Understanding Contention-Based Channels and Using Them for Defense

Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, Mohit Tiwari
The University of Texas at Austin
{casen.h, mikhail.kazdagli, ankitsr}@utexas.edu, {dimakis, sriram, tiwari}@austin.utexas.edu

Abstract—**Microarchitectural resources such as caches and predictors can be used to leak information across security domains. Significant prior work has demonstrated attacks and defenses for specific types of such microarchitectural side and covert channels. In this paper, we introduce a general mathematical study of microarchitectural channels using information theory. Our conceptual contribution is a simple mathematical abstraction that captures the common characteristics of all microarchitectural channels. We call this the Bucket model and it reveals that microarchitectural channels are fundamentally different from side and covert channels in networking.**

We then quantify the communication capacity of several microarchitectural covert channels (including channels that rely on performance counters, AES hardware and memory buses) and measure bandwidths across both KVM based heavy-weight virtualization and light-weight operating-system level isolation. We demonstrate channel capacities that are orders of magnitude higher compared to what was previously considered possible.

Finally, we introduce a novel way of detecting intelligent adversaries that try to hide while running covert channel eavesdropping attacks. Our method generalizes a prior detection scheme (that modeled static adversaries) by introducing noise that hides the detection process from an intelligent eavesdropper.

1. Introduction

Virtual machine isolation is a fundamental security primitive. However, several recent works [1], [2], [3], [4], [5], [6] have shown that shared hardware can be exploited to leak information from one process to another, negating all isolation guarantees offered by operating systems and hypervisors. For example, a '1' in a secret key induces a square-and-multiply operation while a '0' induces only a squaring operation. Using this information, an attacker that shares CPU usage with a victim process can measure its own instruction cache hit rate and use it to determine the secret key [5].

Such information leaks have been used to infer secret keys from cryptographic libraries in commercial clouds like Amazon EC2 [1], [5]. When these covert channels are used deliberately to exfiltrate secret data, rates up to 100 bits per second [2] have been reported on Amazon EC2. These leaks do more damage than just giving up victims' secrets; their threat has led commercial cloud vendors to shut down important features such as hardware counters and simultaneous multi-threading [3] for all customers.

To combat these information leaks, hardware architects have proposed several solutions including partitioning or time-multiplexing caches [7], memory controllers [8], and networks-on-chip [9], [8]; clearing out leftover state in caches [10] and branch predictor on a context switch [11]; and even complete systems-on-chip where processes cannot interfere with each other [12], [13], [14]. Alternatively, architects have also proposed the introduction of random noise to hardware counters [15] or cache replacement policies [7]. While these approaches are promising, they either address only known, specific sources of leaks [7], [8], [9] or require far-reaching changes to the entire microarchitecture [12], [14] that hamper adoption.

A complementary software-only approach to protect sensitive data in the cloud is to rent *dedicated physical machines*, so that only friendly virtual machines co-reside on a physical machine. The challenge for a cloud tenant then is to audit whether an attacker has exploited a vulnerability in the cloud provider's software to co-reside on the same hardware [1]. HomeAlone [3] introduces one such audit technique where a tenant uses cache lines like miners use canaries – the tenant leaves the canary cache lines untouched and if an attacker process evicts a few such lines, the tenant's detector process raises an alarm. Avoiding co-residency thus addresses all microarchitectural attacks at a cost of scaling in terms of physical machines (v. virtual machines), and is used by cloud tenants who work with financial, military, and sensitive personal data.

However, considerable work is required to realize the promise of dedicated machines as a secure, immediately usable approach. First, HomeAlone's detector is specific to caches and an attacker who uses the main memory interface to leak data [2] using cache bypassing instructions will avoid triggering cache-line based alarms. HomeAlone also requires multi-threading and performance counters to be turned off to prevent data leaks. Worse, HomeAlone is vulnerable against an intelligent attacker who is aware of the defense and can adapt to it. For example, HomeAlone has to determine whether canary lines are evicted by an attacker or by the hypervisor. An attacker who a) lowers her hardware activity to match the background noise and b) models HomeAlone's detector to pause the attack while detection is underway can evade HomeAlone's detector.

In this paper, we introduce the formal study of microarchitectural channels using information theory. Our conceptual contribution is a simple mathematical abstraction that captures the common characteristics of all microarchitectural channels. We call this the *Bucket model* and it reveals that microarchitectural covert channels are fundamentally different from previously studied timing channels because reads are destructive, i.e., *reading a bit overwrites the value of the bit*. We start by developing the noise-free case and show that perfect (in the Shannon sense) detection can be obtained against arbitrary co-resident virtual machines. In contrast, provably undetectable covert channels have been shown in networking where attackers encode secret bits into the timing or size of network packets.

We quantify the information theoretic capacity achievable through microarchitectural channels, including new ones through AES hardware and cache bypass instructions. By optimizing the alignment of clocks between sender and receiver processes, we show high capacities of greater than 500 Kbps for both intra-core and inter-core channels. Further, we show that *multiple* channels communicating concurrently can achieve capacities over 200 Kbps. These high capacities, when compared to 100 bps on EC2 today, show that as commercial hypervisors move towards lighter-weight isolation techniques such as Docker [16], microarchitectural channels become tremendously more leaky.

Finally, we extend the Bucket model to account for noise from background processes and apply the model to detecting an intelligent attacker. The intelligent attacker conducts low-bandwidth probes in order to distinguish a detector from the victim application and goes quiet if she sees a detector. In response, our detector *mimics* a victim application's microarchitectural behavior to prevent the attacker from going quiet, and listens to the channel for bits that are overwritten by an attacker during eavesdropping attempts. In the presence of other processes or system noise, however, our detector simulates an optimized amount of random noise to minimize being detected by an intelligent eavesdropper.

Section 3 describes our design and implementation of a diverse set of microarchitectural channels. We generalize these channels and present the Bucket Model with its destructive read feature in Section 4. Further, we quantify the communication capacity of each channel (Section 5) and use these channels for detection games against an intelligent adversary in Section 6. But first, we begin with a summary of the considerable related work that we build upon.

2. Background and Related Work

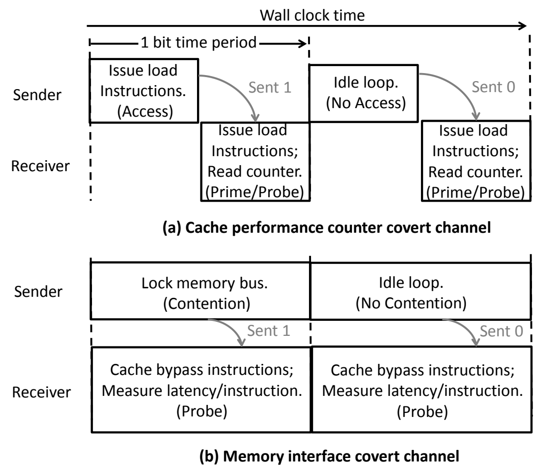
A. Threat Model

We minimally assume two principals in the system: *tenants* and *cloud providers*. Tenants purchase compute and storage resources, typically as virtual machines (VMs) configured equivalent to a real machine, from cloud providers who maintain physical machines in data centers.

We consider tenants to be mutually distrustful: a malicious tenant wishes to either learn other tenants' data (violating confidentiality) or compromise the victim VM's availability by stealing its compute resources. We trust that the cloud providers are neutral – their hypervisor and hardware do not maliciously leak data or compute resources from one tenant VM to another.

Specifically, we address co-residency based attacks through the hardware, where an attacker VM infers information about the victim's VM that is contained in microarchitectural structures like caches and memory controllers. We consider two attack scenarios: a *side-channel* attack, where a victim process such as a cryptographic library *inadvertently* leaks data that is inferred by an attacker process, and a *covert channel attack*, where a malicious process (e.g., a document reader application) that has access to confidential data in a tenant VM *deliberately* leaks the data to a co-resident attacker VM.

In particular, we quantify the covert channel attack surface since they become particularly relevant in systems that include third-party software (as most cloud applications are). Further, covert channel capacities represent an upper bound on side-channel capacities. The Bucket model and detection algorithms, however, target co-resident attackers and hence model/detect both covert and side channel attackers.



Wall clock time
1 bit time period
Sender
Issue load Instructions. (Access)
Sent 1
Idle loop. (No Access)
Sent 0
Receiver
Issue load Instructions; Read counter. (Prime/Probe)
Issue load Instructions; Read counter. (Prime/Probe)
(a) Cache performance counter covert channel
Sender
Lock memory bus. (Contention)
Idle loop. (No Contention)
Sent 1
Sent 0
Receiver
Cache bypass instructions; Measure latency/instruction. (Probe)
Cache bypass instructions; Measure latency/instruction. (Probe)
(b) Memory interface covert channel

Fig. 1: Microarchitecture covert channel examples. Sender-receiver pairs can execute alternately (a) or in parallel (b) – both cases rely on contention for shared hardware.

B. Known microarchitectural channel attacks

Microarchitectural channels can be created by a sender and a receiver VM executing alternately or parallel as shown in Figure 1.

One popular alternating communication technique is termed the Prime-Access-Probe method (Figure 1a). During the Prime stage of communication for (say) a cache channel, the channel receiver sets the cache into a known state by accessing memory locations and bringing its data into the cache. During the Access stage, the channel sender *conditionally* accesses the cache based on a secret bit. To send a 1 (say), the sender accesses memory addresses and evicts the receiver's cache lines. To send a 0, the sender minimizes memory accesses. In the Probe phase, the receiver accesses memory similar to that in the Prime phase and monitors the time it takes to access the entire portion of memory or specific addresses that miss in the cache. The former approach is termed a *timing-driven attack* while the latter is called an *access-driven* attack. After the initial bit is communicated, each subsequent Probe of the channel also serves as a Prime, because both bring the data into the cache. Flush+Reload is another similar technique where a covert channel is created by a receiver first Flushing (instead of Priming) some shared addresses and then Reloading to infer the sender's access trace.

Parallel communication, on the other hand, does not require precise time sharing of the channel between the sender and receiver (Figure 1b). For example, a receiver for the memory interface channel continuously monitors the latency of memory fetching instructions. At the same time, the sender issues its own memory instructions to send a 1, increasing the latency seen by the receiver. To send a 0, the sender decreases contention by idling.

Making such side channels robust in a noisy production cloud setting introduces additional problems. Ristenpart et al. [1] demonstrate how an attacker can reverse engineer Amazon's VM placement algorithms on Amazon EC2 to colocate the attacker VM on the same physical machine as a victim's VM. Once co-located, the attacker can choose from side and covert channel attacks using shared caches (0.2bps to 10bps with varying error rates [17], [18], [19], [10], [1], [5],

[20]) or the shared main-memory interface in symmetric multi-processors as deployed in EC2 (at 100bps [2]). VM migration, frequency scaling, and hypervisor activity add noise and significant synchronization related challenges that the authors address through self-clocking and error correcting codes.

On multi-threaded processors, microarchitectural attacks have been further shown to leak secret keys through branch predictors [11], instruction caches [21], [22], and integer/floating-point units [23]. Such channels have also been *estimated* to have a potential communication bandwidth of over 100 Kbps [24]. To protect against such attacks, multi-threading is not available in production clouds such as Amazon EC2 [3].

We build on this prior work and present the first information theoretic characterization of microarchitectural channels, including known channels, such as caches and branch predictors, and new ones such as memory bandwidth, the new AES-NI hardware units, and performance counters. Performance counters have recently been virtualized (available on both VMWare ESX 5.1 and KVM) and present cross-VM attackers with a new measurement interface that is unaffected even if the VMs' clocks are mutually obfuscated ("fuzzed" [15]). We measure channel capacities using both heavy-weight (KVM) and increasingly popular light-weight (LinuX Containers/Docker) virtualization platforms.

C. Formal model for Covert Channels

Network covert channels have been demonstrated that are provably undetectable from normal "overt" traffic [25]. Such channels are comprised of a sender process exfiltrating secret data through inter-packet timing or packet sizes, for example. A flow is undetectable if the detector cannot distinguish between legitimate and covert flows – i.e, has the same rate of false positives and false negatives for both flows.

Our Bucket Model of microarchitectural channels identifies a crucial difference from network based channels – a receiver *has to* perturb the medium (e.g., caches) in the act of reading a bit, whereas receivers in network channels can read a bit "silently" (without affecting the medium and transmitted bits). We show that a microarchitectural channel thus forces reads to be "destructive", i.e., the receiver overwrites the bit in the act of reading it. Interestingly, this property makes it impossible (in a noise-free setting) to construct an undetectable microarchitectural channel (Section 4).

Other prior work in modeling microarchitectural channel focuses on measuring *relative vulnerability* of two hardware designs to side-channel attacks [26], [27]. Instead of relative leakage, our work exposes the implications of contention based communication and applies it towards detecting attacks. Kopf and Basin [28] quantify side channel capacities by explicitly modeling an attacker's uncertainty about a secret bit after each attack. Their quantitative estimates complement our practical capacity measurements while our model exposes qualitative aspects of contention-driven communication that their model does not cover.

D. Defending against covert channels

Microarchitectural support to minimize interference [8], [9], [12] can eliminate several covert channels. Other architectural measures such as fuzzy time for hardware events [15] or random permutation and partition locked caches [24] also help reduce covert channel leakage. These proposals require substantial modifications to the architecture – we present complementary guest-VM level software techniques to audit microarchitectural usage and detect co-resident adversaries.

In software, StealthMem [29] proposes the use of page coloring to assign a confidential VM's physical memory pages to specific cache lines and then lock the lines in the cache (i.e. they are not evicted by other VMs' accesses). However, locking cache lines has been shown to leak information — the very number of lines that were locked leaks information [30] if an attacker can prime the resource beforehand. Duppel [31] proposes to add noise to memory accesses to lower the bandwidth of the cache channel. Both StealthMem and Duppel are promising but leave non-cache microarchitectural channels unchecked.

The second software-only approach is to reserve a dedicated machine and attempt to ensure that no other malicious VM is co-resident on the machine. HomeAlone [3] proposes that friendly VMs coordinate to avoid specific, randomly chosen L2 cache lines at pre-determined intervals. A foe VM will likely touch the chosen cache line and will be detected (once HomeAlone removes noise from the underlying hypervisor). HomeAlone's approach for dedicated physical instances thus introduces a new design point – achieving security against co-residency attacks for the price of working with dedicated instances.

We generalize HomeAlone's approach in two new directions. First, we show how covert channels other than a shared cache can be used to detect an attacker. Second, we present a technique to detect intelligent attackers who can hide behind background noise (e.g., introduced by a hypervisor) whenever HomeAlone's detector is turned on (Section 6).

3. Creating High Capacity Covert Channels

In this section, we describe our approach to creating efficient microarchitectural covert channels and measure their capacity. Our key insight to achieving high bandwidth is to optimize *synchronization*, i.e., align clocks in the sender and receiver so that they agree on the time period for each bit and when to Access or Probe (see Figure 1). Synchronizing to within a few microseconds enables the sender and receiver to use simple binary signaling without self-clocking codes and yet achieve low bit error rates.

A. Deconstructing Covert Channels

We observe that the root cause of covert channels is *contention* over shared resources between two processes. Wang and Lee [7] identify cache interference as the key to a successful cache side channel attack. We find that this observation extends to every demonstrated side or covert microarchitectural channel attack – including both timing and access driven attacks (the former encodes a bit as contention/no-contention while the latter encodes a bit as contend for address A v. address B) – and attacks through branches, function units, and memory bandwidth. In all cases, a sender accesses a shared microarchitectural structure *predicated on secret data*, and the receiver observes this variation in the degree of contention and infers the secret bit. Such *implicit information flows* through hardware resources create a covert channel.

Note that architectural (ISA) channels where the receiver explicitly reads some processor state left behind by the sender (e.g., floating point registers after a context switch [23] or directly reading a victim's performance counters) are not covered by our analysis. Such *explicit flows* of information require all program-visible hardware state to be correctly virtualized (cleared or saved-and-restored on context switches).

We deconstruct a covert channel into three key primitives that we summarize here before describing each in detail.

1. Communication through contention. A sender and receiver communicate by executing alternately or in parallel. The exact method of contention is specific to each processor resource and (in our prototype) is used to signal a '0' and a '1' through low and high degree of contention. We show high capacities even with simple binary signaling and leave other signaling mechanisms for future work.

2. Offline analysis to determine channel parameters. The sender and receiver have to agree on channel parameters such as frequency (i.e., time slot duration per bit) and within each bit, the duration of an Access and Prime/Probe if the channel is based on Prime-Access-Probe technique. We profile the target machine in order to determine these parameters; e.g., for the load instruction channel, we determine the number of load instructions to be executed per time slot and the range of addresses to be accessed.

Fixing the time slot duration for each bit as opposed to using self-clocking codes [2] is important. This saves the receiver from handling *bit insertions* and *deletions*, i.e. transmission errors that cause phantom bits to be decoded by the receiver or bits to be lost in transmission. Insertion and deletion of bits not only decreases capacity but also make such channels significantly harder to analyze for capacity. In contrast, a fixed time period guarantees that the number of bits that are sent and decoded are equal – the receiver only needs to estimate *bit flip* errors which can then be analyzed to estimate channel capacity [32], [33], [34].

3. Precise Synchronization. The final step is to ensure that the sender and receiver's time slots are *aligned*. If not aligned, both will contend for the channel at overlapping times and introduce bit flip errors. Further, this unconstrained contention leads to a less separable distribution of values for '0' and '1', which in turn forces both the sender and receiver to contend more intensely (i.e. for a longer duration or higher shared resource usage). Unaligned time periods thus reduce channel capacity. On the other hand, aligned time slots enable more precise contention and lower errors even with a low amplitude signal.

We determined channel parameters (Step 2 above) through extensive experiments on our test machine, but achieving precise synchronization (Step 3) and signaling through contention for each channel (Step 1) require further explanation.

B. Synchronizing Sender-Receiver Clocks

The sender and receiver VMs track time using their internal wall-clock timers and our goal is to determine the time-offset between their internal clocks. The first stage of our synchronization protocol is similar to Network Time Protocol, in which the sender and receiver exchange timestamps via explicit communication to align their internal clocks to within one hundred microseconds. In the next stage, the two processes make fine-grained changes to the offset applied to their wall-clock times. Finally, the sender transmits a pilot signal so that the start of communication is clearly detectable by a receiver.

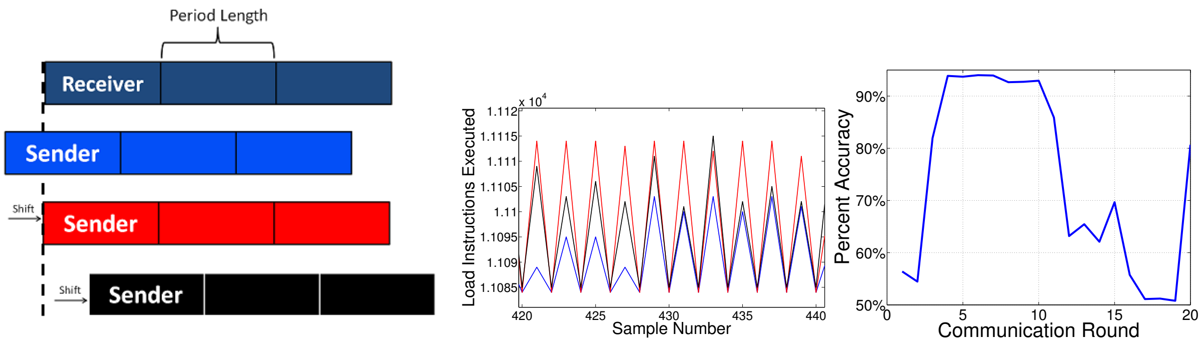
Explicit communication. From our experiments, we find that the wall-clock timers between the sender and receiver VMs can differ by a magnitude of seconds. The receiver determines the offset between their wall-clock times as shown in Figure 3 – the receiver records its own clock value; requests the sender for its wall-clock time; computes the round-trip latency and can thus determine the offset to adjust its own clock value by. Typically, this explicit synchronization phase allows the processes to synchronize their internal clocks within a few hundred microseconds of one another.

One might question why explicit synchronization is allowed when the goal is to establish a covert channel. The answer is that many scenarios allow a covert channel sender and receiver to communicate explicitly. For example, in this paper a sender and receiver are friendly and communicate covertly to detect a malicious co-resident tenant. Hence, they are allowed to first communicate explicitly via a network socket to synchronize their clocks. Another more traditional attack scenario where explicit synchronization can work is in an information flow control (IFC) systems [35], [36]. IFC systems prevent a sender and receiver from communicating only after the sender has accessed secret data. The sender can align itself with a malicious receiver using explicit network messages, access secret data which upgrades its security label to a confined process, and then start leaking secrets covertly. Fundamentally, explicit synchronization primarily *accelerates* the synchronization procedure. We now present our slower, but much more finer-grained, synchronization technique that uses only covert channels.

Fine-grained alignment. Clock alignment within a few hundred microseconds of one another is not close enough to execute reliable communication at high bandwidth rates (over 10 Kbps). Therefore, we execute an additional synchronization protocol to further align the sender-receiver time slots (Figure 2a). We do this by communicating a known bit sequence across the *covert* channel for multiple rounds, as seen in Figure 4. After each round of communication, the sender shifts its clock by one microsecond, slightly changing the time when the two processes will contend for the shared resource. As the clocks become more synchronized, the two processes' contention becomes more aligned and the communication becomes increasingly reliable as seen in Figure 2b.

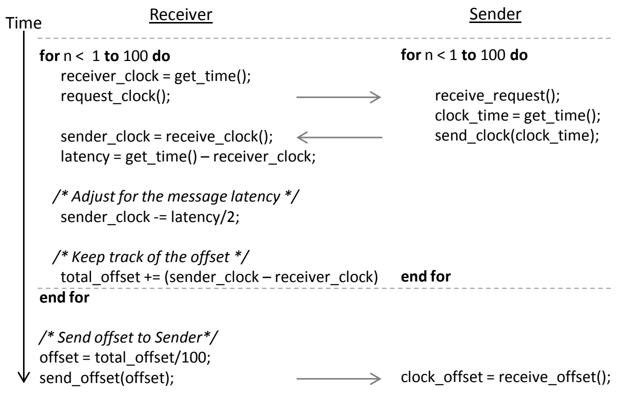
Once the optimal period alignment has been reached, each shift the sender makes degrades the communication across the channel. This spike in communication accuracy can also be seen in Figure 2c. From the best round of communication, the receiver determines the clock offset at 1μs precision. All in all, synchronization takes around 15s in our experiments, depending upon the number of bits to test communication, and only needs to be performed once per VM execution. Figure 5 shows the benefits of precise synchronization compared to a coarse-grained synchronized channel.

Pilot Signal. Once the VMs' clocks are aligned, the receiver only needs to know when the sender is actively



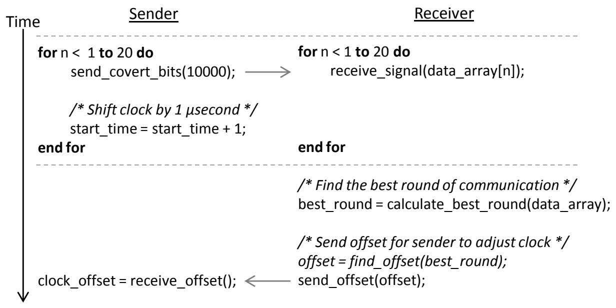
(a) Fine-grained clock adjustments by Sender (b) Signal strength per Sender clock shift (c) Signal accuracy per round of communication

Fig. 2: (a) The sender transmits 10 Kb data to receiver and shifts its internal clock by an offset of 1μs for each round. When sender and receiver time periods align, communication capacity is maximum. (b) As time periods align, contention increases and hence the difference between a 0 and 1 increases (blue v. red). When further shifts throw time periods out of alignment, load instructions per time period decreases (red v. black). (c) Across multiple rounds of communication, accuracy in transmission improves with better alignment of time periods (rounds 1 through 10) and decreases once the time periods go out of alignment (rounds 11 through 19) and improve again when the periods fall back into alignment (round 20).



```
Receiver:
for n < 1 to 100 do
    receiver_clock = get_time();
    request_clock();                      -->
    sender_clock = receive_clock();       <--
    latency = get_time() – receiver_clock;

  /* Adjust for the message latency */
    sender_clock -= latency/2;

  /* Keep track of the offset */
    total_offset += (sender_clock – receiver_clock)
end for

/* Send offset to Sender*/
offset = total_offset/100;
send_offset(offset);                      -->

Sender:
for n < 1 to 100 do
    receive_request();
    clock_time = get_time();
    send_clock(clock_time);
end for

clock_offset = receive_offset();
```

Fig. 3: Explicit synchronization protocol.



```
Sender:
for n < 1 to 20 do
    send_covert_bits(10000);              -->

    /* Shift clock by 1 μsecond */
    start_time = start_time + 1;
end for

clock_offset = receive_offset();          <--

Receiver:
for n < 1 to 20 do
    receive_signal(data_array[n]);
end for

/* Find the best round of communication */
best_round = calculate_best_round(data_array);

/* Send offset for sender to adjust clock */
offset = find_offset(best_round);
send_offset(offset);
```

Fig. 4: Synchronization protocol using covert channels to achieve fine-grained alignment.

communicating. For this purpose, the sender uses a pilot signal to mark the start of each message. In our experiments, we use a pilot signal comprising of five hundred "10" pairs of bits followed by a hundred consecutive 0s. We chose this sequence because a) a repeating 10 pattern creates large number of oscillations that a receiver can easily distinguish from noise, and b) the hundred 0s allow the receiver to detect a flatline signal and recover from errors in decoding the first few 10 oscillations.

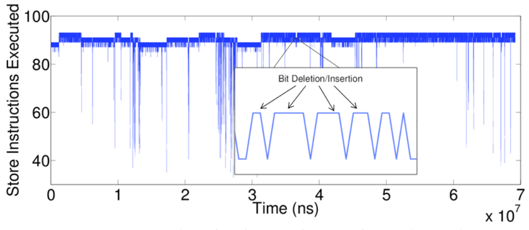
C. Communication through Contention

Performance counter based channels. Since adding noise to the time-stamp counter instruction has been shown to obfuscate purely timing channels, we constructed covert channels using performance counters including L1, L2, and L3 cache misses, branch mispredictions, and load and store instruction counters. Transmission of a 0 or a 1 thus relates to a low or high increment in a performance counter.

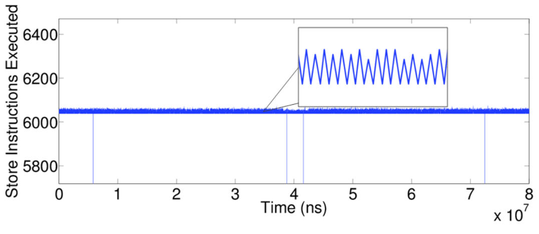
In cache channels, the sender transmits a 1 by accessing cache-line sized data in random order from a working set just greater than the cache size. The sender thus evicts the receiver's data, which is repeatedly accessing a cache sized array, and effectively creates contention for the cache. To transmit a 0, the sender minimizes contention by busy waiting in a loop for one time period. Since L3 caches are larger, we implemented an additional access-driven channel where the sender and receiver identify and then contend for a small number of shared cache sets. While an L3 access driven channel has much higher bandwidth than a timing driven one that contends for the entire cache, the latter can detect contention for any part of the L3 cache.

The branch sender transmits a 1 by executing a parameterizable number of branches, each with 50% probability of being taken, which decreases the branch prediction accuracy of the receiver. To transmit a 0, the sender busy waits to lower its impact on the receiver's branch prediction. The receiver process executes a large number of branches, each with 50% probability, in a loop. Cache and branch channels cause the sender and receiver to execute alternately and are thus based on Prime-Access-Probe method.

Load and store counter channels involve the sender and receiver executing in parallel, where the load (store) sender issues a large number of loads (stores) in a time slot to communicate 1 and does not issue memory access instructions to communicate a 0. The corresponding receivers execute parameterizable number of load (store) instructions in a loop. Load and store instructions are also good candidates for detecting co-resident attackers since they can be constructed to detect contention in both CPU pipeline (instructions committed per cycle) and the memory hierarchy – we use the load channel to detect attackers in Section 6.



(a) Unsynchronized store instructions channel



(b) Synchronized store instructions channel

Fig. 5: (a) Unsynchronized communication leads to poor channels where phantom bits get inserted or real bits get deleted during transmission. This reduces channel capacity. (b) Once synchronized precisely, the signal is stable and the number of bits transmitted is equal to the number received. Thus, only simple bit-flip errors have to be detected.

Memory bus and AES timing channels: The sender and receiver VMs execute in parallel for both these channels. For the memory bus channel, the receiver measures the time required to perform a fixed number of memory bus accesses by issuing cache bypassing instructions. The sender transmits a 1 by issuing bus locking instructions to cause contention for the bus and delaying the receiver's memory traffic. Intel added 6 instructions to their microprocessors to speed up AES encryption/decryption. We exploit contention for these resources using a receiver that continuously executes AES instructions and records the bandwidth available per time slot. The sender issues its own set of AES instructions to send a 1, increasing contention for the 4 available instruction slots in the AES units, or busy waits to send a 0.

Interestingly, we observed that increasing the number of AES instructions per time slot increases the CPU frequency. This sudden change in frequency scaling prevents the receiver from accurately determining the latency of each probe. To prevent frequency scaling from interfering with the channel, we keep the AES workload at a (moderate) rate that we determined using experiments. Finally, we note that both performance counter and timing channels include inter-core as well as intra-core channels.

Multi-input multi-output covert channels. We further construct multiple covert channels on the same core and across cores. For attackers, a multi-input multi-output (MIMO) across covert channels helps create more resilient channels, but at the same time detectors also benefit from listening to multiple channels for eavesdroppers. For intra-core channels, we combine the all senders into one process and the receivers into another. This is crucial since it prevents senders' and receivers' processes from being scheduled out by the OS, causing the precise synchronization to break and bandwidth to drop. Within each process, we schedule the sender and receiver pairs for a part of the overall repeating time-slot, guaranteeing that the correct receiver will be probing the channel simultaneously with its sender.

We present experimental capacity measurements in Section 5, and continue next with a model that captures all the channels presented here.

4. Information-theoretic Model of Covert Channels

In this section, we develop a novel mathematical abstraction to accurately model shared-resource based covert channels. We use a timing-driven cache channel as an example to illustrate this model and then discuss how it applies to other channels. Consider a cache that is accessed by three processes, Alice, Bob and Eve. Each process can load memory addresses into the cache by executing a load command and the most recent loads are kept in the cache by the processor. In the simplest case, assume that each process loads a large amount of data so that the cache is fully overwritten with each operation. When Alice loads her data, this stays in the cache if nobody else makes a request. When Alice loads the same data again (performing a *Probe* operation), the operation will be faster (or a performance counter will indicate a cache hit), so Alice will know that her data is still there.

We model the shared resource (cache in this case) as a bucket — shown in Figure 6 — where each process can place water (data). We emphasize that water is labeled here: there is Alice's water, Eve's water, and Bob's water. When processes load large amounts of data in each operation, the bucket fully refills and there will only be water from one process at any given time. In the Bucket Model, each process can perform two primitive actions: *Write:* Fill the bucket with your water, and *Probe:* Check if *your* water is still there. Eve, a malicious process, will try to eavesdrop or block the communication between Alice and Bob. In microarchitectural channels, and hence in the Bucket model, processes cannot fake someone else's identity since the underlying architecture and OS enforce the absence of explicit flows (e.g., a process cannot read another's cache lines explicitly).

That Eve *must contend* for the shared resource (the bucket) in order to obtain any information from the channel is the fundamental difference of microarchitecture channels versus network covert channels. Specifically, communicating one bit through contention is equivalent to one Write operation followed by a Probe operation – in a microarchitecture channel the two operations in the Bucket Model coalesce into a single Write-and-Probe operation. To see this, observe that reading one bit in the cache channel requires Eve to load her data in the cache and measure the time required to do so (Write), let Alice execute for some time, and then Eve will once again measure the time to load her data in the cache (Probe). In contrast, in a network channel, Alice can Write into the shared medium by varying inter-packet delay or packet sizes while Eve Probes the stream of packets without Writing any packets to the network herself.

While the above example used a timing driven channel where the sender and receiver alternate execution, the insight that "Writes and Probes coalesce into one operation" is true for other microarchitecture channels as well: both Write and Probe become simply a "Use resource" operation. Specifically, we observe an access

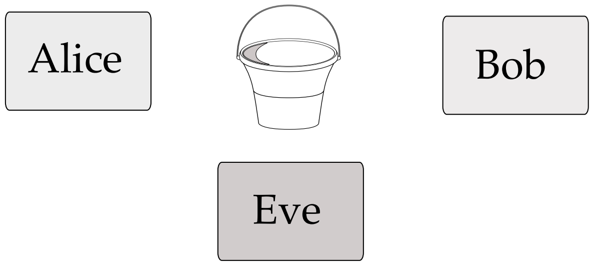


Fig. 6: Alice, Bob and Eve communicating through a shared resource abstracted by our bucket model. Alice, Bob and Eve have two operations: *Write*: add their water into bucket, and *Probe*: ask if their water is still in the bucket. When somebody adds their water into the bucket, previous content is discarded. Users can only ask if their water is currently in the bucket.

driven channel simply encodes one bit of information into the choice of a bucket instead of whether the bucket was written to or not – Probing a specific bucket (e.g., cache line) still Writes to that bucket. Similarly, a channel with sender and receiver operating in parallel (such as the load instruction or AES channels) effectively turns the load instruction bandwidth into the bucket – if Eve sees low bandwidth it means the bucket does not have her water, while high bandwidth implies the bucket has Eve's water. To Probe this channel, Eve still has to issue load/AES instructions and time them, effectively Writing to the channel.

We will show that this simple model captures a surprising number of properties of covert channels and can be used to develop provably secure and robust communication and detection strategies. In this section we make the following assumptions:

A1. No Background Noise: In a real system, OS and other background processes will sometimes influence the caches and the other covert channels adding noise into the bucket: Alice might add her water and some (or all) of it might be disposed even if Eve and Bob remain idle due to background processes adding their water. We observed and measured the amount of such background noise and show how it can be handled statistically in Section 6.

A2. Time Synchronization: We assume that Alice and Bob are synchronized and operate within a time-slotted protocol. This was achieved in our experiments by accessing the system clock and agreeing on a protocol for time-slot beginning and ending. We further assume that Eve is aware of this synchronization and further can interject at arbitrary times between the assigned time slots.

Figure 7 shows a simple communication protocol for Alice to send bits to Bob. Bob writes and probes and Alice chooses to send a zero by doing nothing (N) or write (W). Bob decodes a zero when he finds his water still present using a probe (P). In time-slots 7-9 Eve writes her water right after Bob and probes right before he does, essentially imitating Bob. That way she is able to eavesdrop the fact that Alice tried to send a one since Eve's probe will reveal that her water is no longer there, displaced by Alice's write. Note that in time-slot 9 Bob will still decode a one. This leads to the following simple but fundamental lemma:

Lemma 1: *Eve can flip* $0 \to 1$ *but it is impossible to flip* $1 \to 0$*. Further, when Eve eavesdrops, Bob will always decode a 1 regardless of what was transmitted.*

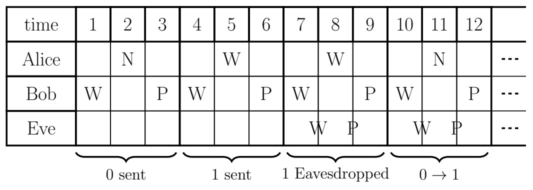


Fig. 7: Twelve time-slots of our simple communication protocol and Eve interjecting and jamming. Each communication round consists of three time-slots. In the first of these Bob performs a write (W) and fills the bucket with his water. In the second time-slot he waits and in the third performs a probe (P) to check if his water is still there. Alice acts in the intermediate time-slots by either doing nothing (N) to send a 0 or by writing (W) that fills the bucket with her water. When Bob probes if he finds his water still there he decodes a 0 otherwise a 1. In time-slots 1-3 a 0 is sent and in 4-5 a 1 is sent. In time-slots 7-9 Eve interjects her own water and probes right before Bob. In time-slots 10-12 she interjects to jam communication by changing a 0 into a 1.

Proof: Bob decodes a 0 when he finds his own water still present in the bucket when he probes. The only way that his water is still there is if Eve and Alice do not displace it by writing. This is because it is impossible for any process to Write water with someone else's identity in the bucket. Therefore, when Bob reads a 0, this is a certificate that both Alice and Eve have remained idle and hence it is impossible to flip $1 \to 0$.

The second claim relies on the fact that a probe by Eve will contain no information unless it follows a write by Eve. This again is because a probe only reveals if Eve's water is still there. Therefore every eavesdropping sequence of time-slots will displace Bob's water and will be read by him as a 1. □

There are, potentially, three goals for a malicious process: disrupt communication between good processes, eavesdrop on communication and activity and finally, remain undetected. It can be seen that in our simple model Eve has a lot of power since she can essentially simulate Bob's behavior.

We now show how a very simple code can provide strong secrecy and integrity guarantees. This relies on the unique characteristic of the covert channels: that one has to use the resource (*i.e.* write) to communicate or eavesdrop. Define the simple *Differential Code* that maps a bit $\mathbf{x}$ as follows:

$$\mathbf{x} \to \{\mathbf{x}, (1 - \mathbf{x})\}. \tag{1}$$

This maps a $0 \to \{0, 1\}$ and $1 \mapsto \{1, 0\}$ and has rate $1/2$ (*i.e.* maps one bit into two).

Theorem 1: *The rate* $1/2$ *Differential Code provides the following guarantees. If Eve tries to eavesdrop, she is detected with probability* $1/2$ *per communicated bit. Further, Eve cannot introduce any bit error without being detected.*

Proof: Assume, without loss of generality, that Eve tries to observe the first symbol. She must interject a write operation before Alice. If Alice wants to transmit 01 then the first 0 will be decoded as a 1 by Bob since Eve's write removed Bob's water from the bucket. Now Eve knows that 01 was the intended symbol but further knows that her detection is unavoidable: the next bit will be a 1 and there is no way to convert it into a zero by lemma 1.

So Bob will decode 11 which is not a symbol of the differential code and can only originate from malicious behavior in our noiseless model. Subsequently, Bob can sound a public alarm and stop the service. The second case is if Alice transmits 10. Then Eve will capture the 1 and stay silent in the next symbol knowing it must be a 0. In this case Eve is successful and undetected. However, since each symbol is equiprobable and independent, she is detected with probability $1/2$ per communicated bit as claimed.

To see the second claim, observe that the code $\{01, 10\}$ maps to the set of symbols $\{01, 10, 11\}$ and any action by Eve during the transmission of a 0 can only produce the symbol 11. Since this is not part of the code, it is immediately detected as the result of malicious behavior. $\square$

This theorem shows that it is possible to obtain very strong secrecy and integrity in Bucket channels. The differential code we presented for Bucket channels allows Alice and Bob to communicate while detecting Eve. But we can devise a simpler protocol if detecting Eve is the sole goal. All friendly VMs elect a leader VM to run the detection algorithm and agree on a specific time to begin the detection process. The detector then transmits a stream of 0s, reads every bit, and raises an alarm if it detects even a single bit flip from 0 to 1. We start with this simple protocol as the baseline in Section 6.

5. Covert Channel Capacities

Having set up covert channels, our next task is to quantify the capacity of these channels in a noisy setting. The channel capacity is by definition [37] the peak communication rate achievable using any coding scheme. We assume that our channels are *memoryless* – *i.e.* that the symbols and noise are independent across time. Subsequently, we empirically establish that most of our channels are indeed memoryless at the time resolution that we use them.

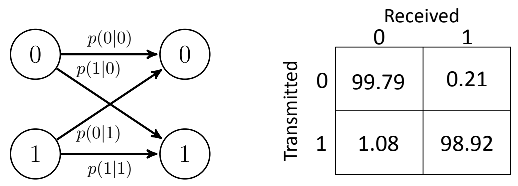
A discrete memoryless channel (DMC) with input alphabet $\mathcal{X}$ and output alphabet $\mathcal{Y}$ is characterized by its transition probabilities

$$P_{Y|X} = \{p(y|x)\}_{x\in\mathcal{X}, y\in\mathcal{Y}}. \quad (2)$$

Here, $p(y|x)$ denotes the probability that channel outputs y when x is input to the channel. Note that the outcome of a discrete memoryless channel y depends only on its current input x (through $P_{Y|X}$) and is independent of the past and future inputs to the channel. For a DMC with transition probabilities $P_{Y|X}$, the capacity (measured in bits per channel use) is given by Shannon's theorem [37]:

$$C = \max_{p(x)} I(X;Y). \quad (3)$$

Here, $I(X;Y)$ denotes the mutual information between X and Y; and the maximization is performed over all probability distributions over the input alphabet $\mathcal{X}$. In our setting, we transmit a single bit over a covert channel in one channel use. Therefore, we have $\mathcal{X} = \{0, 1\}$ as the input alphabet of the channel. Given specific values of transition probabilities for a covert channel – Figure 10 shows an example for the L1 channel – its channel capacity can be obtained as described in (3). Note that the computation of the capacity involves maximization of mutual information over all possible distributions on $\{0, 1\}$.



		Received 0	Received 1
Transmitted	0	99.79	0.21
	1	1.08	98.92

Fig. 10: Bit flip probabilities to be estimated for each binary asymmetric channel (left). (Right) Measured probabilities for the L1 channel at 50 Kbps.

To solve this optimization problem we use the Blahut-Arimoto algorithm [38], [39] for capacity computation.

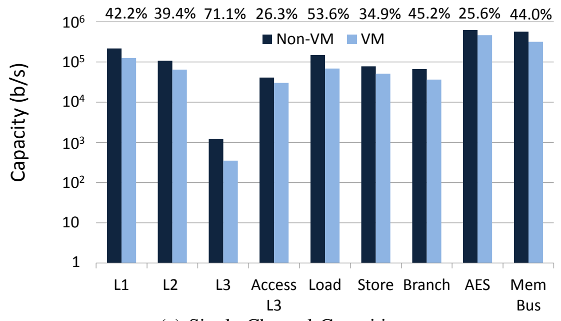
A. Measured Capacities

We ran the experiments on an Intel Quad-Core i7-2600 running at 3.40GHz with a 32kB L1i and L1d cache, a 256KB L2 cache, and an 8MB L3 cache. Figure 8 (a) and (b) shows the capacity of channels that we established in our experiments for single and multiple channels respectively (with the drop due to heavy-weight virtualization represented on top of the graphs). As we discuss below, these numbers are much higher than reported on EC2 or estimated through back of the envelope computations. Figure 9 shows how capacity (blue line) and error rate (bits per channel use; red line) vary with frequency of each covertly transmitted bit. This is interesting because increasing the frequency at first improves capacity but makes error rates worse. Decreasing the frequency reduces error rates but also lowers the potential capacity. Figure 9 also shows the frequencies for which capacities (blue lines) hit peak values in each channel.

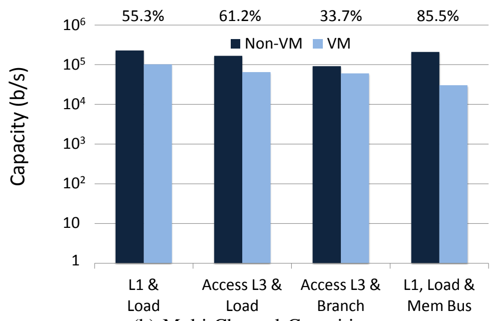
For our experiments, we found capacities for individual and joint channels across both processes and virtual machines. For the cross-process tests, we pinned two processes to specific cores using taskset and ran 20 rounds of communication for every frequency tested. In each round, we communicated a 10,000 bit message across the channel. To determine the difference between a 0 and 1 on the channel, we used kmeans clustering on the performance counter or access latency values. We then used cross correlation to determine the percentage of bits that were flipped during communication. For our virtual machine experiment, each VM was launched using QEMU with KVM enabled. We used the libvirt library [40] to pin a specific virtual CPU to a hardware CPU. This allows for both intra-core and inter-core communication between two VM's when determining joint capacities for channels.

Performance counter channels. Figure 9 shows the theoretical capacities for each channel. Capacities range from only 1.21 Kbps for the timing-driven channel that uses all of L3 to 216.97 Kbps for L1. Contending for only a few sets of L3, predictably, increases its bandwidth and we determine it to be 40.87 Kbps. L3 also has the benefit of being a cross-core channel while L1 and L2 are intra-core channels. Interestingly, the branch predictor channel has a capacity of 66.53 Kbps – branch channel has only been used to infer bits of keys before. Load and Store performance counters also have a surprisingly high bandwidth (148.56 Kbps and 78.44 Kbps respectively).

Timing Channels. We measure capacities of 565.62 Kbps for the memory bus contention channel (MemBus in Figure 9) and 624.97 Kbps for AES-NI instructions. Interestingly, the MemBus channel's peak capacity is achieved with a frequency of 1.1 MHz, which shows how



(a) Single-Channel Capacities



(b) Multi-Channel Capacities

Fig. 8: (a) Capacities for inter-core (L3 and Memory Bus channels) and intra-core (all other) channels. In particular, AES and Mem Bus channels at 500+ Kbps are particularly attractive for attackers due to their high bandwidth. (b) Joint capacities for an intra-core, two inter-core, and a 3-way inter-core combination. Sometimes joint capacity is lower than individual ones (e.g., for MemBus) because of mutual interference. However, joint channels provide more coverage when used for defense. Note the log scale for the y-axis.

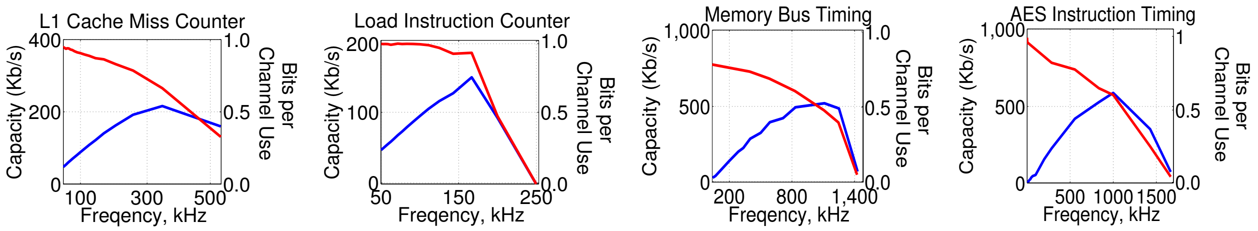


Fig. 9: Capacity (blue) and error rate (red, inverse of bits per channel use) varying with channel frequency. As frequency increases, capacity increases but so does error rate (red line drops). Capacity thus achieves a peak value and decreases.

operating the channel faster with an optimal error rate yields a higher capacity than operating the channel with minimal errors. Overall, our results show that capacity can be substantially improved beyond the 100 bps bandwidth achieved on EC2 [2].

Multi-input Multi-output Communication. We measured the joint capacities of some interesting channel combinations. L1 and Load channels run on a shared core and have a capacity of 228.72 Kbps – just beyond that of the L1 channel itself, which indicates substantial mutual interference between L1 and load channels. Inter-core combinations such as L3 access driven paired with either load or branch channel interfere less – hence the joint capacity is closer to the sum of each channel's capacity. Finally, we measured three channels running concurrently and find that the combination of MemBus, Load, and L1 can operate at 209.75 Kbps. This indicates that there is substantial mutual interference that limits the joint capacity well below that of MemBus alone (although still above just Load and L1 individually).

Our mutual interference results indicate that running multiple channels can benefit the attacker by simultaneously leaking data over intra and inter core channels (such as L3 and branch channels). At the same time, multiple channels when used by a detector increase the chance of exposing an attacker.

Light-weight v. Heavy-weight Virtualization. Our baseline results above are measured for virtualization techniques like LinuX Containers (LXC/Docker [41], [16]) where VMs are almost as light-weight as processes. We also measure capacities for channels with a heavy-weight hypervisor (KVM [42], using hardware virtualization) which results in noisier channels. We find that this decreases capacities for all channels on average by 41.7% (Figure 8).

We have also quantified the memory of each channel but defer the details of this study to an anonymous technical report [43]. We found that channels like load, store, L1, and other caches have low memory – these can be controlled precisely by a sender and hence are well-suited for detecting attackers – while the branch predictor channel exhibits long range memory. This indicates that branch predictor channels have a higher error rate and also are a poor choice for detecting malicious co-residents.

6. Detection Games Using Covert Channels

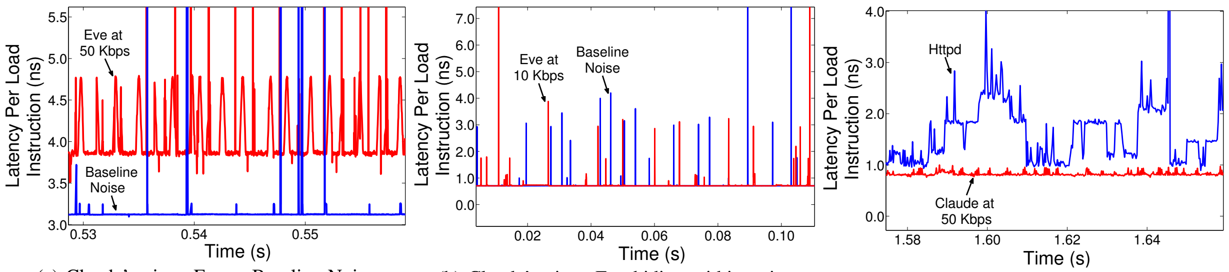
In this section we discuss the problem of how a friendly VM (Claude) can detect an intelligent, co-resident eavesdropper (Eve). As we showed in *Lemma 1*, Eve (in fact, any process) can flip $0 \to 1$ but not flip $1 \to 0$ in Claude's transmission. It follows that writing a 1 is useless for detecting Eve – even if Eve eavesdrops during that time slot, Claude will still read a 1 and hence Eve will remain undetected. For this reason, the optimal Eve detection pattern for Claude to transmit will be a consecutive transmission of 0s (similar to HomeAlone's proposal [3] of leaving cache lines untouched).

We show that, surprisingly, in the presence of noise, an all 0s detection pattern is no longer optimal. The intuition is that Eve can perform a few probes while Claude runs his detector and still remain undetected. This is because, besides Eve, the background noise can also cause evictions of Claude's data. Therefore, Claude cannot easily distinguish between the evictions caused only by the noise and the evictions caused by both the noise and the eavesdropping probes unless Eve's evictions are statistically more frequent. Therefore, Eve can *hide below the noise and try to detect Claude's detector*. In response, Claude now has a reason to *add noise into the detection pattern* to make it harder for Eve to detect Claude.

We now step through the three stages in this detection game between Claude and Eve. Claude and Eve both start with static strategies of probing the channel (Stage 1).

Intelligent Principals	Claude	Eve	Result
Neither	Probes @ 50 Kbps	Probes @ 50 Kbps	Eve is easily detected
Eve	Probes @ 50 Kbps	Hides in noise until Claude has finished detection	Claude cannot distinguish between Eve and noise
Both	Intelligent detector mimicking httpd	Attempts to hide from Claude to avoid detection	Eve cannot distinguish between httpd and Claude, and is detected

TABLE I: Three stages in the detection game between Claude and Eve.



(a) Claude's view: Eve v. Baseline Noise (b) Claude's view: Eve hiding within noise (c) Eve's view: Claude v. httpd

Fig. 11: (a) Stage 1: Claude wins since noise and Eve at 50 Kbps are markedly different. (b) Stage 2: Eve reduces probe bandwidth to 10 Kbps and Claude cannot reliably tell Eve and Baseline noise apart. (c) Eve, however, can tell Claude apart from the target application (`httpd`).

Eve then adapts by lowering her probe bandwidth to hide behind the baseline noise and detect Claude (Stage 2). Finally, in Stage 3, Claude adapts by adding noise to the channel to *mimic* his own application – Eve gets exposed as anomalous interference which turns 0s into 1s. Table I summarizes these stages and the outcome at each stage.

The key reason why Claude wins this game is because he controls the noise in the channel (by switching between the detector and the benign program) while Eve is forced to respond and hide in the noise. Eve's task of distinguishing between the application and Claude's mimicry is much harder than Claude's task of distinguishing between low noise and Eve's probes. Claude can thus force Eve into staying at low probing frequencies that also leak less information. We now demonstrate this experimentally.

All experiments in this section represent Claude and Eve using virtual machines on KVM pinned to separate hardware threads on the same core. Claude's Apache web server (`httpd`) is Eve's target application. We select the load instruction channel for Claude to detect Eve with, since it is sensitive to interference from even simple programs with negligible cache footprints.

Stage 1. Neither Claude nor Eve attempt to hide their detection/attack schemes. Prior to performing detection, Claude first generates a baseline model (defines a 0) for the load channel by probing without any additional processes running (to minimize noise). Next, Claude actively monitors the channel by remaining silent, and detects Eve when she perturbs the channel (sends a 1). Figure 11a shows Claude's time-line view of the channel – plotting Claude's average load latency per time slot – when he probes the channel at 50 Kbps. The two time series correspond to two experiments, one with baseline noise (blue) and the other with Eve probing at 50 Kbps (red). Eve is easily detectable because Eve's activity significantly alters the state of the channel from a 0 into a 1.

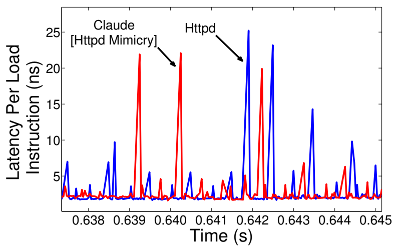
Stage 2. Eve attempts to hide from Claude by a) lowering the frequency of her probes to appear within the noise margin to Claude, and b) distinguishing Claude's detector from the baseline application. Once Claude's detector is off, Eve can resume probing at higher frequencies.

Figure 11a shows Claude's view of how Eve is able to hide behind the noise of the channel by sleeping for 100μs between each probe. Eve looks very similar to baseline noise. At the same time, in Figure 11b we can see Eve's view when she lowers her probe rate to 5 Kbps – Eve can clearly distinguish Claude's detector at 50 Kbps from `httpd` and stay low.

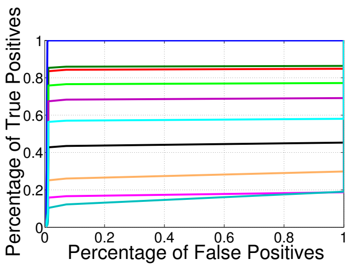
Figure 12b shows the true positive vs. false positive detection rates for each threshold tested as Eve varies her probing rate. As expected, longer calls to sleep increases Eve's ability to hide from Claude (the "knee" of the curve becomes lower). We also find that in order for Eve to successfully hide from Claude, she must call sleep and stop executing any instructions on the hardware. It is not sufficient to perform an empty busy wait, as even this causes enough noise on the load channel for Claude to detect.

Stage 3. Claude now adds noise into the detection pattern (alters his workload during the detection phase) to prevent Eve from hiding. However, because we assume that Eve knows `httpd`'s characteristics, it is not sufficient for Claude to write 1s at arbitrary times as this may be distinguishable from a normal workload. Instead, Claude's noise mimics the channel characteristics of the benign workload and prevents Eve from identifying the detection phase of Claude. Eve must either come out of hiding, allowing Claude to detect her presence, or remain inactive. In either scenario, Claude has successfully prevented Eve from carrying out an attack.

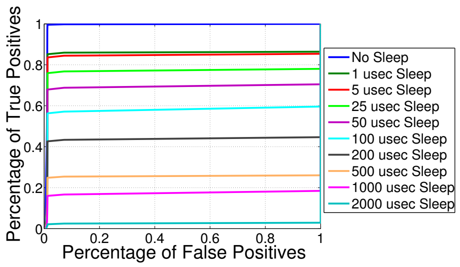
Specifically, we generate a Markov Model for `httpd` by sampling its load instruction performance counters using the Linux `perf` tool at a frequency of 500 microseconds for 30 minutes. We ran `httpd` in a VM pinned to a single hardware thread and used `httperf` to issue 100 requests per second to load Yahoo's homepage. The rate at which we are able to record the performance counter values of `httpd` yields the length of each state of the model. Thus, Claude now probes the channel at 2 Kbps. Next, we transformed the recorded counter values into a Markov Model by clustering the data into 5 states. Here, each state represents the number of load instructions Claude must execute in order to simulate the noise caused



(a) Eve's view: Claude mimicking Httpd v. Httpd



(b) Stage 1 Claude: true v. false positives



(c) Stage 3 Claude: true v. false positives

Fig. 12: (a) Claude adds noise into the channel by creating a Markov Model of `httpd` and mimicking the effect that `httpd` has on the load channel. To Eve, Claude looks like `httpd` but Claude's detector can look for additional noise in the channel to identify Eve. (b) Detection v. false positive results for the unintelligent Claude (Stage 1) generated by varying detection threshold (number of 1s before alarm is raised). Eve at 50 Kbps is almost 100% detectable. Forcing Eve to 10 Kbps (100μs sleep) still yields a detection rate around 40%. (c) Intelligent Claude (Stage 3), by adding noise into the channel should hamper its own detector, yet detection rates are close to that of Stage 1 Claude.

by httpd. Claude can then transition through the states of the model, mimicking the load instructions executed by httpd. Research on synthetic workload generation [44], [45] can be used here to considerably improve Claude's mimicry of `httpd`.

As seen in Figure 12a, it is difficult for Eve to determine the difference between Claude's execution and that of `httpd`, even when Eve samples the channel at 50Kbps. Figure 12c quantifies the true-positive v. false-positive detection rates for different probing frequencies of Eve when Claude mimics `httpd`. When compared to Figure 12b, we see there is not much degradation in Claude's ability to detect Eve. Therefore, mimicking a workload's channel characteristics comes at little cost to Claude's ability to successfully detect Eve.

Finally, note that time to detection is small, since the knee of each curve is close to the Y-axis. We attribute this to each bit's time period in the channel being a long time at processor timescales; for example, 20μs for a 50 Kbps channel. Thus even a few 1s in the channel are sufficient to reach the potential for detectability. Increasing the threshold for 1s further only increases the false positive rate.

7. Conclusion

Third-party software should not leak secrets through the microarchitecture, yet several covert channels have been demonstrated that do so. Indeed, we demonstrate that by careful synchronization, extremely high capacities can be achieved. Our Bucket model captures the contention-driven nature of microarchitecture channels and shows how microarchitecture channels force Eve to leave indelible footprints by turning 0s into 1s. In real systems with noisy channels and adaptive adversaries, this insight leads to a detection game where both detector and eavesdropper do their best to hide. However, the detector has the advantage of controlling the channel which tips the game in its favor. For architects, this work lays an information theoretic foundation for a long-studied problem and opens up a new direction in protecting and detecting microarchitectural channels – exposing contention and its sources directly to software that can then run precise covert channel detectors.

Acknowledgments

The authors would like to thank the anonymous reviewers for insightful comments on this paper. This work was funded in part by Grants NSF CNS-1314709 and NSF CNS-1263341.

References

[1] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud! Exploring information leakage in third-party compute clouds. In Somesh Jha and Angelos Keromytis, editors, *Proceedings of CCS 2009*, pages 199–212. ACM Press, November 2009.

[2] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: high-speed covert channel attacks in the cloud. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[3] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 313–328, Washington, DC, USA, 2011. IEEE Computer Society.

[4] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael M. Swift. Resource-freeing attacks: improve your cloud performance (at your neighbor's expense). In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 281–292, New York, NY, USA, 2012. ACM.

[5] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM.

[6] Yuval Yarom and Katrina Falkner. Flush+reload: a high resolution, low noise, l3 cache side-channel attack. Cryptology ePrint Archive, Report 2013/448, 2013. http://eprint.iacr.org/2013/448.pdf.

[7] Zhenghong Wang and Ruby Lee. New cache designs for thwarting software cache-based side channel attacks. *J. ACM*, 2007.

[8] Yao Wang and G.E. Suh. Efficient timing channel protection for on-chip networks. In *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, pages 142–151, May 2012.

[9] Hassan M. G. Wassel, Ying Gao, Jason K. Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 583–594, New York, NY, USA, 2013. ACM.

[10] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA conference on Topics in Cryptology*, CT-RSA'06, pages 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[11] Aciicmez, Onur and Koc, Cetin Kaya and Seifert, Jean-Pierre. On the power of simple branch prediction analysis. In *Proceedings*

of the 2nd ACM symposium on Information, computer and communications security, ASIACCS '07, pages 312–320, New York, NY, USA, 2007. ACM.

[12] Mohit Tiwari, Hassan Wassel, Bita Mazloom, Shashidhar Mysore, Frederic Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *ASPLOS*, March 2009.

[13] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 493–504, New York, NY, USA, 2009. ACM.

[14] J. Oberg, Wei Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. Theoretical analysis of gate level information flow tracking. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 244 –247, june 2010.

[15] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 118–129, Washington, DC, USA, 2012. IEEE Computer Society.

[16] Docker. https://www.docker.com.

[17] Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

[18] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. In *in Proc. Cryptographic Hardware and Embedded Systems (CHES) 2006. Lecture Notes in Computer Science*, pages 201–215. Springer, 2006.

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

[20] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of l2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 29–40, New York, NY, USA, 2011. ACM.

[21] Onur Aciicmez. Yet another microarchitectural attack: Exploiting i-cache. In *CCS Computer Security Architecture Workshop*, 2007.

[22] Onur Aciicmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *Proceedings of the 12th international conference on Cryptographic hardware and embedded systems*, CHES'10, pages 110–124, Berlin, Heidelberg, 2010. Springer-Verlag.

[23] Onur Aciicmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography*, FDTC '07, pages 80–91, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Zhenghong Wang and Ruby B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 83–93, Washington, DC, USA, 2008. IEEE Computer Society.

[25] Amir Houmansadr and Nikita Borisov. Coco: Coding-based covert timing channels for network flows. In *Proceedings of the 13th International Conference on Information Hiding*, IH'11, pages 314–328, Berlin, Heidelberg, 2011. Springer-Verlag.

[26] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan. Side-channel vulnerability factor: A metric for measuring information leakage. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pages 106–117, 2012.

[27] Tianwei Zhang, Fangfei Liu, Si Chen, and Ruby B. Lee. Side channel vulnerability metrics: The promise and the pitfalls. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 2:1–2:8, New York, NY, USA, 2013. ACM.

[28] Baris Kopf and David Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *Computer Network Security*, 2010.

[29] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. Stealthmem: system-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 11–11, Berkeley, CA, USA, 2012. USENIX Association.

[30] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 2nd ACM workshop on Computer security architectures*, CSAW '08, pages 25–34, New York, NY, USA, 2008. ACM.

[31] Yinqian Zhang and Michael K. Reiter. D&#252;ppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 827–838, New York, NY, USA, 2013. ACM.

[32] Michael Mitzenmacher et al. A survey of results for deletion channels and related synchronization channels. *Probability Surveys*, 6:1–33, 2009.

[33] Mojtaba Rahmati and T Duman. Bounds on the capacity of random insertion and deletion-additive noise channels. 2013.

[34] Suhas Diggavi and Matthias Grossglauser. On information transmission over a finite buffer channel. *Information Theory, IEEE Transactions on*, 52(3):1226–1237, 2006.

[35] Maxwell Krohn. *Information Flow Control for Secure Web Sites*. PhD thesis, MIT, 2008.

[36] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making Information Flow Explicit in HiStar. Berkeley, CA, USA, 2006. USENIX Association.

[37] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, New York, NY, USA, 1991.

[38] S. Arimoto. An algorithm for computing the capacity of arbitrary discrete memoryless channels. *IEEE Transactions on Information Theory*, 18(1):14–20, Jan 1972.

[39] R.E. Blahut. Computation of channel capacity and rate-distortion functions. *IEEE Transactions on Information Theory*, 18(4):460–473, Jul 1972.

[40] Libvirt virtualization library. http://libvirt.org.

[41] LXC–Linux Containers. http://linuxcontainers.org.

[42] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, 2007.

[43] Anonymous Tech Report. https://github.com/covert-channels/defense/wiki.

[44] Mario Badr and Natalie Enright Jerger. Synfull: Synthetic traffic models capturing cache coherent behaviour. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 109–120, Piscataway, NJ, USA, 2014. IEEE Press.

[45] A Awad and Y. Solihin. Stm: Cloning the spatial and temporal memory access behavior. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 237–247, Feb 2014.