, cloud providers, and device manufactures.

*1) Tenant Mitigation:* Mitigating covert channels on the tenant side enjoys the advantages of trust and deployment flexibility. With the implementation of mitigation techniques inside tenant-owned VMs, the tenant has the confidence of covert channel security, regardless whether the cloud provider addresses this issue.

However, due to the lack of lower-level (hypervisor and/or hardware) support, the available options are very limited, and the best choice is performance anomaly detection. Although not affecting the cache performances, memory bus covert channels do cause memory performance degradation. Therefore, an approach similar to that of HomeAlone [3] could be taken. In particular, the defender continuously monitors memory access latencies and asserts alarms if significant anomalies are detected. However, since memory accesses incur much higher cost and nondeterminism than cache probing, this approach may suffer from high performance overhead and high false positive rate.

*2) Cloud Provider Mitigation:* Compared to their tenants, cloud providers are much more resourceful. They control not only the hypervisor and hardware platform on a single system, but also the entire network and systems in a data center. As a result, cloud providers can tackle covert channels through either preventative or detective countermeasures.

The preventative approaches, e.g., the dedicated instances service provided by the Amazon EC2 cloud [25], thwart covert channel attacks by eliminating the exploiting factors of covert channels. While the significant extra service charge of the dedicated instance service reduces its attractiveness, the "no-sharing" guarantee may be too strong for covert channel mitigation. We envision a low-cost alternative solution that allows tenants to share system resources in a controlled and deterministic manner. For example, the cloud provider may define a policy that each server might be shared by up to two tenants, and each tenant could only have a predetermined neighbor. Although this solution does not eliminate covert channels, it makes attacking arbitrary tenants in the cloud very difficult.

In addition to preventative countermeasures, cloud providers can easily take the detective approach by implementing low-overhead detection mechanisms because of their convenient access to the hypervisor and platform hardware. For both cache and memory bus covert channels, being able to generate observable performance anomalies is the key to their success in data transmission. However, modern processors have provided a comprehensive set of mechanisms to monitor and discover performance anomalies with very low overhead. Instead of actively probing cache or accessing memory, cloud providers can leverage the hypervisor to infer the presence of covert channels, by keeping track of the increment rates of the cache miss counters or memory bus lock counters [30]. Moreover, when suspicious activities are detected, cloud providers can gracefully resolve the potential threat by migrating suspicious VMs onto physically isolated servers. Without penalizing either the suspect or the potential victims, the negative effects of false positives are minimized.

*3) Device Manufacture Mitigation:* The defense approaches of both tenant and cloud providers are only secondary in comparison to mitigation by the device manufactures because the root causes of the covert channels are imperfect isolation of the hardware resources.

The countermeasures at the device manufacture side are mainly preventative, and they come in various forms of resource isolation improvements. For example, instead of handling exotic atomic memory operations in hardware and causing system-wide performance degradation, the processor may be redesigned to trap these rare situations for the operating systems or hypervisors to handle, without disrupting the entire system. A more general solution is to tag all resource requests from guest VMs, enabling the hardware to differentiate requests by their owner VMs, and thereby limiting the scope of any performance impact.

While incurring high cost in hardware upgrades, the countermeasures at the device manufacture side are transparent to cloud providers and tenants and can potentially yield the lowest performance penalty and overall cost compared to other mitigation approaches.

## VII. Conclusion and Future Work

Covert channel attacks in the cloud have been proposed and studied. However, the threats of covert channels tend to be downplayed or disregarded due to the low achievable channel capacities reported by previous research. In this paper, we presented a novel construction of high-bandwidth and reliable cross-VM covert channels on the virtualized x86 platform.

With a study on existing cache channel techniques, we uncovered their application insufficiency and limitations in a virtualized environment. We then addressed these obstacles by designing a pure timing-based data transmission scheme and discovering the bus locking mechanism as a powerful covert channel medium. Leveraging the memory bus covert channel, we further designed a robust data transmission protocol. To demonstrate the real-world exploitability of our proposed covert channels, we launched attacks on our testbed system and in the Amazon EC2 cloud. Our experimental results show that, contrary to previous research and common beliefs, covert channel attacks in a virtualized environment can achieve high bandwidth and reliable transmission. Therefore, covert channels pose formidable threats to information security in the cloud, and they must be carefully analyzed and mitigated.

For the future work, we plan to explore various mitigation techniques we have proposed. Especially, we view the countermeasures at the cloud provider side a highly promising field of research. Not only do cloud providers have control of rich resources, they also have strong incentive to invest in covert channel mitigation because ensuring covert channel security gives them a clear edge over their competitors.

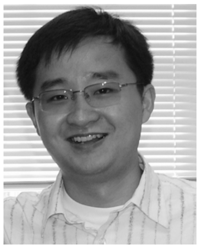
## References

[1] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Proc. 22nd ACSAC*, 2006, pp. 473–482.

[2] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proc. 16th ACM CCS*, 2009, pp. 199–212.

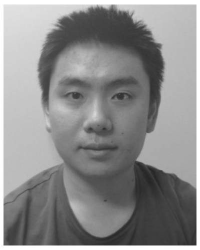
[3] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "HomeAlone: Co-residency detection in the cloud via side-channel analysis," in *Proc. IEEE S&P*, 2011, pp. 313–328.

[4] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of L2 cache covert channels in virtualized environments," in *Proc. 3rd ACM CCSW*, 2011, pp. 29–40.
[5] D. G. Murray, S. H. , and M. A. Fetterman, "Satori: Enlightened page sharing," in *Proc. USENIX ATC*, 2009, pp. 1–14.
[6] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Software side channel attack on memory deduplication," presented at the , 23rd ACM SOSP, 2011, Poster Session.
[7] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. 7th Conf.USENIX Security Symp.*, 1998, pp. 63–78.
[8] J. Wei, X. Zhang, G. Ammons, V. Bala, and P. Ning, "Managing security of virtual machine images in a cloud environment," in *Proc. ACM CCSW*, 2009, pp. 91–96.
[9] S. Bugiel, S. Nürnberger, T. Pöppelmann, A.-R. Sadeghi, and T. Schneider, "AmazonIA: When elasticity snaps back," in *Proc. 18th ACM CCS*, 2011, pp. 389–400.
[10] I. S. Winkler and B. Dealy, "Information security technology?...don't rely on it: A case study in social engineering," in *Proc. 5th Conf. USENIX UNIX Security Symp.*, 1995, pp. 1–5.
[11] G. L. Orgill, G. W. Romney, M. G. Bailey, and P. M. Orgill, "The urgency for effective user privacy-education to counter social engineering attacks on secure computer systems," in *Proc. 5th CITC5*, 2004, pp. 177–181.
[12] B. W. Lampson, "A note on the confinement problem," *Commun. ACM*, vol. 16, pp. 613–615, 1973.
[13] C. H. Rowland, "Covert channels in the TCP/IP protocol suite," *First Monday*, vol. 2, no. 5, 1997, DOI: 10.5210/fm.v2i5.528.
[14] S. Cabuk, C. E. Brodley, and C. Shields, "IP covert timing channels: Design and detection," in *Proc. 11th ACM CCS*, 2004, pp. 178–187.
[15] T. V. Vleck, "Timing channels," presented at the IEEE TCSP, 1990, Poster Session.
[16] C. Percival, "Cache missing for fun and profit," in *Proc. BSDCan*, 2005, pp. 1–13.
[17] U.S. Department of Defense, Washington, DC, USA, "TCSEC: Trusted computer system evaluation criteria," Tech. Rep. 5200.28-STD, 1985.
[18] G. Shah and M. Blaze, "Covert channels through external interference," in *Proc. 3rd USENIX WOOT*, 2009, pp. 1–7.
[19] G. Shah, A. Molina, and M. Blaze, "Keyboards and covert channels," in *Proc. 15th USENIX Security Symp.*, 2006, pp. 59–75.
[20] F. G. G. Meade, "A guide to understanding covert channel analysis of trusted systems," U.S. National Computer Security Center, Washington, DC, USA, Manual NCSC-TG-030, 1993.
[21] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Proc. ACM CCS*, 2012, pp. 305–316.
[22] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou, "Hardware-software integrated approaches to defend against software cache-based side channel attacks," in *Proc. 15th IEEE HPCA*, 2009, pp. 393–404.
[23] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *Proc. 41st Annu. IEEE/ACM MICRO*, 2008, pp. 83–93.
[24] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud," in *Proc. 21st USENIX Security*, 2012, p. 11.
[25] Amazon Web Services, Seattle, WA, USA, "Amazon EC2 dedicated instances," [Online]. Available: http://aws.amazon.com/dedicated-instances/
[26] W. Hu, "Reducing timing charmers with fuzzy time," in *Proc. IEEE S&P*, 1991, pp. 8–20.
[27] W.-M. Hu, "Lattice scheduling and covert channels," in *Proc. IEEE S&P*, 1992, pp. 52–61.
[28] J. W. Gray, III, "On introducing noise into the bus-contention channel," in *Proc. IEEE S&P*, 1993, pp. 90–98.
[29] J. W. Gray, III, "Countermeasures and tradeoffs for a class of covert timing channels," Hong Kong University of Science and Technology, Hong Kong, Tech. Rep., 1994.
[30] Intel, Santa Clara, CA, USA, "The Intel 64 and IA-32 Architectures Software Developer's Manual," 2014 [Online]. Available: http://www.intel.com/products/processor/manuals/
[31] J. Winkler and J. Munn, "Standards and architecture for token-ring local area networks," in *Proc. ACM Fall Joint Comput. Conf. ACM*, 1986, pp. 479–488.
[32] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, 1960.
[33] P. J. Mucci, K. London, and P. J. Mucci, "The CacheBench Report," Nichols Research Corporation, Tech. Rep., 1998.
[34] Amazon Web Services, Seattle, WA, USA, "Micro instances," 2013 [Online]. Available: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts_micro_instances.html



**Zhenyu Wu** (M'12) received the Ph.D. degree in computer science from the College of William and Mary, Williamsburg, VA, USA, in 2012.

He is a Research Staff Member with NEC Laboratories America, Inc., Princeton, NJ, USA. His research focuses on enterprise system security and mobile application security. His research interests also lie in general system and network security, including but not limited to malware analysis, packet filters, and Internet chat, and online game security.



**Zhang Xu** received the B.S. degree in computer science from Beihang University, Beijing, China, in 2010, and the Master's degree in computer science from the College of William and Mary, Williamsburg, VA, USA, in 2012, and is currently pursuing the Ph.D. degree in computer science at the College of William and Mary.

His research interests include cloud computing, system security, and power management of data centers.

**Haining Wang** (S'97–M'03–SM'09) received the Ph.D. degree in computer science and engineering from the University of Michigan, Ann Arbor, MI, USA, in 2003.

He is an Associate Professor of computer science with the College of William and Mary, Williamsburg, VA, USA. His research interests lie in the areas of security, networking systems, and cloud computing.