BusMonitor: A Hypervisor-Based Solution for Memory Bus Covert Channels

Brendan Saltaformaggio, Dongyan Xu, Xiangyu Zhang
Department of Computer Science, Purdue University
West Lafayette, Indiana 47907, USA
{bsaltafo, dxu, xyzhang}@cs.purdue.edu

ABSTRACT

Researchers continue to find side channels present in cloud infrastructure which threaten virtual machine (VM) isolation. Specifically, the memory bus on virtualized x86 systems has been targeted as one such channel. Due to its connection to multiple processors, ease of control, and importance to system stability the memory bus could be one of the most powerful cross-VM side channels present in a cloud environment. To ensure that this critical component cannot be misused by an attacker, we have developed BusMonitor, a hypervisor-based protection which prevents a malicious tenant from abusing the memory bus's operation. In this paper we investigate the dangers of previously known and possible future memory bus based side channel attacks. We then show that BusMonitor is able to fully prevent these attacks with negligible impact to the performance of guest applications.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Design, Experimentation, Security

Keywords

Covert channel, security, virtualization

1. INTRODUCTION

Recent research has show that on virtualized x86 platforms many cross-VM side channels exist and can be readily exploited by malicious tenants. The x86 architecture's processor to main memory communication bus, known as the memory bus, has become the medium of one such side channel. In this paper, we analyze the dangers posed by allowing tenant VMs to control the memory bus's operation. We study the performance and security implications of a malicious tenant's abuse of the memory bus, and during our experimentation we find that these attacks are severe enough to justify the development of a defensive countermeasure.

To this end, we have designed BusMonitor, a hypervisor-based protection which prohibits a malicious tenant from using the memory bus as a cross-VM side channel. Based on the knowledge gained from our attack investigation, BusMonitor is designed to effectively partition the memory bus's operation between VMs. This prevents any misuse of the memory bus to form a cross-VM side channel. By mitigating this powerful side channel, BusMonitor also protects against covert channels which use the memory bus for communication. Further, we show that BusMonitor is customizable and introduces negligible overhead to real world applications.

The remainder of this paper is organized as follows. Section 2 discusses the motivation behind this work and surveys related work. We then analyze the dangers of the memory bus side channel attacks in Section 3. Section 4 describes the design of BusMonitor and how it achieves the desired protection. In Section 5 we evaluate the correctness of BusMonitor's design, how effectively it protects against our previous attacks, and any overhead that BusMonitor induces. Finally, in Section 6 we discuss future work and conclude.

2. RELATED WORK AND MOTIVATION

This work was originally motivated by the discovery made by Wu et al. [14] that the memory bus on virtualized x86 systems is an ideal candidate for transmitting data between VMs covertly. Previous research had shown that covert channels *could* exist in a cloud environment [8] [15] [16], but the covert channel in [14] was the first to achieve bit-rates above the threshold suggested by the Trusted Computer System Evaluation Criteria [1]. We believe that this discovery (and resulting covert channel) is both extremely dangerous to cloud computing security but also easily solvable.

Previous work on cross-VM side channels has heavily targeted the processor's cache. Zhang et al. [17] were able to extract portions of cryptographic keys from a coresident VM by measuring the processor's L1 cache. Ristenpart et al. [8] developed a cache-based side channel which could detect coarse-grained information about a coresident VM. Later, Zhang et al. [16] used the processor cache to detect coresidency in a hosted cloud. More relevant to this work, Xu et al. [15] demonstrated that the L2 cache could be used to establish a covert channel between VMs.

To defend against these cache-based channels, Page [7] and Kong et al. [4] both proposed cache partitioning techniques to disallow VM cache line sharing. This is most relevant to our work because it attempts to deny VMs access to the "physical layer" of the communication channel. Wang and Lee proposed two methods of cache randomization to prevent cache-based side hannels [11] [12]. NoHype [3] tries to prevent side channel attacks by assigning only one VM per CPU core. However, since the memory bus is shared among all CPUs in a system, this cannot protect against the attacks shown in this paper. The sHype architecture [9] supports assigning only certain VMs to an entire hypervisor. This would prevent hardware based side channels, but a cloud provider cannot know a priori if one VM poses a threat to coresident customers.

We consider these defenses complimentary to the research presented here. Since side channels are often highly hardware and architecture specific, no single defensive measure will secure a cloud environment from all or most forms of side channel attack. To the best of our knowledge, this paper represents the first attempt to defend against memory bus based cross-VM attacks.

3. DANGERS OF MEMORY BUS ABUSE

The memory bus on the x86 platform is the main communication channel between the processors and main memory. Because this connection is shared among all processors, contention over its use can negatively affect the performance of an entire system. To minimize this bottleneck, modern CPUs rely heavily on local data caches in an attempt to handle data locally as often as possible. In most cases, this drastically improves the performance of the system because the access times for data fetches from a local cache are significantly faster than from main memory.

Besides the initial fetch to fill a cache line, there is only one reason that a processor would intentionally perform operations against main memory: an *atomic instruction*. Atomic instructions are assembler directives which instruct a processor to perform an operation with "exclusive use of any shared memory" [2]. These instructions are most often used to implement semaphores or other parallel computing primitives (e.g. bit test and set). Further, these operations must be performed against main memory - ignoring the processor's local caches and temporarily locking access to the memory bus.[1]

Since memory bus contention is pivotal to overall system performance, one may assume that any instruction which monopolizes its use must be a privileged instruction. Unfortunately, this is not the case. On all modern x86 systems atomic instructions are executable from any privilege level, thus allowing an attacker with minimal privileges to induce system-wide memory access latency. This small oversight in privilege requirement has now led to a very real security risk. In the remainder of this section we investigate how dangerous an attacker's malicious misuse of atomic instructions can be.

3.1 The Covert Agent

Wu et al. originally discovered in [14] that memory access latency caused by locking the memory bus could be leveraged to form a high bit-rate covert channel. The sender

[1]The implementation of the memory bus locking feature is architecture specific and has changed drastically over various versions of Intel processors. Wu et al. performed a detailed analysis of these changes and documented the important differences in [14].

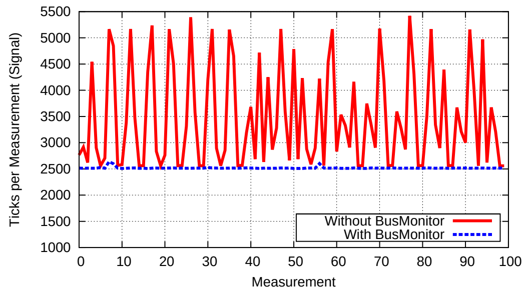


Figure 1: Covert Channel Receiver Measurements.

transmits bits by locking the memory bus and causing latency to assert a "high" signal or allowing latency to stay low. The receiver then measures the memory access latency to detect the signal. The authors also chose to implement three key link layer features which influence the bit-rate and reliability of their covert channel: receiving confirmation, clock synchronization, and error correction. In order to fully understand the severity and practicality of such a covert channel, we had to recreate it.

We chose to simplify the channel design by only implementing clock synchronization. To help lower error rates we encode each data packet as opposed to the entire bit string. This localizes the effect of bit errors (i.e. channel noise causing an erroneous change in signal assertion) to only a single packet.

Our channel transmits an input bit string S as follows. S is partitioned into packets and each is encoded using Differential Manchester coding [13]. To transmit the signal over the physical layer the sender performs atomic instructions repeatedly for a set period of time t. The receiver then measures the signal being asserted (i.e. delay when accessing main memory) by counting the number of atomic instructions (α) that can be performed in t time. If α is below the observed threshold then the receiver can assume that the channel's signal is being asserted high. Figure 1 shows the memory access delay (signal) observed by the receiver during one of our experiments. The link layer on the receiver's side then decodes the signal and reassembles the packets. More detail on the construction and operation of the covert channel can be found in [14].

3.2 Channel Performance

Using this channel design, we were able to recreate, verify, and partially surpass the results reported in [14]. Our experimental setup consists of two Ubuntu 12.04 VMs running on an Ubuntu 12.04 KVM hypervisor [5]. One of these VMs serves as the attacker controlled VM which listens on the covert channel for data transmission. The other VM serves as the victim VM which the attacker infiltrates and executes the transmitting side of the channel. Like the covert channels presented in [14], [15], and others, our attacker only needs to execute code as an unprivileged user in either VM to successfully perform the data exfiltration.

We find that when choosing a packet size that keeps the error rate in an acceptable range, our channel is nearly 1.5 times as fast as the channel in [14]. This is mainly due to our choice not to implement error correction, which requires sacrificing up to 50% of channel bandwidth to parity bits. In turn, we experience higher error rates than the channel

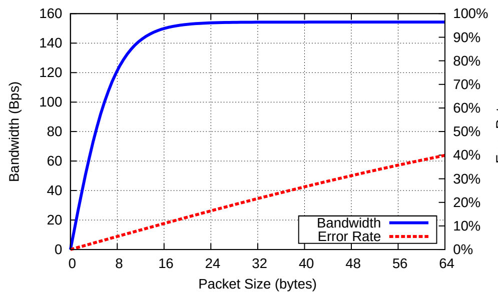


Figure 2: Covert Channel Experimentation Results.

presented in [14], specifically when packet size is chosen to be very large.

Among all configurations we tested, we consider the optimal packet size to be 15 bytes which yields a bandwidth of 146 bytes per second with only an 11% error rate, but this parameter should be configured depending on the use case. With this setup, our channel has a bandwidth 56% higher than the optimal case (i.e. in-house noiseless) covert channel experiments in [14]. Our error rates are higher, but not unacceptable for most applications. Further, we could still choose to apply Forward Error Correction coding to our channel and due to our higher bandwidth, maintain a higher data throughput than achieved in [14]. The average metrics that were observed over 100 experimental runs are shown in Figure 2.

3.3 A Performance Degradation Attack

Besides the covert channel, we found that the memory bus can be misused to perform an additional cross-VM attack: Performance degradation. As we have seen, the covert channel induces a noticeable slow down in memory access time in coresident VMs. Intuitively, an attacker could also induce this slow down to drastically impair the memory access times of coresident VMs and degrade the performance of all VMs sharing the physical server.

We again take on the role of an attacker to show the extreme danger posed by allowing an unprivileged guest VM to abuse the memory bus's operation. To perform this attack, we modified the covert channel design to lock the memory bus as frequently as possible. Again this requires no special permissions, only unprivileged code execution in an attacker owned VM. By constantly locking the memory bus, the attacker can cause all coresident VMs to experience prolonged delays when accessing physical memory.

To measure how locking the memory bus could degrade system-wide performance we conducted memory bandwidth benchmarking on our experimental setup from Section 3.2. In these experiments one VM serves as the attacker controlled VM which runs the memory bus locking attack code. In the victim VM we run the memory bandwidth benchmarking suite `bandwidth` [10]. `bandwidth` is configured to perform random 128-bit memory reads to fill a buffer of varying sizes. Figure 3a shows the resulting memory bandwidth seen by the victim VM with and without an attacker present. From this we observe that the attack degrades memory bandwidth by over 70%.

3.4 The More You Move, The Faster You Sink

The degree to which a coresident VM feels the effects of this attack depends on the workload of the victim VM. Since modern processors strive to perform memory operations against a local cache, the victim may only use the memory bus occasionally to fill the cache. However if the victim is frequently performing operations which invalidate, overwrite, or bypass the processor's cache - such as the random memory reads performed by `bandwidth` - then poor memory bus performance will lead to poor overall system performance.

We analyze the practicality of this attack and its effects by repurposing the same experimental setup from before. For these experiments we use the same attacker controlled VM and install a web-server on the victim VM. In experiments one and two the victim serves requests for a 750 KB file (e.g. a static web-page) and a 3 MB PDF file for download respectively. In our third experiment, the victim web-server serves requests for a website that performs 1000 local MySQL queries to generate an 800 KB webpage file. This third experiment models a webpage with dynamic content such as a blog or social media.

Table 1 shows the average results over 200 iterations of these three experiments. Most notably in experiment three the attacker is able to degrade the victim web-server's response time by up to 21%. Intuitively, the more work the web-server has to perform per request the more detrimental the memory bus contention is. Specifically in experiment three the web-server employs multiple programs to service a single request which requires repeatedly filling the processor's local cache from physical memory. Not surprisingly, experiment three shows the greatest vulnerability to this attack. Further, it is not hard to imagine a targeted attack situation where a 20% decrease in server responsiveness may be very costly for a company or organization.

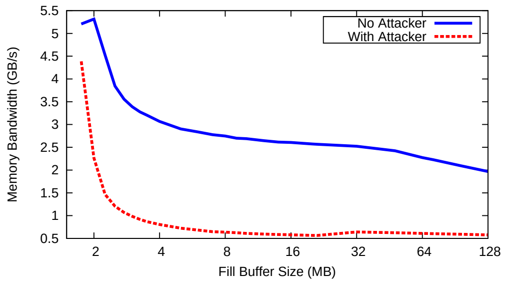
This result is more concerning because this experiment is favorable for the *victim*. As mentioned previously, one of the main reasons a processor will be forced to use the memory bus is to fill its local cache. In our experiment we ran only two VMs but in a production environment there would be many more VMs sharing a physical server. More VMs being scheduled on a server means more opportunity for the cache to be invalidated - causing many more fetches from main memory over the memory bus. This would only cause more contention and delays for VMs attempting to access main memory and being blocked by our attacker.

4. BUSMONITOR

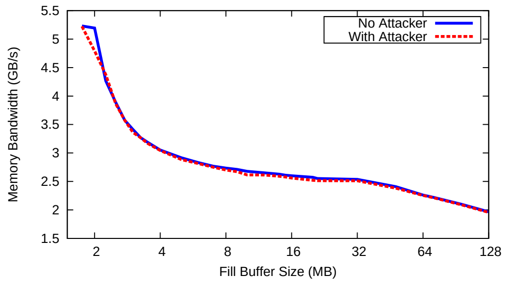
As our investigation in Section 3 shows, in virtualized x86 systems the misuse of atomic instructions has introduced an extreme security risk. We feel that fault lies in processor design and the only complete solution is to require elevated privilege to execute atomic instructions. Indeed it is not uncommon for operating systems to export semaphore functionality via system calls, and since this is the primary use of atomic instructions most applications should be unaffected by the change. In the case of virtualization, we envision a trap to the hypervisor being implemented to prevent (possibly malicious) guest VMs from locking the memory bus.

4.1 Design

BusMonitor is a hypervisor based module which both prevents a malicious tenant from abusing the memory bus's operation and protects the memory bandwidth of the system as a whole. It is necessary for the entire BusMonitor mod-



(a) Without BusMonitor.



(b) With BusMonitor.

Figure 3: Memory Bandwidth Degradation.

ule to reside in the hypervisor to ensure that it cannot be tampered with by a malicious guest VM. In this section we describe the design of BusMonitor and how it achieves the desired memory bus protection described above.

BusMonitor is built on the notion that an atomic instruction in one guest VM should not lock all guest VMs out of memory. The Intel SDM states that an atomic instruction has "exclusive use of any shared memory" [2], and modern processors enforce this by locking the system-wide memory bus. However in a virtualized system the entire main memory bank is not shared. Like processes in an operating system, the hypervisor ensures that each guest VM is given an isolated partition of the memory space. In the presence of memory deduplication, an atomic instrucion modifies writeable data pages which will not be marked for cross-VM sharing. This means that the memory address being manipulated by an atomic instruction is only valid and reachable by a single VM. Therefore only the processors/cores belonging to that VM must be locked from memory when an atomic instruction is executed. BusMonitor implements this relaxation of the hardware specification from within the hypervisor.

BusMonitor is designed to intercept atomic instructions performed within the guest VMs. To do this we first locate and replace each atomic instruction with a trap to the hypervisor. This is done with the help of KVM's shadow paging. With shadow paging, the guest cannot directly add page table entries. Guest updates are made to fake guest page tables then propagated to KVM's shadow page tables during page faults. The processor uses the shadow page tables during guest execution. Since the hypervisor must be invoked to add or change page table entries, we use this opportunity to scan pages for atomic instructions. These instructions are then replaced with traps to the hypervisor.

When the hypervisor intercepts this trap, it pauses the guest's virtual CPU cores (vCPUs) and executes the original instruction without asserting the memory bus lock. Since the guest's vCPUs are paused, no other memory operations can be performed by this guest while the hypervisor is emulating the atomic instruction. This preserves the expected semantics of the original atomic instruction. This also provides that the memory bus is not locked by the atomic instruction and thus cannot be used as a side channel.

4.2 Implementation

BusMonitor consists of two components: A page fault hook and a trap handler. We chose KVM [5] as our hypervisor platform, but BusMonitor's design is generic enough to be easily implemented as a module for any existing hypervisor that supports shadow paging. The components require no modification to the underlying hypervisor code-base other than the insertion of function calls to invoke BusMonitor's execution.

The page fault hook component is responsible for removing atomic instructions from guest code. In virtualized x86 systems, the hypervisor intercepts all page faults generated by the guest VMs. The use of shadow page tables ensures that a page fault is generated whenever guest page tables are changed. We leverage this ability to "prescreen" pages from the hypervisor in our page fault hook component. On each page fault that the guest generates, our hook code assesses the cause of the page fault using a configurable list of policies.[2] If the page fault meets the current set of policies then BusMonitor disassembles the page's content and replaces every atomic instruction with a trap to the hypervisor. The replaced atomic instructions are then registered with the trap handler component and our hook returns execution to the hypervisor.

The guest then executes normally until it encounters one of our inserted traps. This allows the hypervisor to intercept execution and invoke our trap handler component. The trap handler first checks if the guest is trying to execute a previously registered atomic instruction. If so, the trap handler acquires a guest specific semaphore, pauses any running vCPUs assigned to that guest, and executes the instruction without asserting the memory bus lock. When the instruction completes, the vCPUs are restarted, semaphore released, and execution is returned to the guest VM.

5. EVALUATION

The evaluation of BusMonitor in this section is threefold. First, we are interested in the correctness of our implementation. This is because we are emulating a very specific function which is normally performed entirely by hardware

[2]For our experiments these policies include "Page fault caused by an instruction fetch" and "Address is not in kernel space". The policy "Page fault caused by an instruction fetch" restricts BusMonitor to only operate on guest code pages. Additionally, we ignore atomic instructions in the guest's kernel because the system calls which export semaphore and lock functionality are composed of many instructions. Thus an attacker repeatedly calling these system calls could not lock the memory bus rapidly enough to perform any of the attacks from Section 3.

Table 1: Web-server Performance Degradation Experiment

	No Attacker, No BusMonitor		With Attacker, No BusMonitor			Attacker With BusMonitor		
Serves	Transfer Rate	Avg. Time	Transfer Rate	Avg. Time	Degradation	Transfer Rate	Avg. Time	Degradation
750KB Static Webpage	50 MB/s	15 ms	46 MB/s	17 ms	8%	50 MB/s	15 ms	0%
3MB PDF Download	72 MB/s	44 ms	63 MB/s	50 ms	14.2%	72 MB/s	45 ms	2%
800KB Dynamic Webpage	50 MB/s	850 ms	45 MB/s	1031 ms	21%	50 MB/s	854 ms	<1%

- processor instructions are assumed to be correct and so our implementation must meet that standard. Second, we must show that our design completely protects against the attacks demonstrated in Section 3. Finally, we will show that any performance penalty introduced by BusMonitor is negligible and only experienced in very rare situations in the real world.

5.1 Semantic Correctness

Before evaluating the usual benchmarks (protection effectiveness, performance overhead, etc.), we first need to show that BusMonitor correctly emulates the atomicity of the replaced atomic instructions. If BusMonitor did not correctly implement exclusive access to the guest VM's memory then it would introduce a very non-obvious race condition: The execution of two vCPUs interleaving *within* BusMonitor's emulation code. To ensure that this is not the case, we need to evaluate the correctness of our atomic instruction emulation.

BusMonitor allocates a semaphore for each guest VM. When a guest atomic instruction is intercepted, this semaphore is acquired before pausing the guest's other vCPUs and emulating the atomic instruction. This ensures that only one vCPU of that VM can be executing BusMonitor's emulation code at a time. Further, since the guest vCPUs are paused after acquiring the guest's semaphore this ensures that the emulated atomic instruction is given exclusive access to the guest's memory.

To perform a more empirical evaluation, we designed a race condition inducing application and ran it inside a guest VM. This application used a locked increment instruction to update a shared semaphore (written entirely in assembler directives) between as many vCPUs as available. If BusMonitor's atomic instruction emulation was faulty then this code would suffer from the typical read-update-rewrite race condition. Over millions of iterations of the application locking and unlocking the semaphore, never were two vCPUs inside the protected code section at the same time. Further, we used hardware debug registers to guarantee that BusMonitor's traps were executed concurrently from multiple vCPUs. In this case as well, BusMonitor paused the guest's vCPUs, emulated the atomic instruction, and resumed the guest sequentially for every trap it handled. The result is that only one atomic instruction per guest VM can update memory at a time.

5.2 Protection

With the correctness of our emulation demonstrated, we returned to our previous memory bandwidth test to determine the effectiveness of BusMonitor's protection. The key security advantage that BusMonitor provides is that when one guest VM executes an atomic instruction no measurable effects can be seen in coresident VMs. With BusMonitor's protection now in place, this attack should be able to affect the system's memory bandwidth.

We reran the evaluation presented in Section 3.3, using `bandwidth` to measure the victim VM's memory bandwidth. The only difference between this test and our previous runs of this experiment is that now BusMonitor is intercepting all atomic instructions. As Figure 3b shows, the attack had no effect on the (no longer) victim VM's memory bandwidth. This is because the victim VM's memory transactions are not suspended due to a locked memory bus. The attacker VM continues to attempt to execute atomic instructions but now BusMonitor is intercepting and emulating them without asserting the memory bus lock.

Not surprisingly, BusMonitor also prevented the covert channel from transmitting any data. Despite no modification, the sending side of the covert channel cannot cause a measurable difference in the receiver's memory access time. As Figure 1 shows, with BusMonitor enabled the receiver is no longer able to detect a signal. While BusMonitor does perform the expected atomic operation correctly, it renders the covert channel useless because no signal can be asserted or detected.

5.3 Performance Overhead

Finally, we assert that the performance overhead induced by BusMonitor is negligible for real world applications. This is because the amount of overhead that BusMonitor introduces is specific to each guest application and not the system as a whole. Thus one application being heavily instrumented by BusMonitor causes no additional overhead for coresident VMs. This is shown in our experiment with `bandwidth` from Section 5.2. The attacker's memory bus locking process is being constantly intercepted by BusMonitor's replaced atomic instruction traps. However, this causes no disturbance to the victim VM as it tests the memory bus bandwidth. A guest VM will experience overhead *proportional to the number of atomic instructions that only it performs.*

Overhead caused by BusMonitor's trap handler is entirely application dependent. If an application does not use atomic instructions then the trap handler will never be invoked. Naturally, compared to executing only a single (dangerous) instruction, BusMonitor's trap handler does incur a significant performance penalty. However, in real-world applications we find that the use of atomic instructions is quite infrequent. We performed a scripted disassembly and search of our development workstation's `/bin` directory and found only 106 out of 1916 applications containing any occurrences of atomic instructions. Again, this is because atomic instructions are primarily used only to implement semaphores and shared variables.

The overhead from BusMonitor's page fault hook component is both configurable and application specific. Its execution is regulated via the configurable policy list to only run when necessary. When a page does need to be "screened" the little overhead caused by the page fault hook is amortized over the lifetime of the application.

To illustrate the amortization of BusMonitor's overhead on a real world application (and further demonstrate its effective protection) we revisit the victim web-server test case from Section 3.4. As Table 1 shows, the attacker is now unable to degrade the victim web-server's response time. Like before, this is because the attacker's atomic instructions are unable to lock the memory bus with BusMonitor installed. Also, we see that BusMonitor causes very little overhead to any of the three test setups. For test case three, the victim web-server shows less than a 1% slowdown. This is due to the fact that test three requires much more time to process each request. Thus the one-time overhead of BusMonitor's page fault hook component is much less noticeable.

6. CONCLUSION

As we have shown, the memory bus can be misused as a dangerous cross-VM side channel. The covert channel presented in Section 3 is capable of transmitting data "under the radar" of any current protection measures. Further we have shown that memory bus misuse can significantly degrade the performance of coresident VMs. Together these attacks justify the need for BusMonitor. The experiments in Section 5 proved that BusMonitor is able to prevent the attacks which exploited the memory bus side channel. Additionally, it was shown that BusMonitor introduced negligible overhead to the guest VMs.

However, BusMonitor is not a complete solution. As mentioned previously, the memory-bus based side channel is a result of an oversight in processor design. We believe that a more complete solution would be to partition the memory bus's operation via hardware by improving the architecture's design. Such a hardware based solution would introduce less overhead than BusMonitor and provide a finer-grained protection which is not possible in a purely software-based solution.

The authors admit that a known limitation of BusMonitor is the identification of atomic instructions in memory. Currently, BusMonitor does not handle obfuscated or self-modifying binaries. We leave the addition of a more complex conservative disassembler [6] as future work on this project.

As cross-VM side channels gain more attention from the research community, it will become imperative that cloud vendors adopt defensive measures to protect clients. The attacks in this paper are both practical and relatively simple to implement, which leads to the concern that unknown side channels may still exist. In the future, we hope that hypervisor and processor designs will incorporate protections against such attacks.

Acknowledgment

We thank the anonymous reviewers for their insightful comments. This research was supported, in part, by the US NSF under award 0855141 and by DARPA under Contract 12011593. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of NSF or DARPA.

7. REFERENCES

[1] Department of Defense. TCSEC: Trusted computer system evaluation criteria. Technical Report 5200.28-STD, 1985.

[2] Intel Corporation. Intel architecture software developer manual. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`, 2012.

[3] E. Keller, J. Szefer, J. Rexford, and R. Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proc. 37th ACM International Symposium on Computer Architecture*, pages 350–361, 2010.

[4] J. Kong, O. Aciicmez, J. Seifert, and H. Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *Proc. IEEE 15th International Symposium on High Performance Computer Architecture*, pages 393–404, 2009.

[5] KVM. `http://www.linux-kvm.org`, 2012.

[6] S. Nanda, W. Li, L.-C. Lam, and T.-c. Chiueh. Bird: Binary interpretation using runtime disassembly. In *Proc. 2006 Symposium on Code Generation and Optimization*, 2006.

[7] D. Page. Partitioned cache architecture as a side-channel defence mechanism. *IACR Cryptology ePrint Archive Report*, 280, 2005.

[8] 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 Conference on Computer and Communications Security*, pages 199–212, 2009.

[9] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn. Building a mac-based security architecture for the xen open-source hypervisor. In *21st Annual Computer Security Applications Conference*, 2005.

[10] Z. Smith. Bandwidth: a memory bandwidth benchmark. `http://zsmith.co/bandwidth.html`.

[11] Z. Wang and R. Lee. Covert and side channels due to processor architecture. In *Proc. 22nd Annual Computer Security Applications Conference*, pages 473–482, 2006.

[12] Z. Wang and R. Lee. A novel cache architecture with enhanced performance and security. In *Proc. 41st IEEE/ACM International Symposium on Microarchitecture*, pages 83–93, 2008.

[13] J. Winkler and J. Munn. Standards and architecture for token-ring local area networks. In *Proc. 1986 ACM Fall Joint Computer Conference*, pages 479–488, 1986.

[14] Z. Wu, Z. Xu, and H. Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Proc. 21st USENIX Security Symposium*, 2012.

[15] 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 Workshop on Cloud Computing Security*, pages 29–40, 2011.

[16] Y. Zhang, A. Juels, A. Oprea, and M. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proc. 2011 IEEE Symposium on Security and Privacy*, pages 313–328, 2011.

[17] Y. Zhang, A. Juels, M. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proc. 2012 ACM Conference on Computer and Communications Security*, pages 305–316, 2012.