Cross-VM Side Channels and Their Use to Extract Private Keys

Yinqian Zhang
University of North Carolina
Chapel Hill, NC, USA
yinqian@cs.unc.edu

Ari Juels
RSA Laboratories
Cambridge, MA, USA
ari.juels@rsa.com

Michael K. Reiter
University of North Carolina
Chapel Hill, NC, USA
reiter@cs.unc.edu

Thomas Ristenpart
University of Wisconsin
Madison, WI, USA
rist@cs.wisc.edu

ABSTRACT

This paper details the construction of an access-driven side-channel attack by which a malicious virtual machine (VM) extracts fine-grained information from a victim VM running on the same physical computer. This attack is the first such attack demonstrated on a symmetric multiprocessing system virtualized using a modern VMM (Xen). Such systems are very common today, ranging from desktops that use virtualization to sandbox application or OS compromises, to clouds that co-locate the workloads of mutually distrustful customers. Constructing such a side-channel requires overcoming challenges including core migration, numerous sources of channel noise, and the difficulty of preempting the victim with sufficient frequency to extract fine-grained information from it. This paper addresses these challenges and demonstrates the attack in a lab setting by extracting an ElGamal decryption key from a victim using the most recent version of the `libgcrypt` cryptographic library.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection—*Information flow controls*

General Terms

Security

Keywords

Side-channel attack, cross-VM side channel, cache-based side channel

1. INTRODUCTION

Modern virtualization technologies such as Xen, HyperV, and VMWare are rapidly becoming *the* cornerstone for the security of critical computing systems. This reliance stems from their seemingly strong isolation guarantees, meaning their ability to prevent guest virtual machines (VMs) running on the same system from interfering with each other's execution or, worse, exfiltrating confidential data across VM boundaries. The assumption of strong isolation underlies the security of public cloud computing systems [6, 39] such as Amazon EC2, Microsoft Windows Azure, and Rackspace; military multi-level security environments [29]; home user and enterprise desktop security in the face of compromise [20]; and software-based trusted computing [22].

VM managers (VMMs) for modern virtualization systems attempt to realize this assumption by enforcing logical isolation between VMs using traditional access-control mechanisms. But such logical isolation may not be sufficient if attackers can circumvent them via side-channel attacks. Concern regarding the existence of such attacks in the VM setting stems from two facts. First, in non-virtualized, cross-process isolation contexts, researchers have demonstrated a wide variety of side-channel attacks that can extract sensitive data such as cryptographic keys on single-core architectures [1–3, 5, 8, 36, 43]. The most effective attacks tend to be so-called "access-driven" attacks that exploit shared microarchitectural components such as caches. Second, Ristenpart et al. [39] exhibited coarser, cross-VM, access-driven side-channel attacks on modern symmetric multi-processing (SMP, also called multi-core) architectures. But their attack could only provide crude information (such as aggregate cache usage of a guest VM) and, in particular, is insufficient for extracting cryptographic secrets.

Despite the clear potential for attacks, no actual demonstrations of fine-grained cross-VM side-channels attacks have appeared. The oft-discussed challenges [39, 46] to doing so stem primarily from the facts that VMMs place more layers of isolation between attacker and victim than in cross-process settings, and that modern SMP architectures do not appear to admit fine-grained side-channel attacks (even in non-virtualized settings) because the attacker and victim are often assigned to disparate cores. Of course a lack of demonstrated attack is not a proof of security, and so whether fine-grained cross-VM side-channel attacks are possible has remained an important open question.

In this paper, we present the development and application of a cross-VM side-channel attack in exactly such an environment. Like many attacks before, ours is an access-

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.
Copyright 2012 ACM 978-1-4503-1651-4/12/10 ...$15.00.

driven attack in which the attacker VM alternates execution with the victim VM and leverages processor caches to observe behavior of the victim. However, we believe many of the techniques we employ to accomplish this effectively and with high fidelity in a virtualized SMP environment are novel. In particular, we provide an account of how to overcome three classes of significant challenges in this environment: (i) inducing regular and frequent attacker-VM execution despite the coarse scheduling quanta used by VMM schedulers; (ii) overcoming sources of noise in the information available via the cache timing channel, both due to hardware features (e.g., CPU power saving) and due to software ones (e.g., VMM execution); and (iii) dealing with core migrations, which give rise to cache "readings" with no information of interest to the attacker (i.e., the victim was migrated to a core not shared by the attacker). Finally, we customize our attack to the task of extracting a private decryption key from the victim and specifically show how to "stitch together" these intermittent, partial observations of the victim VM activity to assemble an entire private key.

As we demonstrate in a lab testbed, our attack establishes a side-channel of sufficient fidelity that an attacker VM can extract a private ElGamal decryption key from a co-resident victim VM running Gnu Privacy Guard (`GnuPG`) [24], a popular software package that implements the OpenPGP e-mail encryption standard [14]. The underlying vulnerable code actually lies in the most recent version of the `libgcrypt` library, which is used by other applications and deployed widely. Specifically, we show that the attacker VM's monitoring of a victim's repeated exponentiations over the course of a few hours provides it enough information to reconstruct the victim's 457-bit private exponent accompanying a 4096-bit modulus with very high accuracy—so high that the attacker was then left to search fewer than $10,000$ possible exponents to find the right one.

We stress, moreover, that much about our attack generalizes beyond ElGamal decryption (or, more generally, discovering private exponents used in modular exponentiations) in `libgcrypt`. In particular, our techniques for preempting the victim frequently for observation and sidestepping several sources of cache noise are independent of the use to which the side-channel is put. Even those components that we necessarily tune toward ElGamal private-key extraction, and the pipeline of components overall, should provide a roadmap for constructing side-channels for other ends. We thus believe that our work serves as a cautionary note for those who rely on virtualization for guarding highly sensitive secrets of many types, as well as motivation for the research community to endeavor to improve the isolation properties that modern VMMs provide to a range of applications.

2. BACKGROUND

Side-channel attacks and their use to extract cryptographic keys from a victim device or process have been studied in a variety of settings. These attacks are generally categorized into one of three classes. A *time-driven* side-channel attack is possible when the *total* execution times of cryptographic operations with a fixed key are influenced by the value of the key, e.g., due to the structure of the cryptographic implementation or due to system-level effects such as cache evictions. This influence can be exploited by an attacker who can measure many such timings to statistically infer information about the key (e.g., [4, 11, 13, 26]). In the context of our work, the most relevant research of this type is due to Weiß et al. [46]. This work mounted a time-driven attack against an embedded uniprocessor device virtualized by the L4 microkernel. Their techniques do not translate to the style of attack we pursue (see below) or the virtualized SMP environment in which we attempt it.

A second class of side-channel attacks is *trace-driven.* These attacks continuously monitor some aspect of a device throughout a cryptographic operation, such as the device's power draw (e.g., [25]) or electromagnetic emanations (e.g., [21, 38]). The ability to *continuously* monitor the device makes these attacks quite powerful but typically requires physical proximity to the device, which we do not assume here.

The third class of side-channel attack, of which ours is an example, is an *access-driven* attack, in which the attacker runs a program on the system that is performing the cryptographic operation of interest. The attacker program monitors usage of a shared architectural component to learn information about the key, e.g., the data cache [36, 43], instruction cache [1, 2], floating-point multiplier [5], or branch-prediction cache [3]. The strongest attacks in this class, first demonstrated only recently [2, 8], are referred to as *asynchronous*, meaning that they do not require the attacker to achieve precisely timed observations of the victim by actively triggering victim operations. These attacks leverage CPUs with simultaneous multi-threading (SMT) or the ability to game operating system process schedulers; none were shown to work in symmetric multi-processing (SMP) settings. The contribution of this paper is to extend the class of asynchronous, access-driven attacks to VMs running on virtualized SMP systems.

The closest work in this area is due to Ristenpart et al. [39], who gave an access-driven data-cache side channel sufficient for learning coarse-grained information, such as the current load, of a co-resident victim VM. They did not offer any evidence, however, that fine-grained information such as keys could be extracted through cross-VM side-channels. Subsequent work [48, 49] showed how to build various covert channels in cross-VM SMP settings, but these require cooperating VMs and so cannot be used as a side-channel attacks. Also of interest is work of Owens and Wang [34], who gave an access-driven attack for fingerprinting the OS of a victim VM by leveraging memory deduplication in the VMWare ESXi hypervisor. They did not show how to extract fine-grained information such as cryptographic keys.

To our knowledge, no prior works have demonstrated cross-VM side-channels with sufficient fidelity to extract cryptographic keys, however; this is what we show here. Moreover, as discussed next, the features of virtualized SMP systems are such that new techniques are required to succeed.

3. OVERVIEW AND CHALLENGES

Attack setting. The setting under consideration is the use of confidential data, such as cryptographic keys, in a VM. Our investigations presume an attacker that has in some manner achieved control of a VM co-resident on the same physical computer as the victim VM, such as by compromising an existing VM that is co-resident with the victim.

We focus on the Xen virtualization platform [9] running on contemporary hardware architectures. Our attack setting is inspired not only by public clouds such as Amazon EC2 and Rackspace, but also by other Xen use cases. For example, many virtual desktop infrastructure (VDI) solutions (e.g., Citrix XenDesktop) are configured similarly, where virtual desktops and applications are hosted in centralized datacenters on top of a XenServer hypervisor and delivered remotely to end user devices via network connections. Another representative use case separates operating systems into several components with different privilege levels and that are isolated by virtualization [20, 37]. An example of such systems is Qubes [41], which is an open source operating system run as multiple virtual machines on a Xen hypervisor.

In terms of computer architecture, we target modern multicore processors without SMT capabilities or with SMT disabled. This choice is primarily motivated by contemporary processors used in public clouds such as Amazon AWS and Microsoft Azure, whose SMT features are intentionally disabled, if equipped, since SMT can facilitate cache-based side channel attacks [28].

We assume the attacker and victim are separate Xen DomU guest VMs, each assigned some number of disjoint virtual CPUs (VCPUs). A distinguished guest VM, Dom0, handles administrative tasks and some privileged device drivers and is also assigned some number of VCPUs. The Xen credit scheduler [16] assigns VCPUs to the physical cores (termed PCPUs in Xen's context), with periodic migrations of VCPUs amongst the cores.

Our threat model assumes that Xen maintains logical isolation between mutually untrusting co-resident VMs, and that the attacker is unable to exploit software vulnerabilities that allow it to take control of the entire physical node. We assume the attacker knows the software running on the victim VM and has access to a copy of it.

The attack we consider will therefore uses cross-VM side-channels to reveal a code path taken by the victim application. We will use as a running example—and practically relevant target—a cryptographic algorithm whose code-path is secret-key dependent (look ahead to Fig. 2). However, most steps of our side-channel attack are agnostic to the purpose for which the side-channel will be used.

Constructing such a side channel encounters significant challenges in this cross-VM SMP setting. We here discuss three key challenge areas and overview the techniques we develop to overcome them.

Challenge 1: Observation granularity. The way Xen scheduling works in our SMP setting makes spying on a victim VM challenging, particularly when one wants to use per-core microarchitectural features as a side channel. For example, the L1 caches contain the most potential for damaging side-channels [36], but these are not shared across different cores. An attacker must therefore try to arrange to frequently alternate execution on the same core with the victim so that it can measure side-effects of the victim's execution. This strategy has been shown to be successful in single-core, non-virtualized settings [8, 32, 43] by attackers that game OS process scheduling. But no gaming of VMM scheduling suitable for fine-grained side-channels has been reported, and the default scheduling regime in Xen would seem to bar frequent observations: the credit scheduler normally reschedules VMs every 30ms, while even a full 4096-bit modular exponentiation completes in about 200ms on a modern CPU core (on our local testbed, see Sec. 6). This leaves an attacker with the possibility of less than 10 side-channel observations of it.

In Sec. 4, we overcome this challenge to use the L1 instruction cache as a vector for side-channels. We demonstrate how to use interprocess interrupts (IPIs) to abuse the Xen credit scheduler in order to arrange for frequent interruptions of the victim's execution by a spy process running from within the attacker's VM. This takes advantage of an attacker having access to multiple VCPUs and allows the spy to time individual L1 cache sets. The scheduling nuances abused are a vulnerability in their own right, enabling degradation-of-service attacks and possibly cycle-stealing attacks [44, 50].

Challenge 2: Observation noise. Even with our IPI-based spying mechanism, there exists significant noise in the measured timings of the L1 instruction cache. Beyond the noise involved in any cache-based measurements, the VMM exacerbates noise since it also uses the L1 cache when performing scheduling. Manual analysis failed to provide simple threshold-based rules to classify cache timings as being indicative of particular victim operations.

In Sec. 5.1, we use a support vector machine (SVM) to relate L1 cache observations to particular operations of the victim. A critical challenge here is gathering accurate training data, which we accomplish via careful hand instrumentation of the target victim executable. Still, the SVM is error-prone, in the sense that it classifies a small fraction of code paths incorrectly. Fortunately, for many types of victim operations, the fine granularity achieved by the attack VM's IPI-based spying can yield multiple observations per individual operation. We use the redundancy in these observations together with knowledge of the set of possible victim code paths to correct SVM output errors by means of a hidden Markov model (HMM). This is detailed in Sec. 5.2. The SVM plus HMM combination, when correctly trained, can translate a sequence of observations into a sequence of inferred operations with few errors.

Challenge 3: Core migration. Our SMP setting has attacker and victim VCPUs float amongst the various PCPUs. The administrative Dom0 VM and any other VMs may also float amongst them. This gives rise to two hurdles. First, we must determine whether an observation is associated with the victim or some other, unrelated VCPU. Second, we will only be able to spy on the victim when assigned to the same PCPU, which may coincide with only some fraction of the victim's execution.

In Sec. 5.2, we describe how the HMM mentioned above can be modified to filter out sequences corresponding to unrelated observations. In Sec. 5.3, we provide a dynamic programming algorithm, like those in bioinformatics, to "stitch" together multiple inferred code-path fragments output by the SVM+HMM and thereby construct fuller hypothesized code-paths. By observing multiple executions of the victim, we can gather sufficiently many candidate sequences to, through majority voting, output a full code path with negligible errors.

Putting it all together. Our full attack pipeline is depicted in Fig. 1. The details of our measurement stage that addresses the first challenge described above are presented in Sec. 4. The analysis of these measurements to address the second and third challenges is then broken down into three phases: cache-pattern classification (Sec. 5.1), noise reduction (Sec. 5.2), and code-path reassembly (Sec. 5.3).

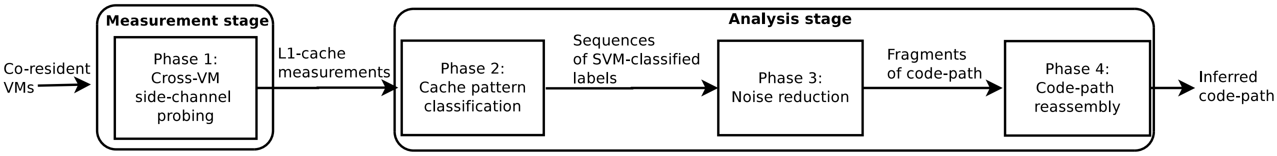

Co-resident VMs
Measurement stage
Phase 1: Cross-VM side-channel probing
L1-cache measurements
Analysis stage
Phase 2: Cache pattern classification
Sequences of SVM-classified labels
Phase 3: Noise reduction
Fragments of code-path
Phase 4: Code-path reassembly
Inferred code-path

Figure 1: Diagram of the main steps in our side-channel attack.

4. CROSS-VM SIDE CHANNELS

In this section, we demonstrate how an access-driven cross-VM side channel can be constructed on the L1 instruction cache in a modern x86 architecture running Xen.

4.1 Instruction Cache Spying

Caches are usually *set-associative*. A w-way set-associative cache is partitioned into m sets, each with w lines of size b. So, if the size of the cache is denoted by c, we have $m = c/(w \times b)$. For example, in the L1 instruction cache of an Intel Yorkfield processor as used in our lab testbed, $c = 32$KB, $w = 8$, $b = 64$B. Hence, $m = c/(w \times b) = 64$. Moreover, the number of cache-line-sized memory blocks in a 4KB memory page is 4KB$/b = 64$. Therefore, memory blocks with the same offset in a memory page will be mapped to the same cache set.

Probing the instruction cache. A basic technique for I-cache side-channels is timing how long it takes to read data from memory associated with individual cache sets, as described previously by Aciiçmez [1]. To do so, we first allocate sufficiently many contiguous memory pages so that their combined size is equal to the size of the I-cache. We then divide each memory page into 64 cache-line-sized blocks. The i^{th} data block in each page will map to the same cache set. To fill the cache set associated to offset i, then, it suffices to execute an instruction within the i^{th} block of each of the allocated pages. Filling a cache set is called PRIME-ing. We will also want to measure the time it takes to fill a cache set, this is called PROBE-ing. To PROBE the cache set associated with offset i, we executes the `rdtsc` instruction, then jumps to the first page's i^{th} block, which has instructions to jump to the i^{th} block of the next page, and so on. The final page jumps back to code that again executes `rdtsc` and calculates the elapsed time. This is repeated for each of the m cache sets to produce a vector of cache set timings.

The prime-probe protocol. A common method for conducting an access-driven cache attack is to PRIME and later PROBE the cache, a so-called PRIME-PROBE protocol, as introduced by Osvik et al. [33]. More specifically, a VCPU U of the attacker's VM spies on a victim's VCPU V by measuring the cache load in the L1 instruction cache in the following manner:

PRIME: U fills one or more cache sets by the method described above.

IDLE: U waits for a prespecified PRIME-PROBE *interval* while the cache is utilized by V.

PROBE: U times the duration to refill the same cache sets to learn V's cache activity on those sets.

Cache activity induced by V during U's PRIME-PROBE interval will evict U's instructions from the cache sets and replace them with V's. This will result in a noticeably higher timing measurement in U's PROBE phase than if there had been little activity from V. Of course, PROBE-ing also accomplishes PRIME-ing the cache sets (i.e., evicting all instructions other than U's), and so repeatedly PROBE-ing, with one PRIME-PROBE interval between each PROBE, eliminates the need to separately conduct a PRIME step.

4.2 Preempting the Victim

A fundamental difficulty in an I-cache attack without SMT support is for the attacker VCPU to regain control of the PCPU resource sufficiently frequently (i.e., after the desired PRIME-PROBE interval has passed). To accomplish this, we leverage the tendency of the Xen credit scheduler to give the highest run priority to a VCPU that receives an interrupt. That is, upon receiving an interrupt, the attacker VCPU will preempt another guest VCPU running on the PCPU, provided that it is not also running with that highest priority (as a compute-bound victim would not be). As such, our attack strategy is to deliver an interrupt to the attacker VCPU every PRIME-PROBE interval.

We consider three types of interrupts to "wake up" the attacking VCPU: timer interrupts, network interrupts and inter-processor interrupts (IPIs).[1] Timer interrupts in a guest OS can be configured to be raised with a frequency of at most 1000Hz, but this is not sufficiently granular for our attack targets (e.g., a cryptographic key). Network interrupts, as used in OS level CPU-cycle stealing attacks by Tsafrir et al. [44], can achieve higher resolution, but in our experiments the delivery times of network interrupts varied due to batching and network effects, rendering it hard to achieve microsecond-level granularity.

We therefore turn to IPIs. In SMP systems, an IPI allows one processor to interrupt another processor or even itself. It is usually issued through an advanced programmable interrupt controller (APIC) by one core and passed to other cores via either the system bus or the APIC bus. To leverage IPIs in our attack, another attacker VCPU, henceforth called the *IPI VCPU* , executes an endless loop that issues IPIs to the attacker VCPU which is conducting the PRIME-PROBE protocol, henceforth called the *probing VCPU* . This approach works generally well but is limited by two shortcomings.

First, due to interrupt virtualization by Xen, the PRIME-PROBE interval that can be supported through IPIs cannot be arbitrarily small. In our local testbed (see Sec. 6), we find it hard to achieve a PRIME-PROBE interval that is shorter than 50,000 PCPU cycles (roughly 16 microseconds). More frequent interrupts will be accumulated and delivered together. Second, if the IPI VCPU is descheduled then this can lead to periods during which no usable observations are made. If the Xen scheduler is non-work-conserving—meaning that a domain's execution time is limited to a budget, dictated by the *cap* and *weight* parameters assigned to

[1] A fourth option is high precision event timer interrupts as used by Bangerter et al. [8], but these are not available to guests in Xen.

it by Xen—then the IPI VCPU will be descheduled when it exceeds its budget. These periods then must be detected and any affected PRIME-PROBE instances discarded. However, if the scheduler is work-conserving (the default) and so allows a domain to exceed its budget if no other domain is occupying the PCPU, then descheduling is rare on a moderately loaded machine. It is worth noting that the probing VCPU executes so briefly that its execution appears not to be charged toward its budget by the current Xen scheduler.

4.3 Sources of Noise

In this section we discuss sources of noise in the side channel described above, and how we deal with each one.

4.3.1 Hardware Sources of Noise

TLB misses. In x86 processors, hardware TLBs are usually small set-associative caches that cache the translation from virtual addresses to physical addresses. A TLB miss will cause several memory fetches. Because the TLB is flushed at each context switch, the PROBE of the first cache set (see Sec. 4.1) will always involve TLB misses and so will be abnormally high; as such, the PROBE results for the first cache set will be discarded. However, in our approach, the number of memory pages used for the PRIME-PROBE protocol is small enough to avoid further TLB evictions.

Speculative execution. Modern superscalar processors usually fetch instructions in batches and execute them out-of-order. In order to force the in-order execution of our PROBE code for accurate measurement, the instructions need to be serialized using instructions like `cpuid` and `mfence`.

Power saving. The speed of a PROBE may be subject to change due to PCPU power saving modes. If the attacker VM is solely occupying a PCPU core, when it finishes its PROBE and relinquishes PCPU resources, the core may be slowed to save power. When the interrupt is delivered to the attacker VCPU, it appears to take longer for the PCPU to recover from the power saving mode and, in our experience, yields a much longer effective PRIME-PROBE interval. Thus, longer-then-expected PRIME-PROBE intervals may indicate there was no victim on the same core and so their results are discarded.

4.3.2 Software Sources of Noise

Context switches. A Xen hypervisor context switch will pollute the I-cache (though less than the D-cache in our experience). Moreover, noise due to a guest OS context switch in the attacker VM may introduce additional difficulties. Although there is not much we can do about the hypervisor context-switch noise, we minimize the OS context switch noise by modifying the core affinity of all the processes in the attacker VM so that all user-space processes are assigned to the IPI VCPU, minimizing context switches on the probing VCPU. This is beneficial also because it enables the probing VCPU to relinquish the PCPU as much as possible, allowing another VCPU (hopefully, the victim's) to share its PCPU.

Address space layout randomization. Address space layout randomization (ASLR) does not interfere with our attack (with 32KB L1 cache) because the L1 cache set to which memory is retrieved is determined purely by its offset in its memory page, and because ASLR in a Linux implementation aligns libraries to page boundaries.

Emulated RDTSC instruction. In the absence of an invariant timestamp counters [18], Xen 4.0 or later emulates the `rdtsc` call to prevent time from going backwards. In this case, the `rdtsc` call is about 15 or 20 times slower than the native call [27], which diminishes the attacker VM's ability to measure the duration to PROBE a cache set. As such, the attack we describe here works much more reliably when the `rdtsc` call is not emulated.

Interference from other domains. One important aspect of software noise in the cache-based side channel is interference from other domains besides the victim. The fact that the attacker VCPU may observe activities of Dom0 or other domains brings about one of a major difficulties in our study: how can an attacker VM distinguish victim activity of interest from cache activity from unrelated domains (or victim activity that is not of interest)?

We discuss our solution to this hurdle in detail in Sec. 5.2, though even with our solution described there, it is beneficial if we can minimize the frequency with which PROBE results reflect a VM other than the victim's. In the configurations we will consider in Sec. 6, if Dom0 is idle then the Xen scheduler will move the attacker and victim VCPU's to distinct cores most of the time. Thus, an effective strategy is to induce load on Dom0: if multiple victim VCPUs and the IPI VCPU are also busy and so together with Dom0 consume all four cores of the machine, then the probing VCPU, by relinquishing the PCPU frequently, invites another VCPU to share its PCPU with it. When the co-resident VCPU happens to be the victim's VCPU that is performing the target computation, then the PROBE results will be relevant to the attacker.

Since Dom0 is responsible for handling network packets, a general strategy to load Dom0 involves sending traffic at a reasonably high rate to an unopened port of the victim VM and/or attacker VM from a remote source. This can be especially effective since traffic filtering (e.g., via `iptables`) and shaping are commonly implemented in Dom0. In some cases (e.g., Amazon AWS), the attacker can even specify filtering rules to apply to traffic destined to his VM, and so he can utilize filtering rules and traffic that will together increase Dom0's CPU utilization.

5. CLASSIFYING CODE PATHS

In this section we introduce a set of techniques that, when combined, can enable an attacker VM to learn the code path used by a co-resident victim VM. In settings where control flow is dependent on confidential data, this enables exfiltration of secrets across VM boundaries. While the techniques are general, for concreteness we will use as a running example the context of cryptographic key extraction and, in particular, learning the code path taken when using the classic square-and-multiply algorithm. This algorithm (and generalizations thereof) have previously been exploited in access-driven attacks in non-virtualized settings (e.g., [1, 36]), but not in virtualized SMP systems as we explore here.

The square and multiply algorithm is depicted in Fig. 2. It efficiently computes the modular exponentiation $x^s \bmod N$ using the binary representation of e, i.e., $e = 2^{n-1}e_n + \cdots + 2^0e_1$. It is clear by observation that the sequence of function calls in a particular execution of SquareMult directly leaks e, which corresponds to the private key in many decryption or signing algorithms. We let M, S, and R stand for calls

```
SquareMult($x, e, N$):
  let $e_n, \ldots, e_1$ be the bits of $e$
  $y \leftarrow 1$
  for $i = n$ down to 1 {
    $y \leftarrow$ Square($y$)                 ($S$)
    $y \leftarrow$ ModReduce($y, N$)           ($R$)
    if $e_i = 1$ then {
      $y \leftarrow$ Mult($y, x$)              ($M$)
      $y \leftarrow$ ModReduce($y, N$)         ($R$)
    }
  }
  return $y$
```

Figure 2: The square-and-multiply algorithm.

to Mult, Square, and ModReduce, respectively, as labeled in Fig. 2. Thus, the sequence $SRMRSR$ corresponds to exponentiation by $e = 2$.

The techniques we detail in the next several sections show how an attacker can, despite VMM isolation, learn such sequences of operations.

5.1 Cache Pattern Classifier

Recall from Sec. 4 that the output of a single prime-probe instance is a vector of timings, one timing per cache set. The first step of our algorithm is to classify each such vector as indicating a multiplication (M), modular reduction (R) or squaring (S) operation. To do so in our experiments, we employ a multiclass support vector machine (SVM), specifically that implemented in `libsvm` [15]. An SVM is a supervised machine learning tool that, once trained, labels new instances as belonging to one of the classes on which it was trained. It also produces a *probability estimate* in $(0, 1]$ associated with its classification, with a number closer to 1 indicating a more confident classification.

To use an SVM to classify new instances, it is necessary to first train the SVM with a set of instance-label pairs. To do so, we use a machine with the same architecture as the machine on which the attack will be performed and configure it with the same hardware settings. We then install a similar software stack for which we have total control of the hypervisor. To collect our training data, we create a victim VM and attacker VM like those one would use during an attack. We use the `xm` command-line tools in Dom0 to pin the VCPUs of the victim VM and attacker's probing VCPU to the same PCPU. We then set the victim VM to repeatedly performing modular exponentiations with the same arguments and, in particular, with an exponent of all 1's, and the probing VCPU to repeatedly performing prime-probe instances.

This allows for the collection of vectors, one per prime-probe instance, but there remains the challenge of accurately labeling them as M, S or R. To do so, we need to establish communication from the victim VM to the attacker VM to inform the latter of the operation being performed (multiplication, squaring, or modular reduction) at any point in time. However, this communication should take as little time as possible, since if the communication takes too long, measurements collected from the prime-probe trials during training would differ from those in testing.

We employ cross-VM shared memory for this communication. Briefly, Xen permits different domains to establish memory pages shared between them (see [16, Ch. 4]). We utilize this shared memory by modifying the victim VM to write to the shared memory the type of operation (M, S, or R) immediately before performing it, and the attacker VM reads this value immediately after completing each Probe of the entire cache.

Another challenge arises, however, which is how to modify the victim VM's library that performs the exponentiation while keeping other parts of the library unchanged. Adding the shared-memory writes prior to compilation would change the layout of the binary and so would ruin our Probe results for the purpose of training. Instead, we prepare the instructions for shared-memory writing in a dynamic shared library called `libsync`. Then we instrument the binary of the exponentiation library to hook the Square, Mult, and ModReduce functions and redirect each to `libsync`, which simply updates the shared memory and jumps back to the Square, Mult or ModReduce function, as appropriate. Because the `libsync` and victim's library are compiled independently, the address space layout of the latter remains untouched. Even so, the memory-writing instructions slightly pollute the instruction cache, and so we exclude the cache sets used by these instructions (three sets out of 64).

5.2 Noise Reduction

There are two key sources of noise that we need to address at this phase of the key extraction process. The first is the classification errors of the SVM. Noise arising from events such as random fluctuations in probe timings causes the majority of SVM classification errors. Incomplete prime-probe overlap with exponentiation operations occasionally creates ambiguous cache observations, for which no classification is strictly correct. The second is the presence of prime-probe results and, in turn, SVM outputs that simply encode no information of interest to the attacker for other reasons, e.g., because the victim VCPU had migrated to a different PCPU from the attacker or because it was performing an operation not of interest to the attacker. Consequently, we develop a sequence of mechanisms to refine SVM outputs to reduce these sources of noise.

Hidden Markov Model. We start by employing a hidden Markov model (HMM) to eliminate many of the errors in SVM output sequences. (We assume reader familiarity with HMM basics. For an overview, see, e.g., [12].) Our HMM models the victim's exponentiation as a Markov process involving transitions among hidden "square," "multiply," and "reduce" states, respectively representing Square, Mult, and ModReduce operations. As exponentiation is executed by the victim, labels output by the attacker's SVM give a probabilistic indication of the victim's hidden state.

As the SVM outputs multiple labels per Square, Mult, or ModReduce operation, we represent an operation as a *chain* of hidden states in the HMM. As is the case with any error-correcting code, the redundancy of SVM output labels helps rectify errors. Intuitively, given multiple labels per operation (e.g., induction of correct sequence $SSSSS$ by a Square operation), the HMM can correct occasional mislabelings (e.g., the outlying M in $SSSMS$). The HMM also corrects errors based on structural information, e.g., the fact that a square or multiply is always followed by a modular reduction.

While the HMM takes as input a sequence of SVM labels, each such label, S, M, or R, is first mapped into an expanded label set that reflects its corresponding level of SVM confidence. Given a "high" SVM confidence, in the

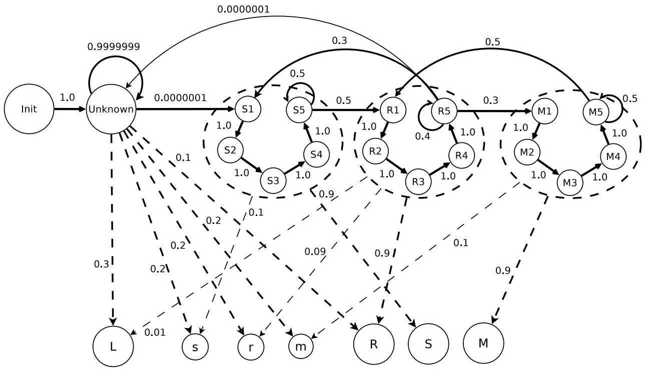


Figure 3: Diagram of the HMM used in our experiments with 4096-bit base x and modulus N. Emission labels are depicted in the lower half, hidden states in the upper half. Solid arrows indicate transitions, dotted arrows denote emissions. Emission probabilities below 0.01 are omitted.

range $[0.8, 1.0]$, a label remains unchanged. For "medium" confidence, lying within $[0.6, 0.8)$, a label is transformed into a different label indicating this medium confidence; i.e., S, M, and R are mapped respectively to new labels s, m, and r. Finally, given a "low" confidence, in $[0, 0.6)$, any of S, M, or R is mapped to a generic, "low confidence" label L.

In brief, then, the SVM output label set $\{S, M, R\}$ is expanded, through coarse integration of SVM confidence measures, into a set of seven labels $\{S, M, R, s, m, r, L\}$. This expanded set constitutes the emission labels of the HMM. We found that hand-tuning the transition and emission probabilities in the HMM resulted in better performance, i.e., fewer errors in HMM decoding, than training via the Baum-Welch algorithm [10]. The full HMM, with hidden states, emission labels, and transition and emission probabilities, is depicted in Fig. 3.

Given this HMM and a sequence of expanded-label SVM emissions, we run a standard Viterbi algorithm [45] to determine the maximum likelihood sequence of corresponding hidden states. The result is a sequence of labels from the hidden-state set $\{S1, \ldots, S5, M1, \ldots, M5, R1, \ldots, R5, Unknown\}$. We refer to this as the HMM output sequence.

Post-processing HMM Outputs. We post-process HMM output sequences to remove state labels that are redundant or agnostic to key-bit values. The states in an HMM operator chain (e.g., $S1 \ldots S4\ S5^+$, where "$S5^+$" denotes one or more occurrences of $S5$) collectively indicate only a single operation (e.g., one instance of Square). Thus, our post-processing step replaces every chain of the form $S1 \ldots S4\ S5^+$ with a single S and every chain of the form $M1 \ldots M4\ M5^+$ with a single M. *Unknown* states carry no information about key bit values and so are discarded.

We found it necessary to post-process $R1 \ldots R4\ R5^+$ chains in a somewhat more refined manner. Recall that ModReduce operations are key-agnostic, and so we discarded such chains, provided that they were short. The HMM output sequence, however, would sometimes include long chains of the form $R1 \ldots R4\ R5^+$, which would typically signal a cryptographic observation that passed unobserved. Any such chain of sufficient length was thus replaced in post-processing with a $*$, indicating a hypothesized omitted Mult or Square. In our experiments described in Sec. 6, we did so for chains of the form $R1 \ldots R4\ R5^+$ of length 24 or more. Chains of length less than this were discarded.

Consequently, post-processing yields refined HMM outputs over the label set $\{S, M, *\}$.

Filtering out non-cryptographic HMM outputs. Successful key reconstruction requires that we reliably identify and retain observed sequences of cryptographic operations, i.e., those involving private key material, while discarding non-cryptographic sequences. Extraneous code paths arise when the victim migrates to a core away from the attacker or executes software independent of the cryptographic key.

The SVM in our attack does not include a label for extraneous code paths. (Non-cryptographic code constitutes too broad a class for effective SVM training.) Long non-cryptographic code paths, however, are readily distinguishable from cryptographic code paths based on the following observation. A random private key—or key subsequence—includes an equal number of 0 and 1 bits in expectation. As a 1 induces a square-and-multiply, while a 0 induces a squaring only, the expected ratio of corresponding S to M labels output by the SVM, and thus the HMM, is 2:1. Thus, a reasonably long key subsequence will evidence approximately this 2:1 ratio with high probability. In contrast, a non-cryptographic code path tends to yield many *Unknown* states and therefore outputs sequences that are generally discarded, or in rare cases yields short sequences with highly skewed S-to-M ratio.

The following elementary threshold classifier for cryptographic versus non-cryptographic post-processed HMM output sequences proves highly accurate. Within a given HMM output sequence, we identify all subsequences of S and M labels of length at least α, for parameter α. (In other words, we disregard short subsequences, which tend to be spurious and erroneously skew S-to-M ratios.) We count the total number a of S labels and b of M labels across all of these subsequences, and let $a/(b + 1)$ represent the total S-to-M ratio. (Here "+1" ensures a finite ratio.) If this S-to-M ratio falls within a predefined range $[\rho_1, \rho_2]$, for parameters ρ_1 and ρ_2, with $0 < \rho_1 < 2 < \rho_2$, the output sequence is classified as a cryptographic observation. Otherwise, it is classified as non-cryptographic.

We found that we could improve our detection and filtering of inaccurate cryptographic sequences by additionally applying a second, simple classifier. This classifier counts the number of MM label pairs in an HMM output sequence. As square-and-multiply exponentiation never involves two sequential multiply operations—there is always an interleaved squaring—such MM pairs indicate a probable erroneous sequence. Thus, if the number of MM pairs exceeds a parameter β, we classify the output sequence as inaccurate and discard it.

We applied these two classifiers (S-to-M ratio and MM-pair) to all HMM output sequences, and discarded those classified as non-cryptographic. The result is a set of post-processed, filtered HMM outputs, of which an overwhelming majority represented observed cryptographic operations, and whose constituent labels were largely correct.

5.3 Code-path reassembly

Recall that a major technical challenge in our setting is the fact that the victim VM's VCPUs float across physical cores. This movement frequently interrupts attacker VM

PRIME-PROBE attempts, and truncates corresponding HMM output sequences. It is thus helpful to refer to the post-processed, filtered HMM outputs as *fragments*.

Fragments are short, more-or-less randomly positioned subsequences of hypothesized labels for the target key operations. Despite the error-correcting steps detailed above, fragments also still contain a small number of erroneous S and M labels as well as $*$ labels. The error-correcting steps detailed in Sec. 5.2 operate *within* fragments. In the final, sequence-reconstruction process described here, we correct errors by comparing labels *across* fragments, and also "stitch" fragments together to achieve almost complete code-path recovery. This will reveal most of the key sequence; simple brute forcing of the remaining bits reveals the rest.

Accurate sequence alignment and assembly of fragments into a full key-spanning label sequence is similar to the well-known sequence-reconstruction problem in bioinformatics. There are many existing tools for DNA sequencing and similar tasks, e.g., [7, 17]. However, various differences between that setting and ours, in error rates, fragment lengths, etc., have rendered these tools less helpful than we initially hoped, at least so far. We therefore developed our own techniques, and leave improving them to future work.

In this final, sequence-reconstruction step of key recovery, we partition fragments into batches. The number of batches ζ and number of fragments θ per batch, and thus the total number $\zeta\theta$ of fragments that must be harvested by the attacker VM, are parameters adjusted according to the key-recovery environment. It is convenient, for the final stage of processing ("Combining spanning sequences," see below) to choose ζ to be a power of three.

Our final processing step here involves three stages: inter-fragment error correction, fragment stitching to generate sequences that span most of the code-path, and then a method for combining spanning sequences to provide an inferred code-path. The first two stages operate on individual batches of fragments, as follows:

Cross-fragment error-correction. In this stage, we correct errors by comparing labels across triples of fragments.

First, each distinct pair of fragments is aligned using a variant of the well-known dynamic programming (DP) algorithm for sequence alignment [31]. We customize the algorithm for our setting in the following ways. First, we permit a $*$ label to match either a S or an M. Second, because two fragments may reflect different, potentially non-intersecting portions of the key, terminal gaps (i.e., inserted before or after a fragment) are not penalized. Third, a contiguous sequence of nonterminal gaps is penalized quadratically as a function of its length, and a contiguous sequence of matches is rewarded quadratically as a function of its length.

We then construct a graph $G = (V, E)$ in which each fragment is represented by a node in V. An edge is included between two fragments if, after alignment, the number of label matches exceeds an empirically chosen threshold γ. Of interest in this graph are *triangles*, i.e., cliques of size three. A triangle (v_1, v_2, v_3) corresponds to three mutually overlapping fragments / nodes, v_1, v_2, and v_3, and is useful for two purposes.

First, a triangle permits cross-validation of pairwise alignments, many of which are spurious. Specifically, let k_{12} be the first position of v_1 to which a (non-gap) label of v_2 aligned; note that k_{12} could be negative if the first label of v_2 aligned with an initial terminal gap of v_1, and similarly for k_{13} and k_{23}. If $|(k_{13} - k_{12}) - k_{23}| \leq \tau$, where τ is an algorithmic parameter (5 in our experiments), then the alignments are considered mutually consistent. (Intuitively, $k_{13} - k_{12}$ is a measure of alignment between v_2 and v_3 with respect to v_1, while k_{23} measures direct alignment between v_2 and v_3. Given perfect alignment, the two are equal.) Then, the triangle is *tagged* with the "length" of the region of intersection among v_1, v_2 and v_3

The second function of triangles is error-correction. Each triangle (v_1, v_2, v_3) of G is processed in the following way, in descending order. Each position in the region of intersection of v_1, v_2 and v_3 has three corresponding labels (or gaps), one for each fragment. If two are the same non-gap label, then that label is mapped onto all three fragments in that position. In other words, the three fragments are corrected over their region of intersection according to majority decoding over labels.

Cross-fragment error-correction changes neither the length nor number of fragments in the batch. It merely reduces the global error rate of fragment labels. We observe that if the mean error rate of fragments, in the sense of edit distance from ground truth, is in the vicinity of 2% at this stage, then the remaining processing results in successful key recovery. We aim at this mean error rate in parameterizing batch sizes (θ) for a given attack environment.

Fragment stitching. In this next processing stage, a batch of fragments is assembled into what we call a *spanning sequence*, a long sequence of hypothesized cryptographic operations. In most cases, the maximum-length spanning sequence for a batch covers the full target key.

The DP algorithm is again applied in this stage to every pair of fragments in a batch, but now customized differently. First, terminal gaps are still not penalized, though a contiguous sequence of matches or (nonterminal) gaps accumulates rewards or penalties, respectively, only linearly as a function of its length. This is done since the fragments are presumably far more correct now, and so rewarding sequences of matches superlinearly might overwhelm any gap penalties. Second, the penalty for each nonterminal gap is set to be very high relative to the reward for a match, so as to prevent gaps unless absolutely necessary.

Following these alignments, a directed graph $G' = (V', E')$ is constructed in which each node in V' (as in V above) represents a fragment. An edge (v_1, v_2) is inserted into E' for every pair of fragments v_1 and v_2 with an alignment in which the first label in v_2 is aligned with some label in v_1 after the first. (Intuitively, v_2 overlaps with and sits to the "right" of v_1.) Assuming, as observed consistently in our experiments, that there are no alignment errors in this process, the resulting graph G' will be a directed *acyclic* graph (DAG).[2]

A path of fragments / nodes $v_1, v_2, \ldots, v_m \in V'$ in this graph is stitched together as follows. We start with a source node v_1 and append to it the non-overlapping sequence of labels in v_2, i.e., all of the labels of v_2 aligned with the ending terminal gaps of v_1, if any. (Intuitively, any labels in v_2 positioned to the "right" of v_1 are appended to v_1.) We build up a label sequence in this way across the entire path. The resulting sequence of labels constitutes a spanning sequence. We employ a basic greedy algorithm to identify

[2] A cycle in this graph indicates the need to adjust parameters in previous stages and retry.

the path in G' that induces the maximum-length spanning sequence.

Combining spanning sequences. The previous stages, applied per batch, produce ζ spanning sequences. The ζ spanning sequences emerging from the fragment stitching stage are of nearly, but not exactly, equal length, and contain some errors. For the final key-recovery stage, we implement an alignment and error-correction algorithm that proceeds in rounds. In each round, the sequences produced from the previous round are arbitrarily divided into triples, and each triple is reduced to a single spanning sequence that is carried forward to the next round (and the three used to create it are not). For this reason, we choose ζ, the number of batches, to be a power of three, and so we iterate the triple-merging algorithm $\log_3 \zeta$ times. The result is a sequence of hypothesized cryptographic operations covering enough of the target key to enable exhaustive search over all possibilities for any remaining $*$ values.

Each round proceeds as follows. Each triple (s_1, s_2, s_3) is first aligned using a basic three-way generalization of DP (e.g., see [23, Section 4.1.4]). This may insert gaps (rarely consecutively) into the sequences, yielding new sequences (s'_1, s'_2, s'_3). Below we denote a gap so inserted by the "label" $\sqcup$. The algorithm is parameterized to prevent alignment of S and M labels in the same position within different spanning sequences.

To the resulting aligned sequence triple (s'_1, s'_2, s'_3), the length of which is denoted as ℓ, is applied a modified majority-decoding algorithm that condenses the triple into a single, merged output sequence. We say that s'_1, s'_2, and s'_3 *strongly agree* at position j if all three sequences have identical labels at position j or, for $1 < j < \ell$, if any two of the three have identical labels at each of positions $j-1$, j, and $j+1$. In this step, the output sequence adopts a label at position j if the three strongly agree on that label at position j and the label is S, M, or, in the last round, $\sqcup$. Otherwise, the output sequence adopts $*$ at position j. At the end of the last round, any residual $\sqcup$ labels are removed.

6. EVALUATION

We performed a case study using the `libgcrypt` v.1.5.0 cryptographic library (see `http://www.gnu.org/software/libgcrypt/`). This is the most recent version of `libgcrypt`; our results extend to cover earlier versions as well. To be concrete, we also fixed an application that uses the library: Gnu Privacy Guard (`GnuPG`) v.2.0.19 (`http://www.gnupg.org/`). `GnuPG` is used widely for encrypting and signing email, but we note that `libgcrypt` use goes beyond just `GnuPG`. The attack should extend to any application using the vulnerable routines from `libgcrypt`.

ElGamal encryption. Manual code review revealed that `libgcrypt` employs a more-or-less textbook variant of the square-and-multiply modular exponentiation algorithm for use with cryptosystems such as RSA [40] and ElGamal [19]. Our case study focuses on the latter.

ElGamal encryption in `libgcrypt` uses a cyclic group $\mathbb{Z}_p^*$ for prime p and generator g. The bit length size of p is dictated by a user-specified security parameter κ. Given g and p, a secret key is chosen uniformly at random to be a non-negative integer x whose bit length is, for example, 337, 403, or 457 when κ is 2048, 3072, or 4096, respectively. We note that this deviates from standard ElGamal, in which one would have $|x| \approx |p|$. The smaller exponent makes decryption faster. The public key is set to be $X = g^x \bmod p$.

To encrypt a message $M \in \mathbb{Z}_p^*$, a new value $r \in \mathbb{Z}_m$, for $m = 2^{|x|}$, is chosen at random; the resulting ciphertext is $(g^r, X^r \cdot M)$. (Typically M is a key for a separate symmetric encryption mechanism.) Decryption of a ciphertext (R, Y) is performed by computing $R^x \bmod p$, inverting it modulo p, and then multiplying Y by the result modulo p.

Our attack abuses the fact that computation of $R^x \bmod p$ during decryption is performed using the square-and-multiply modular exponentiation algorithm. The pseudocode of Fig. 2 is a close proxy of the code used by `libgcrypt`.

6.1 With a Work-Conserving Scheduler

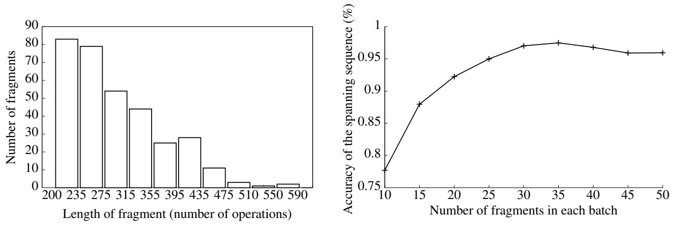
Experiment settings. We evaluated our attack in a setting in which two paravirtualized guest VMs, each of which possesses two VCPUs, co-reside on a single-socket quad-core processor, specifically an Intel Core 2 Q9650 with an operating frequency of 3.0GHz. The two guest VMs and Dom0 were each given *weight* 256 and *cap* 0; in particular, this configuration is work-conserving, i.e., it allows any of them to continue utilizing a PCPU provided that no other domain needs it. Dom0 was given a single VCPU. One guest VM acted as the victim and the other as the attacker. We ran Xen 4.0 as the virtualization substrate, with `rdtsc` emulation disabled. Both VMs ran an Ubuntu 10.04 server with a Linux kernel 2.6.32.16. The size of the memory in the guest VMs was large enough to avoid frequent page swapping and so was irrelevant to the experiments. The victim VM ran `GnuPG` v.2.0.19 with `libgcrypt` version v.1.5.0, the latest versions as of this writing. The victim's ElGamal private key was generated with security parameter $\kappa = 4096$. Other parameters for our attack are shown in Fig. 4.

In general, the attacker can either passively wait for periods where it shares a PCPU with the victim, or can actively "create" more frequent and longer such periods on purpose. To abbreviate our experiment, we assumed a situation that is to the attacker's advantage (but is nevertheless realistic), in which both Dom0 and one victim VCPU are CPU-bound, running non-cryptographic computational tasks. These conditions maximized the frequency with which the attacker and the *other* victim VCPU share a PCPU. As discussed in Sec. 4.3.2, Dom0 can be loaded by, for example, forcing it to analyze a high rate of traffic with expensive filtering rules. We experimented with several such scenarios (varying in the number of rules and packet rates), as well as other situations that would encourage the attacker and victim to share a PCPU (e.g., dedicating one core to Dom0, which "might be a good idea for systems running I/O intensive guests" [47]), many of which gave results similar to those reported here.

Parameter	Sec. 6.1	Sec. 6.2
$[\rho_1, \rho_2]$	$[1, 4]$	$[1, 4]$
α	200	100
β	5	5
ζ	9	9
θ	30	35
τ	5	5
γ	100	50

Figure 4: Parameter settings for attacks

Another way in which we were generous to the attacker in this demonstration was that we assumed the victim VM would often perform ElGamal decryption with the target



(a) (b)

Figure 6: Results with work-conserving scheduler (Sec. 6.1). (a) Frequency of fragment lengths extracted, each bar represents the number of fragments whose length falls between the x-axis labels. (b) Accuracy of spanning sequences as a function of number of fragments in a batch.

key—e.g., because the attacker had the ability to remotely trigger decryption, as might be realistic for a network service that the attacker could invoke—and that this decryption executed on the victim VCPU that was not already compute-bound. As such, in our demonstration, the attacker did not have to wait indefinitely for a private-key decryption, but rather the victim performed decryptions over and over. Given our ability to filter non-cryptographic observations, less frequent exponentiations would slow down, but not prevent, the attack. In fact, it is worth noting that exponentiation with the private exponent under attack constitutes only roughly 2% of the execution time of a private-key decryption, and so even in this demonstration, 98% of victim execution was irrelevant to our attack and filtered out by our techniques.

Experiment results. Our SVM was trained as per the procedure discussed in Sec. 5.1 with PRIME-PROBE results from 30,000 Square operations, 30,000 Mult operations, and 80,000 ModReduce operations. We skewed the training data toward ModReduce operations to minimize ModReduce operations being misclassified as Square or Mult operations. A three-fold cross validation resulted in the confusion matrix shown in Fig. 5.

In the attack, we performed 300,000,000 PRIME-PROBE trials in chunks of 100,000. The data collection lasted about six hours, during which roughly 1000 key-related fragments were recovered from our HMM (Sec. 5.2).

	S	M	R
S	.91	.00	.09
M	.01	.92	.07
R	.02	.01	.97

Figure 5: Confusion matrix for SVM

Of these, 330 key-related fragments had length at least α. The lengths of these fragments are shown in Fig. 6(a). Let the *accuracy* of a fragment be defined as 1 minus the normalized edit distance of the fragment from ground truth, i.e., the edit distance divided by the length of the fragment. On average, these fragments had 0.958 accuracy with a standard deviation of 0.0164.

We then combined fragments (Sec. 5.3) to produce spanning sequences. The accuracy of these spanning sequences was a function of the number of fragments in each batch, as shown in Fig. 6(b). We chose to use batches of 30 fragments each, yielding an average spanning sequence accuracy of 0.981.

The final step was to combine the spanning sequences (end of Sec. 5.3). The resulting key had no erasures, insertions or replacements. The only uncertainties arose from $*$ labels at the two ends or occasionally in the middle, each representing "no-op" (a spurious operation) or a single Square or Mult operation. This left us needing to perform a brute-force search for the uncertain bits, but the search space was only 9,862 keys.[3]

6.2 With a Non-Work-Conserving Scheduler

We also evaluated the attack for a non-work-conserving setting of the Xen scheduler which is configured as *weight* = 256 and *cap* = 80 (and other parameters as shown in Fig. 4). The induced workload in Dom0 and the victim remains the same as in the previous section. Recall from Sec. 4 that this is a more difficult case for our attack, since it causes the IPI VCPU to be descheduled more aggressively, which in turn interferes with initiating an IPI to the probing VCPU. When this occurs, the corresponding PRIME-PROBE result must be discarded. Therefore, if the scheduler is non-work-conserving, data collection takes longer and the fragments resulting from our HMM tend to be shorter.

These effects are demonstrated in Fig. 7, which shows the fragment lengths at the same stage of processing as is reflected in Fig. 6(a) for the work-conserving case. Despite the fact that these fragments are based on 1,900,000,000 PRIME-PROBE trials (collected during about 45 hours), over six times the number we collected in Sec. 6.1, only 322 fragments of length at least α resulted—an order of magnitude less than the work-conserving case. And this occurred despite the fact that we set α to only 100 in the non-work-conserving case, i.e., half of the value in Sec. 6.1. These 322 fragments yielded 9 spanning sequences with average accuracy 0.98, which were "stitched" together into a single key with only a few missing bits, yielding a search space of only 6615 keys.

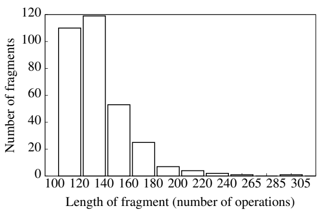


Figure 7: Fragment lengths, non-work-conserving scheduler (Sec. 6.2)

7. COUNTERMEASURES

There are multiple avenues for possible defenses against cross-VM side-channels, whose benefits and downsides we discuss here.

Avoiding co-residency. In high-security environments, a longstanding practice is to simply not use the same computer to execute tasks that must be isolated from each other, i.e., to maintain an "air gap" between the tasks. This remains the most high-assurance defense against side-channel (and many other) attacks. But this would obviate many of the

[3] Rather than searching over all three possible operation assignments to each $*$ symbol, we prune the search space by grouping assignments into functional equivalence classes; e.g., (Square, no-op) is equivalent to (no-op, Square).

current and future uses of VMs, including public clouds that multiplex physical servers such as Amazon EC2, Windows Azure, and Rackspace, and the other VM-powered applications discussed in the introduction.

Side-channel resistant algorithms. There exists a long line of work on cryptographic algorithms designed to be side-channel resistant (e.g., [11, 33, 35, 35, 36]). Recent versions of some cryptographic libraries attempt to prevent the most egregious side-channels; e.g., one can use the Montgomery ladder algorithm [30] for exponentiation or even a branchless algorithm. But these algorithms are slower than leakier ones, legacy code is still in wide use (as exhibited by the case of `libgcrypt`), and proving that implementations are side-channel free remains beyond the scope of modern techniques. Moreover, our techniques are applicable to non-cryptographic settings where there are few existing mechanisms for preventing side-channels.

Core scheduling. Another defense might seek to modify scheduling to at least limit the granularity of interrupt-based side-channels. The current Xen credit scheduler optimizes low latency at the cost of allowing frequent interrupts, even by non-malicious programs. Future Xen releases [42] already have plans to modify the way interrupts are handled, allowing a VCPU to preempt another VCPU only when the latter has been running for a certain amount of time (default being 1ms). This will reduce our side-channel's measurement granularity, but not eliminate the side-channel. Coarser side channels may already prove damaging [39]. A fundamental question for future work, therefore, is what interruption granularity best balances performance and security.

8. CONCLUSION

The use of virtualization to isolate a computation from malicious ones that co-reside with it is growing increasingly pervasive. This trend has been facilitated by the failure of today's operating systems to provide adequate isolation, the emergence of commodity VMMs offering good performance (e.g., VMWare, Xen, HyperV), and the growth of cloud facilities (e.g., EC2, Rackspace) that leverage virtualization to enable customers to provision computations and services flexibly. Given the widespread adoption of virtualization, it is thus critical that its isolation properties be explored and understood.

In this paper, we have shed light on the isolation properties (or lack thereof) of a leading VMM (Xen) in SMP environments, by demonstrating that side-channel attacks with fidelity sufficient to exfiltrate a cryptographic key from a victim VM can be mounted. Ours is the first demonstration of such a side-channel in a virtualized SMP environment. Challenges that our attack overcomes include: preempting the victim VM with sufficient frequency to enable fine-grained monitoring of its I-cache activity; filtering out numerous sources of noise in the I-cache arising from both hardware and software effects; and core migration that renders many attacker observations irrelevant to the task of extracting the victim's key. Through a novel combination of low-level systems implementation and sophisticated tools such as classifiers (e.g., SVMs and HMMs) and sequence alignment algorithms, we assembled an attack that was sufficiently powerful to extract ElGamal decryption keys from a victim VM in our lab tests.

Acknowledgments

We are grateful to Victor Heorhiadi for his help with experiment setup and for his comments on drafts of this paper, and to Jan Prins for helpful discussions on sequence-reconstruction algorithms. This work was supported in part by NSF grants 0910483 and 1065134, the Science of Security Lablet, and a grant from VMWare.

9. REFERENCES

[1] O. Aciiçmez. Yet another microarchitectural attack: Exploiting I-cache. In *ACM Workshop on Computer Security Architecture*, pages 11–18, October 2007.

[2] O. Aciiçmez, B. B. Brumley, and P. Grabher. New results on instruction cache attacks. In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop*, pages 110–124, August 2010.

[3] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert. On the power of simple branch prediction analysis. In *2nd ACM Symposium on Information, Computer and Communications Security*, pages 312–320, March 2007.

[4] O. Aciiçmez, W. Schindler, and Ç. K. Koç. Cache based remote timing attack on the AES. In *Topics in Cryptology – CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, pages 271–286, February 2007.

[5] O. Aciiçmez and J.-P. Seifert. Cheap hardware parallelism implies cheap security. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 80–91, September 2007.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[7] Celera Assembler. `http://wgs-assembler.sourceforge.net/`.

[8] E. Bangerter, D. Gullasch, and S. Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *32nd IEEE Symposium on Security and Privacy*, 2011.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[10] L. E. Baum, T. Petrie, G. Soules, and N. Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *The Annals of Mathematical Statistics*, 41(1):164–171, 1970.

[11] D. J. Bernstein. Cache-timing attacks on AES, 2005.

[12] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, October 2007.

[13] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[14] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. Openpgp message format. Technical report, RFC 2440, November, 1998.

[15] C. C. Chang and C. J. Lin. LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

[16] D. Chisnall. *The Definitive Guide to the Xen Hypervisor (Prentice Hall Open Source Software Development Series)*. Prentice Hall PTR, November 2007.

[17] ClustalW2. http://www.clustal.org/clustal2/.

[18] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual, vol 1-3. http://www.intel.com/products/processor/manuals/.

[19] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT-31(4), July 1985.

[20] P. England and J. Manferdelli. Virtual machines for enterprise desktop security. *Information Security Technical Report*, 11(4):193 – 202, 2006.

[21] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems – CHES 2001*, volume 2162 of *LNCS*, pages 251–261, May 2001.

[22] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *ACM Symposium on Operating Systems Principles*, pages 193–206. ACM, 2003.

[23] N. Gautham. *Bioinformatics: Databases and Algorithms*. Alpha Science International Ltd., 2006.

[24] Gnu Privacy Guard. www.gnupg.org, 2012.

[25] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology – CRYPTO '99*, volume 1666 of *LNCS*, pages 388–397, August 1999.

[26] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology – Crypto'96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, 1996.

[27] D. Magenheimer. TSC mode HowTo. Available: http://mirror.choon.net/xen/xen-unstable.hg/docs/misc/tscmode.txt.

[28] Andrew Marshall, Michael Howard, Grant Bugher, and Brian Harden. Security best practices for developing windows azure applications, June 2010.

[29] R. Meushaw and D. Simard. A network on a desktop. *NSA Tech Trend Notes*, 9(4), 2000. http://www.vmware.com/pdf/TechTrendNotes.pdf.

[30] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp*, 48(177):243–264, January 1987.

[31] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, March 1970.

[32] M. Neve and J.-P. Seifert. Advances on access-driven cache attacks on AES. In *Selected Areas in Cryptography, 13th International Workshop, SAC 2006*, pages 147–162, August 2006.

[33] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology – CT-RSA 2006*, pages 1–20. Springer-Verlag, 2005.

[34] R. Owens and W. Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *IEEE International Performance Computing and Communications Conference*, 2011.

[35] D. Page. Partitioned cache architecture as a side-channel defence mechanism, 2005.

[36] C. Percival. Cache missing for fun and profit. In *BSDCon 2005*, 2005.

[37] M. Piotrowski and A. D. Joseph. Virtics: A system for privilege separation of legacy desktop applications. Technical Report EECS-2010-70, U.C. Berkeley, 2010.

[38] J.-J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001*, volume 2140 of *LNCS*, pages 200–210, September 2001.

[39] 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 *16th ACM Conference on Computer and Communications Security*, pages 199–212, 2009.

[40] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2), February 1978.

[41] J. Rutkowska and R. Wojtczuk. Qubes OS architecture. http://qubes-os.org, 2012.

[42] Xen 4.2: New scheduler parameters. http://blog.xen.org/index.php/2012/04/10/xen-4-2-new-scheduler-parameters-2/.

[43] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[44] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Secretly monopolizing the CPU without superuser privileges. In *16th USENIX Security Symposium*, pages 1–18, 2007.

[45] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Inform. Theory*, IT-13:260–269, April 1967.

[46] M. Weiß, B. Heinz, and F. Stumpf. A cache timing attack on AES in virtualization environments. In *16th International Conference on Financial Cryptography and Data Security*, February 2012.

[47] Can I dedicate a cpu core (or cores) only for dom0? http://wiki.xen.org/wiki/XenCommonProblems#Can_I_dedicate_a_cpu_core_.28or_cores.29_only_for_dom0.3F.

[48] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *ACM Cloud Computing Security Workshop*, pages 29–40, 2011.

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

[50] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram. Scheduler vulnerabilities and coordinated attacks in cloud computing. In *IEEE International Symposium on Networking Computing and Applications*, 2011.