Dogo Rangsang Research Journal
ISSN : 2347-7180
UGC Care Group I Journal
Vol-08 Issue-14 No. 04, April 2021

Attacks on cross-tenant spatial multiplexing in PaaS clouds

Ms.ANANYA PREETI PADMA*, Dr. RAMAKANTA BHOI
Dept. OF Computer Science and Engineering, NIT , BBSR
ananyapriti@thenalanda.com*,ramakantabhoi@thenalanda.com

ABSTRACT

In commercial Platform-as-a-Service (PaaS) clouds, we demonstrate our new attack methodology for carrying out cache-based side-channel assaults amongst tenants. Our system extends this work by utilising the Gullasch et al. FLUSH- RELOAD attack as a primitive within an automaton-driven technique for tracing a victim's execution. We use our architecture to confirm tenant co-location before extracting information across tenant boundaries. Designers particularly show how to conduct attacks to steal user credentials, break SAML single sign-on, and acquire potentially sensitive application data (such as the quantity of products in a shopping cart). As far as we are aware, our attacks are the first granular, cross-tenant, side-channel attacks that have been successfully tested on modern commercial clouds, whether PaaS-based or not.

Categories and Subject Descriptors

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

General Terms

Security

Keywords

Cloud security; side-channel attacks; cache-based side channels; Platform-as-a-Service

1. INTRODUCTION

Public Platform-as-a-Service (PaaS) clouds are an important segment of the cloud market, being projected for compound annual growth of almost 30% through 2017 [20] and "on track to emerge as the key enabling technology for innovation inside and outside enterprise IT" [23]. For our purposes here, a PaaS cloud permits tenants to deploy tasks in the form of interpreted source (e.g., PHP, Ruby, Node.js, Java) or application executables that are then executed in a provider-managed host OS shared with other customers' applications. As such, a PaaS cloud often leverages OS-based techniques such as Linux containers to isolate tenants, in contrast to hypervisor-based techniques common in Infrastructure-as-a-Service (IaaS) clouds.

A continuing, if thus far largely hypothetical, threat to cloud tenant security is failures of isolation due to side-channel information leakage. A small but growing handful of works have explored side channels in settings characteristic of IaaS clouds, to which tenants deploy tasks in the form of virtual machines (VMs). Demonstrated attacks include side channels by which an attacker VM can extract coarse load measurements of a victim VM with which it is co-located [32]; identify pages it shares with a co-located victim VM, allowing it to detect victim VM applications, downloaded files [33] and its operating system (OS) [29]; and even exfiltrate a victim VM's private decryption key [40]. However, only the first of these attacks was demonstrated on a public cloud, with the others being demonstrated in lab settings. To the best of our knowledge, no side-channel attack capable of extracting granular information from a victim has been demonstrated in the wild.

In this paper, we initiate the study of cross-tenant side-channel attacks specifically in PaaS clouds and, in doing so, provide the first demonstration of granular, cross-tenant side channels in commercial clouds of any sort. Existing side-channel attacks mountable by one process on another running on the same OS, particularly those that leverage processor caches (e.g., [28, 30, 25, 34, 14, 38]), seem well suited to performing attacks across boundries between tenant instances[1] in PaaS deployments. This is largely true in our experience, though directly leveraging these attacks in PaaS settings is not as straightforward as one might think. One reason is that even identifying suitable targets to attack in a PaaS deployment requires some thought. After all, cryptographic keys that commonly form their most natural targets are largely absent in typical PaaS environments where cryptographic protections (e.g., storage encryption, or application of TLS encryption to network traffic) are com-

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s). Copyright is held by the author/owner(s).
CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
ACM 978-1-4503-2957-6/14/11.
http://dx.doi.org/10.1145/2660267.2660356.

[1]While "instance" typically refers to an instantiated VM in an IaaS setting, here we borrow the term for the PaaS setting, to refer more generically to a collection of running computations on one physical machine that are associated with the same tenant and should be isolated from other tenants.

Copyright @ 2021 Authors

monly provided as a service by the cloud operator, often on a different computer than those used to host tenant instances.

In this paper we report on our investigation of cache-based side channels in PaaS clouds that, among other things, identifies several novel targets (in the context of cross-tenant side-channel attacks) for PaaS environments:

1. We show how an attacker instance can infer aspects of a victim web application's responses to clients' service requests. In particular, we show that an attacker can reliably determine the number of distinct items in an authenticated user's shopping cart on an e-commerce site (the victim instance) running the popular Magento e-commerce application.
2. We show how an attacker instance can hijack a user account on a web site (the victim instance) by predicting the pseudorandom number it embeds in a password reset link. We specifically demonstrate this attack against the PHP pseudorandom number generator that the site uses.
3. We show how an attacker instance can monitor the victim so as to obtain a padding oracle to break XML encryption schemes. In particular, we demonstrate a Bleichenbacher attack [6] against SimpleSAMLphp, an open-source SAML-based authentication application that implements PKCS#1 v1.5 RSA encryption in a manner resistant to these attacks via other vectors (but not via our side-channel attacks).

We stress, moreover, that we have successfully mounted each of these attacks in commercial PaaS clouds (though obviously against victims that we deployed ourselves). Moreover, as a side effect of doing so, we have also addressed how to achieve co-location of an attacker instance with a victim instance in these PaaS clouds. To our knowledge, our attacks are thus the first granular, cross-tenant attacks demonstrated on commercial clouds, PaaS or otherwise.

A key ingredient in our attacks is a framework we develop through which the attacker instance can trace a victim's execution paths inside shared executables. Starting with the control-flow graph (CFG) of an executable shared with the victim, our framework consists of building an *attack nondeterministic finite automaton* (*attack NFA*) that prescribes the memory chunks (see Sec. 3.1) that the attack instance should monitor over time (using a known cache-based side channel [14, 38]) in order to trace the victim's execution path in the CFG. This general framework can then be used to characterize the victim's execution for specific attacks, such as the exact number of times a certain execution path segment was traversed in a short interval (in the first attack above); the precise time at which certain path segments were traversed by the victim (as in the second attack); or the direction taken in a specific branch of interest (in the third attack). We believe the attack NFA framework that we introduce here will be similarly useful in subsequent work on both evaluating and defending against cross-tenant side-channel attacks.

To summarize, then, the contributions of this paper are threefold: (i) a general framework for expressing and guiding cross-tenant side-channel attacks leveraging shared executables; (ii) identification of novel and important targets for side-channel attacks in PaaS environments; and (iii) demonstration of attacks against these targets in commercial PaaS clouds. Sec. 2 provides background on PaaS clouds and common isolation techniques they employ and specifies the threat model we assume in our work. Sec. 3 describes our attack framework. Sec. 4 discusses our strategies for achieving and confirming co-location of attacker instances with victims. Sec. 5–7 then detail our three attack demonstrations outlined above. We discuss ethical considerations, extensions of the attacks, and potential countermeasures in Sec. 8 and conclude in Sec. 9.

PaaS cloud	URL (http://...)	Isolation
AppFog	www.appfog.com	User
Azure	azure.microsoft.com	VM
Baidu App Engine	developer.baidu.com/en	Container
Cloud Foundry	cloudfoundry.org	User
DotCloud	www.dotcloud.com	Container
Elastic Beanstalk	aws.amazon.com/elasticbeanstalk/	VM
Engine Yard	www.engineyard.com	VM
Heroku	www.heroku.com	Container
HP Cloud Application PaaS	www.hpcloud.com/products-services/application-paas	Container
Joyent SmartOS	www.joyent.com	VM
OpenShift	www.openshift.com	Container
WSO2	wso2.com/cloud	Runtime

Table 1: Example PaaS isolation techniques

2. BACKGROUND

Cloud computing systems are often categorized as either IaaS, PaaS, or Software-as-a-Service (SaaS). IaaS clouds enable users to launch virtual machines that they control on the provider's infrastructure, and provide access to various low-level resources including storage and networking. A canonical public PaaS cloud allows customers to upload interpreted source code (e.g., PHP, Ruby, Node.js, Java) or even application executables, which are then run in a provider-managed host operating system (OS). This OS may itself be running within a guest VM on a public IaaS platform such as Amazon EC2. The host OS facilitates data storage, monitoring and logging, and other value-adds that enable customers to quickly provision applications.

A canonical PaaS use case is dynamic web hosting, where the customer provides scripts or applications defining the webpage (i.e., PHP scripts or similar) and a MySQL schema, while the cloud provides integration of middleware to facilitate data storage, performance monitoring and mobile integration if desired. The convenience and flexibility that PaaS provides to customers, together with the fact that mature IaaS clouds enable quick time-to-market for a new PaaS system, has lead to an explosion in the number of offerings.

PaaS Tenant Isolation

In order to increase server utilization and reduce operating cost, PaaS systems are usually multi-tenant, meaning they run multiple customers' instances on the same operating system. As such, isolation between tenants is essential for the security of PaaS clouds. In Table 1 we summarize the isolation mechanisms used in a variety of PaaS systems, and describe these models in more detail below.

Runtime-based isolation. Some PaaS clouds host applications owned by multiple tenants in the same process and isolate them with application runtimes. Multiple tenants

Copyright @ 2021 Authors

therefore may share, e.g., the same JVM environment, and be isolated only by JVM runtime security mechanisms.

User-based isolation. A more widely used isolation technique is traditional user-based isolation within the host OS. Each hosted application runs as a non-privileged user on the OS, and the instance is a set of processes run by that user. Basic OS-facilitated memory protection prevents illegal memory accesses across instance boundaries, and correctly configured discretionary access control (DAC) in Unix-like systems prevents cross-tenant file accesses.

Container-based isolation. The main limitation of user-based isolation is the unrestricted use of computer resources by individual instances. This has been relatively recently addressed with the advent of Linux containers, as implemented by Linux-VServer (linux-vserver.org), OpenVZ (openvz.org), and LXC (linuxcontainers.org). The last has been merged into mainstream Linux kernels. A container is a group of processes that are isolated from other groups via distinct kernel namespaces and resource allocation quotas (so-called control groups or cgroups). A popular open-source project, Docker, which has been adopted by several PaaS offerings, is built atop LXC to facilitate the management of Linux containers.

VM-based isolation. Some PaaS clouds give each customer instance a separate IaaS VM instance, thereby leveraging the isolation offered by modern virtualization.

The attacks we demonstrate in this paper were performed on clouds offering container-based isolation, but as they exploit features common to both container-based and user-based isolation, we believe they are equally applicable to clouds protected by user-based isolation.

Threat model

We consider attacks by the PaaS provider (or other malicious insiders) as out of scope. The same trust extends to any underlying IaaS provider. Should the IaaS cloud be public (e.g., EC2) then its malicious IaaS customers represent a threat to PaaS customers, but not one that we explore further. Rather we focus on other malicious customers of the PaaS cloud, and container-based isolation in particular.

Thus both the adversaries and the victims in our threat model are users of a PaaS system. An adversary seeks to (i) arrange for a malicious instance it controls to be scheduled to run within a different container on the same host OS as the target victim and (ii) extract confidential information from the target victim using this vantage point.

3. ATTACK FRAMEWORK

In this section, we present an attack framework that enables an adversary to conduct a cache-based attack to track the execution path of a victim and, in doing so, to extract a secret of interest from the victim. We will first describe the FLUSH-RELOAD-based side channels exploited in this study (Sec. 3.1), and then develop an *attack nondeterministic finite automaton* or *attack NFA* (Sec. 3.2–3.3) from the control-flow graph (CFG) of an executable shared with the victim. We defer the actual demonstration of security attacks to later sections.

Side Channels via Flush-Reload

We leverage a type of cache-based side channel that was first reported by Gullasch et al. [14], who demonstrated its use by an attack process to extract Advanced Encryption Standard (AES) keys from a victim process when both were running within the same OS. The attack was studied on a single-core processor and exploits the adversary's process' ability to evict data in physical memory pages it shares with the victim process from the CPU cache (e.g., via the instruction clflush[2]). The technique was later extended by Yarom and Falkner to multi-core systems with a shared last-level cache [38]. They refer to their attack as FLUSH-RELOAD. In this work, we further extend the use of the FLUSH-RELOAD side channels to more general attack scenarios.

Basic Flush-Reload. The basic building block of a FLUSH-RELOAD attack is as follows. A *chunk* is a cacheline-sized, aligned region in the physical memory that is mapped into the adversary's address space. For example, if a cacheline is of size 64B, then each address that is a multiple of 64B defines the chunk starting at that address.

- FLUSH: The adversary flushes chunks containing specific instructions located in a memory page it shares with the victim out of the entire cache hierarchy (including the shared last-level cache) using the clflush instruction.
- FLUSH-RELOAD interval: The adversary waits for a prespecified interval while the last-level cache is utilized by the victim running on another CPU core.
- RELOAD: The adversary times the reload of the same chunks into the processor. A faster reload suggests these chunks were in the last-level cache and so were executed by the victim during the FLUSH-RELOAD interval; a slower reload suggests otherwise.

We refer to the chunks being FLUSH-RELOADed by the adversary as being *monitored*, since FLUSH-RELOAD essentially monitors access to data in the chunk. FLUSHing a chunk via clflush, and so monitoring that chunk, can be done without knowing the physical address of the chunk, since clflush takes the chunk's *virtual* address (in this case, in the adversary's address space) as its operand. We call a faster reload during the RELOAD phase an *observed event* or *observation*. We also adopt concepts from statistical classification and use the term *false negative* to refer to missed observations of the victim's access to the monitored chunk and *false positives* to refer to observed events that are caused by reasons other than the victim's access to the monitored chunk.

Flush-Reload Protocols. We define a FLUSH-RELOAD protocol, in which the adversary process monitors a list of chunks simultaneously and repeatedly until instructed otherwise. It will first try to RELOAD the first chunk, record the reload time and FLUSH it immediately afterwards. Then it will repeat these steps on the second chunk, the third, and so on, until the last chunk in the list. Then the adversary will wait for a carefully calculated time period before starting over from the first chunk, so that the interval between the FLUSH and RELOAD of the same chunk is of a target duration.[3] From the RELOAD of the first chunk to the end of the waiting period is called one *Flush-Reload cycle*. An illustration of the FLUSH-RELOAD protocol is shown in Fig. 1.

[2]The clflush instruction takes a virtual address as the operand and will flush all cachelines with the corresponding physical address out of the entire cache hierarchy.
[3]Variation in the duration on the order of one or two hundred CPU cycles may occur as the reload of a chunk does not take constant time.

Copyright @ 2021 Authors

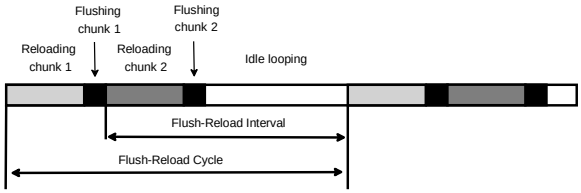

Figure 1: An example of a Flush-Reload protocol in which two chunks are monitored at the same time. Gray rectangles are Reloads of two chunks and dark squares are immediate Flushes of the prior Reloads.

From CFGs to Attack NFAs

In this section we provide a framework to leverage FLUSH-RELOAD attacks as a primitive in a larger attack strategy to trace the execution path of a victim instance during (at least part of) its execution. Specifically, we develop *attack NFAs* that prescribe the order in which different chunks should be monitored using FLUSH-RELOAD attacks, based on what has been learned so far.

The development of an attack NFA to attack a target victim begins with a control-flow graph (CFG) [1] of the executable[4] shared with the victim. As usual, each node of the CFG is a *basic block* of instructions, and an edge from one basic block to another indicates that the latter can immediately follow the former in execution. Let B denote the set of basic blocks of the victim instance, and let E denote the directed edges of its CFG.

When the shared executable is loaded, its organization in memory determines a function $\mathsf{BBToChunks} : B \to 2^{\mathsf{C}}$ that describes how each basic block shared with the victim (i.e., in the shared executable) is stored in one or more chunks mapped into the adversary instance's virtual memory. Here, C is the set of all chunks mapped into the adversary's virtual memory and occupied by the shared executable, and 2^{C} denotes the power set of C. That is, each basic block in B is mapped to one or more chunks, by $\mathsf{BBToChunks}$. Although the chunks to which each basic block is mapped are usually contiguous in memory, this might not be true when those chunks span the end of a memory page.

Like a regular NFA, the attack NFA is defined as a tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, Σ is a set of symbols, $\delta : Q \times \Sigma \to Q$ is a transition function, q_0 is the initial NFA state, and $F \subseteq Q$ is a set of *accepting* states. To each state $q \in Q$ is associated a set of chunks, denoted $\mathsf{mon}(q) \subseteq \mathsf{C}$, that contains the chunks the adversary will monitor while in state q. Note that $\mathsf{mon}(q)$ might be the same for multiple states q.

The symbols Σ consumed by the NFA is the set $\Sigma = \mathsf{C} \times \mathbb{N} \times \mathbb{N}$ where $\mathbb{N}$ is the set of natural numbers. Specifically, the meaning of the transition $(q, (c, \ell, u), q') \in \delta$ is: while in state q and so monitoring the chunks $\mathsf{mon}(q)$, if the adversary detects the victim's use of chunk c within the interval $[\ell, u]$ (in units of FLUSH-RELOAD cycles since entering q), then the adversary transitions to q' and begins monitoring the chunks $\mathsf{mon}(q')$. We allow ℓ to be zero; detecting the victim's use of c in zero FLUSH-RELOAD cycles since entering state q means that c was detected in the same FLUSH-RELOAD cycle that caused state q to be entered.

[4]We use the term "executable" to refer to both executable files and shared libraries in this paper.

In light of this intended meaning of the attack NFA, the transition function should satisfy certain constraints.

- Observability: If $(q, (c, \ell, u), q') \in \delta$, then $c \in \mathsf{mon}(q)$. Otherwise, an adversary in state q will not observe the victim using c. If in addition $(q', (c', 0, u'), q'') \in \delta$, then $\mathsf{mon}(q') \subseteq \mathsf{mon}(q)$, since for transition $(q', (c', 0, u'), q'')$ to become enabled with no FLUSH-RELOAD cycles after transitioning to q', c' must be monitored in q (as must other chunks included in $\mathsf{mon}(q')$ due to recursive application of this rule to additional "downstream" states like q'').
- Feasibility: To each state q there corresponds a basic block b such that for each transition $(q, (c, \ell, u), q') \in \delta$, there is a (possibly empty) path in the CFG from b to a basic block b' (corresponding to q') that can be traversed in no fewer than ℓ and no more than u FLUSH-RELOAD cycles and such that $c \in \mathsf{BBToChunks}(b')$. Intuitively, it is this execution path that the adversary detects in transitioning from state q to q'.

In addition, in practice it is important to design the attack NFA so that the number of monitored chunks in any state is constrained, since monitoring many chunks simultaneously poses difficulties.

A transition is taken out of a state at the first FLUSH-RELOAD cycle that enables a transition. Still, it is possible for multiple transitions to become enabled in the same FLUSH-RELOAD cycle, in which case an arbitrary enabled transition is taken. In this respect, the automaton is nondeterministic.

The designated initial state q_0 represents the shared executable's entry point(s) of interest to the adversary. That is, $\mathsf{mon}(q_0) \cap \mathsf{BBToChunks}(b) \neq \emptyset$ for each basic block b that the adversary wants to detect initially. The set F of accepting states is chosen by the adversary to reflect having tracked the execution of the victim sufficiently far to permit his inference of the targeted information about the victim with sufficient confidence.

After the attack NFA is constructed, the adversary may employ it to reconstruct the victim's execution path by simultaneously (i) triggering the victim's execution by sending a request to victim's web application interface, and (ii) inducing its co-located attacker application to start monitoring $\mathsf{mon}(q_0)$. If the NFA transitions to an accepting state, the adversary knows the execution path of interest is taken by the victim. We have found that in practice, a well designed NFA usually leads to successful identification of an execution path of the victim application.

Practical Construction of Attack NFAs

In this section we discuss how an adversary can construct attack NFAs in practice.

Basic Strategy

In PaaS clouds an application usually consists of a set of scripts written in scripting languages that manage dynamic web content, a set of shared libraries that implement the runtime of the programming language or any other supporting functionality (e.g., cryptography, database access), and a web server executable that serves the web requests and interacts with the scripting language runtime.

It is not necessary to construct the attack NFA from a full CFG of the victim application. If the source code of the shared executables of interest is unavailable to the adversary,

Copyright @ 2021 Authors

he can make progress on the attack with the following steps: (1) disassembling these shared executables and constructing partial CFGs from the results; (2) manually analyzing these partial CFGs and selecting blocks along the execution paths of interest for which chunks should be monitored; and (3) constructing the attack NFA with the help of online training, in which the adversary monitors all chunks of interest at once and triggers the victim's activity that he would like to capture by submitting appropriate requests. During phase (3), the FLUSH-RELOAD protocol will report a sequence of observed events on the monitored chunks. The temporal order of these events suggests the NFA states and chunks to monitor in each, and the relative timestamps can help train the ℓ and u values for each transition. Multiple training trials will help refine the constructed attack NFA.

However, adversaries usually face a more favorable scenario in practice. Source code of the victim application is available to the adversary in many cases: Since most PaaS clouds are built based on Linux distributions, most web servers, application runtimes and supporting libraries are open-source. Moreover, about 37% of the top 10 million websites use a third-party content management system such as WordPress, Drupal, or Magento [37], the source code of which is either open or obtainable with a fee.

If the adversary has the source code of the shared executables as well as the scripts for managing dynamic web content, the above attack steps can be facilitated with the additional information. For instance, step (1) can be replaced by performing static control-flow analysis on the source code, and step (2) and (3) can be assisted by replicating the same PaaS environment offline and tracking the victim application's control flow dynamically. For example, the adversary may run the entire web application in Valgrind [24] and trigger various victim activities with manufactured web requests, recording the control flow that results from each. However, even so, the training step (3) is still necessary to determine the ℓ and u values for the transitions.

Although we have developed some software tools to facilitate attack NFA construction, constructing an attack NFA with or without source code is still mostly a manual process and depends in large part on the attack goals — in particular, which execution paths in the victim the adversary needs to detect. In Sec. 5–7, we will give several examples of how to construct attack NFAs for different types of attacks.

Reducing Side-Channel Noise

One challenge that we have overlooked so far is noise in the FLUSH-RELOAD side channel. Here, "noise" refers to false positives and false negatives in the RELOAD phase. Compared with the PRIME-PROBE attacks used in many previous works (e.g., [28, 30, 25, 34, 40]), FLUSH-RELOAD attacks involve relatively less noise, since the adversary is able to tell whether the victim accessed the data in the chunk the adversary is monitoring, versus simply some data mapped to the same cache set. Nevertheless, the technique still suffers from many sources of noise in practice.

Sources of noise. We discuss, in turn, false negative noise due to race conditions and unobserved duplicate accesses, and then false positive noise due to false sharing of chunks, hardware cache prefetching, and background activities from other processes sharing the same memory pages. These sources of noise affect the granularity and reliability of the attacks that we will develop in subsequent sections.

A *race condition* here refers to the situation where two memory loads of the same chunk are issued from two CPU cores roughly at the same time. In such cases, the outcome of the RELOAD step can be unpredictable. The access of the shared chunk by the victim may be missed by the adversary if it overlaps with the adversary's memory load. Because the adversary increases the risk of such an overlap as it shortens its FLUSH-RELOAD interval, the adversary is limited in how far it can shrink that interval.

Another source of noise is the victim itself—a victim's first access of a chunk can be missed by FLUSH-RELOAD monitoring if it accesses the chunk a second time before the RELOAD of the adversary. We call this an *unobserved duplicate access*. This type of noise is particularly significant when applying the attack framework to count the repeated use of the same chunk, which will be discussed in our attack scenarios.

False sharing usually refers to a cache usage pattern in distributed, coherent cache systems that degrades the performance of the cache [7]. Here we refer to false sharing of a cacheline to refer to cases in which two separate program components share the same chunk and hence the FLUSH-RELOAD monitoring of one component may be misled by the execution of another. For example, the memory layout of a function rarely aligns perfectly within chunks, and the beginning and the end of a function usually share the same chunks with other functions.

Depending on the hardware implementation, data cache *prefetching* may load more than one consecutive chunk into the cache upon a cache miss. This behavior can result in false positives, since observed events may be caused by data prefetching.

As multiple processes in the operating system may share the same executables, and so the same memory pages that contain executable code, activities from processes other than the victim may trigger false positives in the RELOAD phase. Especially in PaaS cloud settings, tens or even hundreds of applications may share the same set of executables, and careful use of the FLUSH-RELOAD side channel is required to reduce such *background noise*.

Overcoming noise. We have found that several design principles help to overcome the above sources of noise.

- Select an appropriate FLUSH-RELOAD interval. A shorter interval will increase the chance of race conditions, and longer intervals will incur more unobserved duplicate accesses. As such, in our case studies (Sec. 5–7) we determined the length of the interval empirically to minimize false negatives, which resulted in a FLUSH-RELOAD interval of about 1μs in each case.
- Avoid monitoring chunks that correspond to frequently used basic blocks in the CFG, to reduce false positives due to background noise. For instance, the wrapper functions of system calls in libc are inevitably shared and used by multiple processes concurrently, and therefore will frequently induce noise in the FLUSH-RELOAD channel. It is better to monitor entries in the procedure linkage tables (PLT) of other libraries that call these functions, instead, as they tend to be less frequently used.
- When the same chunk contains the end of one basic block and the beginning of another, avoid monitoring this chunk if possible, due to false positives resulting from the false sharing.

Copyright @ 2021 Authors

- Use the timing constraints, ℓ and u, of the transitions to reduce false positives. While false positives may occur due to false sharing, cache prefetching, and background noise, the timing constraints of a transition can often rule out these false observations for not falling into the specified interval $[\ell, u]$.

4. CO-LOCATION IN PAAS

To exploit side channels in PaaS environments, an adversary must first somehow achieve co-location of a malicious instance on the same OS as a target. Ristenpart et al. [32] explored co-location vulnerabilities in the setting of IaaS clouds. To the best of our knowledge, no one has investigated co-location in PaaS settings. We therefore provide a preliminary empirical study of the ability to co-locate an attacker instance with a victim instance in modern public PaaS clouds, leveraging our proposed attack framework to detect success.

Co-location attacks consist of two steps. First, the adversary employs some strategy for launching (typically a large number of) instances on the cloud service. Second, each of these instances attempts to perform co-location detection. For the first step, we explore only the simplest strategy in which we repeatedly launch instances that check for co-location until success is achieved.

Co-location detection. For the second step, we use a FLUSH-RELOAD side channel to detect whether any of the instances co-locates with the victim instance. To detect co-location, the adversary sends an HTTP query to the victim instance and instructs each of the attacker instances to simultaneously monitor a certain execution path using the techniques proposed in Sec. 3. If the execution path is detected, the adversary will have some confidence that the detecting attacker instance is co-located with the victim. However, this approach may have false positives, in which not-co-located instances were reported as co-located due to activities of other tenants sharing the same OS, and false negatives, in which co-located instances were not reported so. In order to increase the confidence, two strategies can be taken: (1) induce and monitor for rare events to reduce false positives; or (2) use multiple trials to reduce both false positives and false negatives.

The execution path to be monitored may vary. In our experiments, we considered a victim instance that ran a popular PHP e-commerce application, Magento. To differentiate the query sent by the adversary from background noise, we simply used a relatively unusual query with an associated uncommon execution path. By inspecting the source code of the Magento application, facilitated by dynamic analysis using Valgrind, we found the functions `xmlXPathNodeSetSort()` in `libxml2.so` and `php_session_start()` in either `libphp5.so` or `php5-fpm` (depending on the version of PHP used by the cloud) are called sequentially during a (failed) login attempt. We confirmed with dynamic analysis of other types of queries that the execution paths that traverse both functions are uncommon. Therefore, we constructed an attack NFA as shown in Fig. 2. In this figure, for example, c2 corresponds to a chunk in `php_session_start()`, and the number of FLUSH-RELOAD periods allowed to transition out of the state q in which BBToChunks(q) = {c2} is any in the range $[1, T]$, where T is the maximum FLUSH-RELOAD cycles before the attack NFA stops accepting new inputs. In

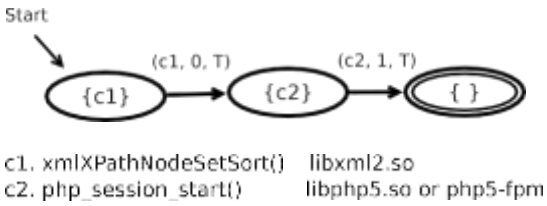

Figure 2: The attack NFA used for detection of co-location with PaaS Magento e-commerce instances. Initial state q_0 indicated by "Start" and accepting states indicated with double ovals. T is the maximum Flush-Reload cycles without transitioning before the NFA stops accepting new inputs.

our experiments, T corresponded to one or two seconds of wall clock time. One can of course adapt the above strategy easily to targets beyond Magento.

We observed in earlier experiments that some cloud services tend to schedule applications with different runtimes (e.g., PHP versus Ruby) on different machines. Fortunately (for an adversary) it is easy to choose the same runtime as the victim should it be known to the adversary, which we did in all of our experiments. If it is not known, the adversary can simply repeat the co-location attack for each runtime as there are only a handful in any given cloud.

Co-location validation. To obtain ground truth for evaluating efficacy, we took advantage of the fact that during our experiments we controlled both the attacker and victim instances. In particular, we augmented the above procedure to also have both attack instances and the target victim establish a TCP connection with an external server under our control. (Most clouds have their firewalls configured to allow outbound traffic.) This revealed the IP addresses associated with each instance; if two instances shared the same IP address they were hosted on the same (virtual) server. It is worth noting that a Network Address Translation (NAT) configuration in the cloud provider's network would hinder this approach. However, we did not observe this problem in our experiments. We also note that this co-location check could potentially be used in cases in which real adversaries can obtain the IP address of the target, and so this might be directly useful by real adversaries. However, in many cases clients do not directly connect to PaaS instances, hitting a load balancer or HTTPS endpoint first. Thus we only used the IP comparison approach to validate that the previously described side-channel based co-location check worked.

Co-location experiments. We provide some initial proof-of-concept experiments regarding the ability of an adversary to obtain co-location with a single victim. We do so for two popular public PaaS services: DotCloud and OpenShift.

The client control interfaces are different in the two services. In OpenShift, a target victim instance was launched and after a certain amount of time (typically on the order of a few hours, though times varied), the adversary launched attack instances one-at-a-time (with a 30-second interval to reduce the stress of the experiments on the cloud fabric) until one obtained co-location with the victim as indicated by the attack NFA. In DotCloud, the experiments were conducted similarly, except that the control interface enabled us to launch attack instances ten-at-a-time via static scaling

Copyright @ 2021 Authors

(vs. dynamic scaling in OpenShift which prevents doing so). We repeated this process three times for each cloud. We report in Table 2 the number of instances that the adversary launched before a successful co-location. As can be seen, every trial succeeded in every cloud, providing strong evidence that an adversary is very likely to be able to obtain co-location with a target. (Indeed, in the course of writ- ing this paper, we never were unable to achieve co-residency with our victim instance in these clouds.) The number of trials required, however, varied greatly. Even in the worst observed case, with 120 instances in OpenShift, co-location was obtained after 3.2 hours and at a total cost of zero US dollars, as we did not exceed the limits of the free tier.

	Trials			Miss Detection	
	1st	2nd	3rd	FP	FN
DotCloud	≤ 10	≤ 10	≤ 10	0.00	0.03
OpenShift	98	120	5	0.00	0.49

Table 2: Number of sequentially launched instances before co-location.

We used this experimental data to test the accuracy of our co-location detection attack NFA. Specifically, we ran for each cloud the co-location detection 100 times using two instances which were co-located (as per IP address checks) and 100 times on instances which were not co-located (as per IP address checks). The detection rates are also shown in Table 2: *FP* indicates the rate of false positives, and *FN* indicates the rate of false negatives. We believe the high false negative rate in OpenShift was due to CPU resource contention, as the applications were run on a two-core VM sharing CPUs with hundreds of processes. We defer further discussion on this issue to Sec. 8.3. Nevertheless, the result indicates the rare execution path represented in Fig. 2 successfully reduced background noise; repeating the co-location test five times resulted in a false negative rate of $.28 \approx (.49)^5$.

5. CASE STUDY 1: INFERRING SENSITIVE USER DATA

In this section and the two that follow, we present three examples in the form of case studies that demonstrate how an adversary can apply our proposed framework to accomplish a variety of real-world attacks. Our experimental environment was common to all three studies.

Experimental environment. Our evaluations were conducted in a public PaaS cloud, DotCloud. We will discuss the ethical considerations surrounding our experiments in Sec. 8.1. The software and hardware stack in DotCloud was out of our control and was not officially reported by the provider. By observing data extracted from procfs, a pseudo filesystem presenting system information, and data available from the PaaS control fabric, however, we believe the applications in our experiment were run on a VM with four virtual CPUs operated by Amazon EC2 in us-east-1a datacenter. The physical CPU was a 2.4GHz Intel Xeon processor E5-2665, which has 8 cores sharing a 20MB last-level cache. Moreover, we believe the operating systems that supported the applications were Ubuntu 10.04.4 LTS on Linux kernels version 2.6.38. The tenants were isolated with Linux containers.

In all three case studies, we created two accounts using different email addresses and user information in DotCloud, designating one of them as the victim account and the other as the attacker account. We believe the victim and attacker accounts were treated as two separate, mutually-distrusting accounts by the cloud provider. Since the victims were PHP applications in our case studies, all attacker applications were designed to operate on the same runtimes to facilitate their co-location with the victim, which was achieved as described in Sec. 4. DotCloud used php-fpm (version 5.4.6), which interacted with the Nginx web server (http://nginx.org) and processed PHP requests. In all experiments, the FLUSH-RELOAD cycle was set to be 2400 clock cycles, corresponding to about one microsecond in real time.

Attack Background

Our first case study explores a relatively simple attack, a good starting point for end-to-end illustration of our techniques. We show how our proposed attack framework permits inference of the responses of a victim web application to client requests. Specifically, an adversary may combine what is known as a cross-site request (CSR) with the FLUSH-RELOAD side channel to infer the number of distinct items in a user's shopping cart on an e-commerce server.

There have been various related *timing* attacks demonstrated on web privacy, e.g., [13, 22]. Particularly similar to our case study here is a CSR-based attack described by Bortz et al. [8] that likewise infers the number of distinct items in a user's shopping cart. As their attack relies on the timing of request fulfillment, they propose and implement a countermeasure that enforces uniform server response timing. The attack we present here depends instead on execution tracking via an attack NFA, and thus defeats timing-side-channel countermeasures of this kind.

Cross-site requests. The target of the adversary in this case study is, specifically, a user that is authenticated to a victim e-commerce site. We presume, however, that the adversary cannot compromise the credentials of the user for the victim site, and only makes use of the side channel to observe data retrieved by the user. A passive adversary might be unable to determine the identities of users accessing the victim site. We consider an alternative strategy in which the adversary prompts user retrieval of the target data by means of a *cross-site request*.

CSRs are HTML requests made to a third-party resource, that is, one hosted by a domain other than that serving the HTML. While there are legitimate uses for such indirection, it can also serve as a basis for requests that make improper use of a user's credentials, as in our attack here. A CSR requires that the adversary lure the user to a site that serves HTML crafted by the adversary to redirect the user's browser to the victim's e-commerce site, e.g., <img src = "http://victim-site.com/index.php/checkout/cart/">. If a user Alice has been previously authenticated to the domain www.victim-site.com, then her browser will often obtain and cache credentials for the domain, such as cookies, and automatically re-authenticate on subsequent visits. Thus, in our attack, www.victim-site.com will see an authenticated request originating from Alice's browser, unaware that the request was triggered by an adversary.

Web applications may include protections against malicious CSRs, such as requiring explicit user authorization of resource requests or inserting session-specific random syn-

Copyright @ 2021 Authors

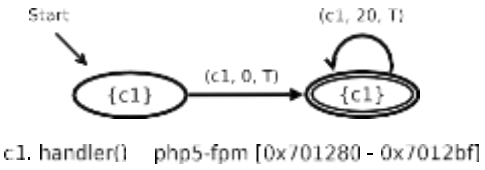

Figure 3: Attack NFA for case study in Sec. 5. Initial state q_0 indicated by "Start" and accepting states indicated with double ovals. T is the maximum Flush-Reload cycles without transitioning before the NFA stops accepting new inputs.

chronizer tokens in HTML forms and links. Often these protections are confined, however, to what are called cross-site request forgery (CSRF) attacks, which cause state changes (known as "side-effects") in the server. The CSR we exploit for our attack here has no side effects, and will thus be allowed by most victim servers.

Evaluation in Public PaaS

We empirically evaluated our proposed attack in DotCloud against the Magento e-commerce application (version 1.8, the latest version as of this writing). This is a popular open source e-commerce application, used by roughly 1% or about 200,000 of the top 10 million websites ranked by Alexa (`http://www.alexa.com`) [37]. We reiterate that our goal is for an attacker instance to reliably determine the number of distinct items in an authenticated user's shopping cart on the e-commerce site of the victim. Our attack cannot determine the quantity count for a given item.

We assume, as noted above, that the adversary can lure an authenticated user of the victim Magento website to an HTML page hosted in its own webserver, thereby triggering a CSR in which the user requests her shopping cart on the victim site. We simulated the user on Google Chrome (v34). We expect the attack to work on other browsers that support a similar range of cross-origin requests.

Attack details and results: The attack NFA we constructed in this example is highly dependent on the specifics of the Magento web application. We analyzed the application with Valgrind. We observed that a Zend opcode handler (which we call `handler()` for convenience[5]), which is implemented in the executable `php5-fpm`, is invoked every time an item in the shopping cart is displayed.

To count the number of items in a shopping cart, therefore, it suffices for the adversary to count the number of invocations of the `handler()` function using the FLUSH-RELOAD side channel. In our experiments, an interval of at least 20 FLUSH-RELOAD cycles elapsed between the display of two distinct items. We took this interval length to be a lower bound on the time between calls to `handler()` within the NFA we constructed for the attack, depicted in Fig. 3.

The evaluation was performed on DotCloud as follows. The victim user placed m distinct items in her shopping cart, for $m \in \{0, 1, 2, 3, 4, 5, 6\}$. We repeated our experiment 10 times for each value of m. The number of successes for each number m of distinct items, that is, the frequency with which the adversary correctly determined m from a single trial, is shown in Table 3. Also shown is that when the adversary inferred m incorrectly, its inference was nevertheless very close to correct.

Items in cart (m) \ Items detected in cart	0	1	2	3	4	5	6	7
0	10							
1		10						
2			9	1				
3				10				
4					1	9		
5						1	9	
6							1	8 1

Table 3: Item count inferences by the adversary. Each table entry indicates the number of experiments yielding a given (true count, inferred count) pair over 10 trials per row. Entries on the diagonal, which predominate, correspond to correct inference.

6. CASE STUDY 2: PASSWORD-RESET ATTACKS

In this second case study, we show how to employ our attack framework to compromise the pseudorandom number generators (PRNGs) used by many web applications in authenticating password reset requests. An adversary can exploit this ability to reset the passwords for and thus obtain control of the accounts of arbitrarily selected users.

Our attack targets the PRNG present in certain programming language runtimes (e.g., PHP), which relies upon system time (e.g., `gettimeofday()`) as a source of seed entropy. With a malicious application that is co-located with the victim application, the adversary is able to detect system calls such as `gettimeofday()`, reconstruct the internal state of the PRNG, and thereby reproduce its entire output.

The ability to mount password-reset attacks is one consequence of this PRNG vulnerability. Such attacks are of particular concern because an adversary can trigger a password reset on a web application for a user with knowledge of the user's account name or email address alone. To authenticate the user, a web application will typically use a PRNG to generate a random string R, and then embed this string in the URL of a password reset link sent to the user's registered email address. By learning the state of the web application's PRNG, a co-located attacker instance can reproduce the password reset token R, reset the password before the user does, and hijack the user's account. We stress that the adversary does not need access to the user's email to accomplish this attack. In this section, we demonstrate such a password-reset attack against PHP-based web applications in public clouds.

Weaknesses in PHP PRNGs have been previously reported (e.g., [12, 17]). A recent study by Argyros and Kiayias [3] gave several attacks, one of which involves recovery of the seed values of the PHP system's PRNGs for password reset and so has the same goal as the attack in our own case study. Their attacks (which are against victims presenting a much smaller search space than ours, see [3, Sec. 6.2]), however, require sending repeated requests to the victim server, which

[5]As only the virtual address of the handler was required to construct the attack NFA, we were able to perform the attack without studying the Zend source code. Therefore, the name of the function, which is hidden in the result of an `objdump`, remains unknown to us.

Copyright @ 2021 Authors

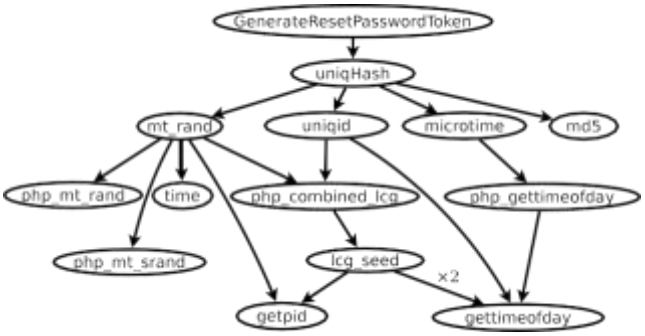

Figure 4: The call graph of password reset token generation in PHP applications.

may take several minutes and may result in attack detection. In comparison, after a setup phase requiring a small brute-force attack (2^{20} offline trials), our attack requires at most four online queries to compromise a user account. It is thus almost instantaneous and scales easily to a large number of accounts.

Background on PRNG in PHP

The PHP runtime provides several functions by which applications can obtain or generate (pseudo)random numbers. For instance, during the process of password reset token generation, most PHP applications call APIs such as microtime(), mt_rand(), and uniqid(). Internally, the microtime() function calls gettimeofday() to obtain the current system time in the form of the number of seconds and microseconds since the Unix epoch (0:00:00 1 January 1970 UTC). The mt_rand() function, which is the interface to the PHP internal Mersenne Twister generator, automatically initializes its own internal state, if mt_srand() has not yet been invoked, with a random seed generated using functions time(), php_combined_lcg(), and getpid(). The time() function merely returns the number of seconds since the Unix epoch, and therefore has low entropy. The function php_combined_lcg() combines two linear congruential generators (with prime periods $2^{31}-85$ and $2^{31}-249$) to generate a long-period pseudorandom sequence (the product of the primes). The initialization of php_combined_lcg() depends on the lcg_seed() function, which generates random seeds by calling getpid() once and gettimeofday() twice. These function calls and dependencies are shown in Fig. 4. While the range of options for seeding the PRNG in PHP systems may seem convoluted, as Fig. 4 shows, the only sources of entropy for the PRNG seed are gettimeofday(), time(), and getpid(). By monitoring invocations of the gettimeofday() function, the adversary can immediately issue another call to gettimeofday() once it is called by the victim. As the adversary shares the OS with the victim web application, the result of the adversary's invocation of gettimeofday() will be very close to, if not exactly the same as, that returned to the victim application. The same is (even more) true of time(). As such, the only input to the victim PRNG that may be unknown to the adversary is the result of getpid(), which may assume any of 2^{16} values.

An adversary can initiate a password reset for its own account with the victim web application. As the adversary receives the corresponding secret string *R*, it can guess the pid and verify its correctness against *R*. Subsequent password reset attacks issued from the same connection will be served by the same process. The adversary therefore resolves virtually all entropy in the initial state of the PHP application. By continuously monitoring the invocations of mt_rand() and php_combined_lcg(), the adversary can keep track of the evolution of the PRNG and guess all the random numbers generated.

As described above, it is critical that the adversary monitors the initialization process of the PRNG, which takes place only once in the lifetime of a server process. A very common configuration (see www.apache.org and www.php.net for more information about PHP web server configurations) is to have one process, either an Apache process or a standalone PHP process, to serve each new request. As such, it is possible for an adversary to mount an active attack in which it triggers the PRNG initialization process for observation. To do so, the adversary can saturate existing server processes and force the victim application to instantiate new processes to serve subsequent requests.

Evaluation in Public PaaS

As in our previous attack, we experimented in DotCloud with the Magento eCommerce application (version 1.8). Not only are e-commerce applications very popular, and Magento especially so as mentioned above, but they are likely targets because of the severity of the password resetting attacks against them. Our investigation of the source code of other web applications reveals that a few more widely used PHP applications are susceptible to such attacks as well, such as the latest version of WordPress (http://www.wordpress.com) that is reportedly used by 21.9% of the top 10 million websites.

By default, a Magento application launches two instances, a www instance and a db instance, running on separate machines. In this experiment, the parameters in php-fpm.conf were set so that the FastCGI processes were created and terminated *dynamically* and only a small number of processes were kept when idle, which are typical settings in many web hosting configurations. The proposed attack is simple in this case, as the requests are likely to be served by newly created php-fpm processes. However, if the PHP is configured to maintain a *static* set of FastCGI processes, the adversary needs to crash the php-fpm process to create a fresh one, which could be achieved by various means [2].

Attack details and results: The strategy we employ is for the adversary to create enough HTTP "keep-alive" connections to force the creation of a new php-fpm process; then within the same connection, the adversary sends two password-reset requests—one request for an account under the adversary's control, another for the victim's account. As the first request results in email being sent to the account under the adversary's control, the adversary can use the URL (and embedded secret *R*), together with the timing information collected from the side channel, to recover the pid of the new php-fpm process. Then the password reset token generated by the second request becomes entirely predictable. The adversary maintains a local copy of the PRNG modified to inject results collected from the side channel instead of those from real system calls.

As shown in Fig. 5, five code chunks were monitored: one chunk from each of the three functions php_gettimeofday(), lcg_seed(), and uniqid(), which calls the entry point of

Copyright @ 2021 Authors

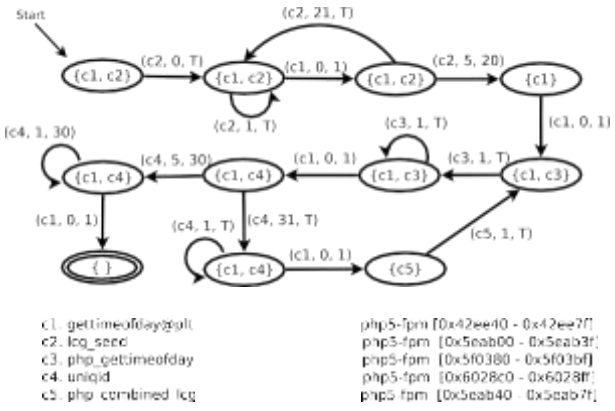

Figure 5: Attack NFA for case study in Sec. 6. Initial state q_0 indicated by "Start" and accepting states indicated with double ovals. T is the maximum Flush-Reload cycles without transitioning before the NFA stops accepting new inputs.

gettimeofday() in the procedure linkage table (PLT); the first chunk of the function php_combined_lcg(); and the chunks that contain the entry point of gettimeofday() in the PLT. A complete execution path of the password reset action that initializes the PRNG in the PHP application is c2 → c1 → c2 → c1 → c3 → c1 → c4 → c1 → c5 (indices as shown in Fig. 5). The second password reset action follows the path c3 → c1 → c4 → c1. The attack NFA is shown in Fig. 5.

In our experiments, the adversary and victim measurements of gettimeofday() sometimes differed by one bit; the response time (about 0.3μs) of the system call may at most cause a single microsecond discrepancy. Thus, to recover the pid upon initialization of the PRNG the adversary needed to perform a (trivial) offline brute-force guessing attack in a search space of size $2^{20} = 2^{16} \times 2^4$ (space 2^{16} for the pid and 2^4 for four invocations of gettimeofday()).

Once the adversary recovers the pid, a password-reset attack against a victim requires only two invocations of gettimeofday(), and thus, in our experiments, an online attack against a (tiny) space of size $4 = 2^2$. We emphasize that because the adversary is performing password reset and not password guessing, there is no account lockdown in response to an incorrect guess. So the adversary in our experiments could quickly guess the correct embedded secret R in the URL of the password link sent to the victim and then reset the victim's password.

7. CASE STUDY 3: SAML-BASED SINGLE SIGN-ON ATTACKS

In this final case study, we use our side-channel attack framework to instantiate a padding error oracle sufficient for mounting a Bleichenbacher attack [6] against PKCS#1 v1.5 RSA encryption as used in XML. Bleichenbacher attacks allow the decryption of a target RSA ciphertext (although not key recovery). While this class of attacks has been known since 1998 and the insecurity of XML encryption in the face of a kind of Bleichenbacher attack was shown by Jager et al. in 2012 [16], implementations of PKCS#1 v1.5 persist in deployments and, instead of moving on to inherently more secure encryption, practitioners have deployed a sequence of countermeasures that prevent each attack. Current implementations are not exploitable by prior attacks, but our new attack circumvents all the existing countermeasures to (yet again) break XML encryption. We emphasize that the main takeaway is not that PKCS#1 v1.5 is inherently broken (as already known), but rather that our new side-channel attack framework and PaaS environments provide new opportunities for adversaries.

Bleichenbacher Attacks

PKCS#1 specifies an algorithm for encryption using RSA. Recall that with RSA, one generates a key pair by choosing a modulus $N = pq$ for primes p, q and exponents e, d for which $ed \equiv 1 \bmod \varphi(N)$; the public key is then (N, e) and secret key is (N, d). Let n be the length of N in bytes. With the PKCS#1 v1.5 padding scheme, one encrypts a message M of size m bytes with $m < n - 11$. Letting $r = n - m - 3$, a byte string P of length r is generated in which each byte is randomly selected from $\{0, 1\}^8 \setminus \{0\}$. Letting $X = 00 \| 02 \| P \| 00 \| M$, the ciphertext is then $C = X^e \bmod N$.

To decrypt, one computes $X = C^d \bmod N$ and then checks the padding. A padding error occurs if the first two bytes of X are not $00 \| 02$, there exists a 00 byte among the first 11 bytes, or there does not exist a 00 byte at all after the first two bytes. Decryption fails in such a case.

Bleichenbacher [6] showed how to exploit decryption implementations that notify the sender of a ciphertext when a padding error occurs. Given a challenge ciphertext C^* encrypting some unknown message M, the adversary sends a sequence of adaptively chosen ciphertexts to the oracle, using the response to learn whether the padding is correct or not. Bleichenbacher attacks were first used against XML encryption by Jager et al. [16], with improvements shortly after by Bardou et al. [5]. Below we use the latter's experimental results to estimate timings of the full attack.

Modern implementations attempt to defend against Bleichenbacher attacks by uniform error reporting, in which padding errors are not reported differently from other errors, and by ensuring that decryption runs in essentially the same time when padding errors occur as when not. We will show, however, that our side-channel attack framework can be used in PaaS type settings to re-enable Bleichenbacher attacks despite such countermeasures.

Evaluation in a Public PaaS

We demonstrate this attack in DotCloud. The target of the attack is an active open source project, SimpleSAMLphp (version 1.12, http://simplesamlphp.org/), which implements a SAML-based authentication application in PHP that can be used as either a service provider or an identity provider. It is worth noting that recent SimpleSAMLphp implementations (>v1.9.1) have provided defenses against the traditional Bleichenbacher attack (see changelog v1.9.1) by generating uniform error messages and eliminating timing differences due to invalid padding in session-key decryption. As we show in this section, however, these defenses do not prevent our attack. As we also explain, a recent change in v1.10 of SimpleSAMLphp to a better padding scheme (RSA-OAEP) does not prevent our attack either, as it is possible

Copyright @ 2021 Authors

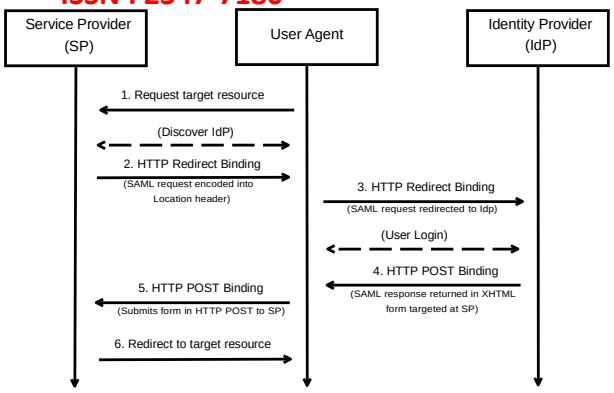

Figure 6: The targeted SAML 2.0 protocol.

to force SimpleSAMLphp to roll back to use PKCS#1 v1.5 instead as long as it is not explicitly disallowed.

A set of protocol bindings [26] and profiles [27] are defined in the SAML 2.0 specification. We investigated the default protocol bindings implemented in SimpleSAMLphp for the web browser SSO profile. As shown in Fig. 6, a web browser acting as a user agent interacts with the service provider (SP) to access resources with the identify provider (IdP) for authentication. Upon receiving a resource access request, the SP issues an <AuthnRequest> message via HTTP redirect binding. The message in XML format is uncompressed and then base64-encoded in the redirect URL query string. After authenticating the user's identity, the IdP will return a SAML response message via HTTP POST binding, in which a signed and encrypted XML file is base64-encoded as a POST parameter which is then sent by the user agent to the service provider using the HTTP POST method.

The padding oracle. In the SAML 2.0 core specification, XML encryption and signing work as follows. The message is first signed, and then encrypted under a symmetric session key. The session key is in turn encrypted. This means that the XML signature is only validated after performing the RSA decryption. While the default padding for encryption is RSA-OEAP, because the padding type is specified in the assertion itself, it is possible to modify the assertion and force the service provider to roll back to PKCS#1 v1.5 padding. The server generates an error whether or not the PKCS padding is correct, to eliminate timing channels. But we will now show how to use the side-channel attack to differentiate between code paths associated with padding errors and non-errors, enabling a Bleichenbacher-style attack.

Attack details and results. The victim account operated a PHP application integrated with the latest stable version of SimpleSAMLphp. The PaaS environment ran OpenSSL version 0.9.8k (which we could not change) and was invoked by the victim application. As such, the adversary monitored the shared library `libcrypto.so`, a component of OpenSSL, and specifically the chunks associated with the basic blocks of function `RSA_padding_check_PKCS1_type_2()` that internally reports a padding error by calling ERR_put_error(). As the padding check procedure is only used during the RSA decryption, other operations do not invoke these functions and it is thus sufficient to monitor only the first chunk of each of the two functions. In practice, though, we found it helped to monitor the first two chunks of the function to

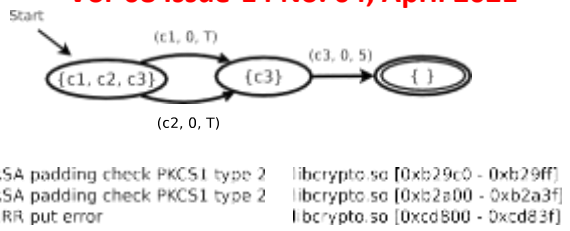

Figure 7: Attack NFA for case study in Sec. 7. Initial state q_0 indicated by "Start" and accepting states indicated with double ovals. T is the maximum Flush-Reload cycles without transitioning before the NFA stops accepting new inputs.

increase the chance of capturing the events. The adversary repeated step 5 in Fig. 6 with manipulated ciphertexts and while applying the side-channel attack framework to detect the occurrence of padding errors.

The attack NFA is shown in Fig. 7. We continuously sent 10,000 requests with conformant padding and 10,000 requests with non-conformant padding, and report the rate of acceptance by the NFA. The results are shown in Table 4. The average time for making one request and padding error detection in this experiment was 0.544 seconds. Optimized attack software could achieve a much higher request rate.

The results indicate that we only had one-sided errors: an execution path accepted by the attack NFA correlated with a non-conformant padding with 100% accuracy. Therefore, the best strategy for the adversary is to send k requests to the padding oracle for each padding, and stop once an execution is accepted by the attack NFA and consider it to be non-conformant padding. If none of the k requests are accepted by the NFA, then no padding error occurred.

This approach will yield no false positives (i.e., false appearances of non-conformant padding). Given the error rate of 12% and assuming errors are independent of ciphertext values, the probability of a false negative (i.e., failure to observe non-conformant padding) in this procedure is $(.12)^k$.

Bardou et al. [5] estimated that their modified Bleichenbacher attack against 2048-bit RSA keys could require about 335,065 queries. We take $(.12)^k$ to be an upper bound on the probability of a false negative for non-conformant padding across k queries. Thus for all queries, an error bound is $335,065(.12)^k$; a choice of $k = 7$ yields an error probability of less than 1% for $335,065 \times 7 = 2,345,455$ total queries. This is about the same number of queries as the original Bleichenbacher attack, and significantly better than, for example, the same estimate of Jager et al. [16] that would require about 85 million queries and only works against old versions of SimpleSAMLphp.

		Attack NFA	
		Accepted	Rejected
Padding	Non-conformant	8800 (88%)	1200 (12%)
	Conformant	0 (0%)	10000 (100%)

Table 4: Confusion matrix for padding error detection. The adversary has only one-sided errors, 12% of the time failing to observe a padding error.

Copyright @ 2021 Authors

8. DISCUSSION

Ethical Considerations

The experiments discussed in Sec. 4–7 were run on production PaaS platforms. As such, our experiments were designed to conform with PaaS provider acceptable use, the law, and proper ethics.

Our attacks only targeted tenants running accounts that we setup and controlled, and no information about other customers was ever collected in our experiments. Our attacker instances did not conduct FLUSH-RELOAD attacks indiscriminately, but rather these were carefully timed to coincide with requests that we initiated to our victim instances. In this way, we limited the risk of our attacker instances observing activities of tenants other than our own.

It is possible that another tenant's programs made use of the same shared executable as our attacker and victim, in which case there is a concern that other tenants might experience degraded memory hierarchy performance as compared to running while co-located with different tenants. Moreover, the acceptable use policies of the clouds on which we demonstrated our attacks include general requirements that we not interfere with other users' enjoyment of their services, which could be interpreted to preclude our demonstrations if they slowed down other tenants substantially as a side effect. We therefore designed our experiments so that they do not cause *undue harm* and, specifically, do not degrade performance of such bystanders significantly more than their performance could be degraded by other workloads.

To ensure no undue harm, we ran local micro benchmarks to evaluate the possible overhead observed by a bystander due to our attacks. For example, to gain confidence that the attack of Sec. 6 would introduce minimal overhead on a bystander, in one container we constructed an attacker application that, in each FLUSH-RELOAD cycle, monitored *every* chunk monitored in *any* state of the attack NFA of Fig. 5. The "bystander" in another container ran a web server hosting a dynamic web page that was artificially constructed to touch (i.e., execute some instruction in) *every* chunk monitored by the adversary before returning. We forced the attacker application and the bystander to share the last level cache in all experiments.

We configured a separate machine in the same LAN to represent a client that repeatedly issued HTTP requests (in the same HTTP session) to the dynamic web page served by the bystander. To measure the bystander's performance degradation resulting from the attacker application's activity, we instrumented the client with httperf and apachebench. In the absence of the attacker application, the client received responses with an average latency of .306ms, and the throughput of the bystander was 461 requests per second. With the attacker application active, the results were nearly identical: an average latency of .307ms and, again, 461 requests per second. Given the conservative nature of these experiments (with the attacker application monitoring more chunks than in the actual attack, and the bystander touching all of them per request), we concluded that our attack demonstrations posed negligible risk to bystanders.

Finally, we attempted to inform affected parties well in advance of publicly disclosing the vulnerabilities documented here. Specifically, we disclosed our findings to selected cloud operators and software vendors directly, and to the CERT (`http://cert.org`) for dissemination more broadly, starting roughly six months prior to publication. Some vendors have made changes to address these issues; e.g., SimpleSAMLphp will blacklist PKCS#1 v1.5 by default in version 1.13.0 [10].

Extending the Attacks

Attacks in IaaS clouds. We believe our NFA-based attack framework can work in IaaS clouds as well, as long as memory de-duplication is enabled and memory pages that contain executables are shared between tenants. For instance, Irazoqui et al. [15] utilized a similar FLUSH-RELOAD side channel (a special case of our NFA-based framework) in a cross-VM context to break AES keys. However, to the best of our knowledge, memory deduplication across VMs is not commonly used in many IaaS clouds (e.g., EC2), which limits the applicability of the FLUSH-RELOAD side-channel attack in those settings.

Multiple victim copies. Multiple copies of the same victim application may co-exist behind a load balancer to increase the throughput and reliability of the services. In such cases, beyond the steps described in this paper, the adversary needs to further determine whether the requests sent to the web server are served by the instance that is co-located with his attacker instance. Such hurdles can be overcome by issuing multiple requests concurrently or co-locating multiple attacker applications with the victim replicas.

Other attack targets. The NFA-based attack framework proposed in this paper provides a general control-flow analysis approach to side-channel observations. We stress its application extends beyond the three examples discussed in the paper. For instance, we believe the latest version (v3.2.8 as of this writing) of the GnuTLS libraries are subject to plaintext-recovery attacks as the co-located adversary can employ the attack framework to construct a padding oracle which reveals the correctness of the CBC mode padding during symmetric key decryption processes. We also believe that an adversary may FLUSH-RELOAD a shared MySQL client library to monitor the victim's SQL query execution (e.g., invocation of `mysql_error()` and a few other functions), thus facilitating blind SQL injection attacks.

Countermeasures

A key question for future research is how to design effective defenses against the attacks enabled within our proposed framework. Various countermeasures to cache-based side channels (not necessarily FLUSH-RELOAD channels) have been proposed in IaaS cloud contexts [31, 4, 36, 18, 41, 19, 35]. However, none of these is applicable to our attacks. We briefly discuss some other possible defenses for our setting.

Mitigating side channels through program analysis. A general countermeasure to control-flow side channels, proposed by Molnar et al. [21] and further explored by Coppens et al. [9], involves the automatic detection of such side channels in source code and their remediation by means of generic source-to-source translation. This approach would thwart the attacks we describe here but does incur significant overhead. A complementary approach involves static analysis of binaries to measure their vulnerability to cache-based side channels [11]; this approach yields only an approximation of the degree of vulnerability of an application, and no countermeasure.

Copyright @ 2021 Authors

Disabling the clflush instruction. It is tempting to disable clflush instructions altogether in PaaS applications to prevent side-channel attacks. However, as clflush is a non-privileged instruction, trapping its execution to mitigate its effect in the privileged software layer, i.e., operating system, is difficult without hardware modification. An alternative solution is to sandbox PaaS applications and specifically disallow the use of clflush in the application code. Still, the adversary might be able to accomplish the effects of a clflush by other means akin to a PRIME-PROBE protocol.

Increasing background noise with more applications. Increasing the number of applications sharing the last-level caches increases background noise in two ways. First, if more processes share the monitored executables, false positive noise increases. Second, as more processes are queued by the CPU scheduler, the last-level caches are less likely to be shared by the attacker and the victim at the same time. However, the security provided by this approach is weak in practice, unless the number of applications is artificially sustained even when there is less real demand.

Disallowing resource sharing. The most general countermeasure for any side-channel attack is to prevent sharing of the exploited resource. In our setting, this would mean disallowing sharing of memory pages that serve as FLUSH-RELOAD attack vectors. An extreme realization would be a prohibition on sharing any memory pages among different users, for instance by duplicating binary files for each user in the OS. Such a defense, however, would increase the memory footprint of each tenant, decreasing the number of tenants that a PaaS provider could provision on a (virtual) machine and reducing machine utilization and service-provider profit. Selective memory sharing promises a more cost-effective approach; sharing of memory pages specifically carrying vulnerable code might then be disallowed. We leave the challenges of identifying, annotating, and protecting such code, as well as the development of alternative defenses, as interesting lines of future research.

Detecting Flush-Reload attacks. An interesting side observation from the experiments conducted in Sec. 8.1 is that the minimal performance degradation induced by the attacks in this paper offers little hope for detecting these attacks. That is, prior studies suggest that cross-tenant side-channel attacks in cloud settings can induce significant performance degradation in victim workloads, as was the case in, e.g., the attacks demonstrated by Zhang et al. [40]. It is possible, therefore, that the attacks of Zhang et al. might be detected by monitoring the performance of the victim application. While our results here do not conclusively rule out the use of victim application performance monitoring to detect the attacks in this paper, they also do not offer much promise for doing so.

However, it might be possible to employ the same type of side-channel analysis to detect FLUSH-RELOAD attacks, similar to the ideas of HomeAlone [39]. That is, a victim might be able to infer the presence of an adversary by means of performing FLUSH-RELOAD monitoring of the cache to detect the FLUSH-RELOAD pattern induced by an adversary's likely choice of NFA. We leave the implementation of this defense as future work.

9. CONCLUSION

We have proposed a general automaton-driven framework to mount cache-based side-channel attacks and demonstrated its potency specifically in PaaS environments. Our three case studies demonstrate that an attacker co-located with a victim can learn sensitive user data, such as the number of distinct items in a shopping cart; perform password-reset attacks against arbitrary users; and break XML encryption in a SAML-based authentication application. The attacks we illustrate are especially significant in some cases in that they bypass existing or proposed side-channel countermeasures. Our shopping-cart attack is immune to defenses proposed for analogous, timing-based side-channel attacks. Our study of RSA private-key decryption re-enables the classic Bleichenbacher padding-oracle attack despite widely deployed countermeasures against remote adversaries.

In sum, we believe our work presents: (1) the first exploration of cache-based side-channel attacks specifically in PaaS environments, and (2) the first report of granular, cross-tenant, side-channel attacks successfully mounted in any existing commercial cloud, PaaS or otherwise, against state-of-the-art applications.

Acknowledgments

This work was supported in part by NSF grants 1065134, 1253870, 1330308, and 1330599, as well as a Google Ph.D. Fellowship for Yinqian Zhang.

10. REFERENCES

[1] F. E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[2] I. Alshanetsky. Top 10 ways to crash PHP. http://ilia.ws/archives/5_Top_10_ways_to_crash_PHP.html. Accessed: 2014-08-17.

[3] B. Argyros and A. Kiayias. I forgot your password: Randomness attacks against PHP applications. In *21st USENIX Security Symposium*, 2012.

[4] A. Aviram, S. Hu, B. Ford, and R. Gummadi. Determinating timing channels in compute clouds. In *2010 ACM workshop on Cloud computing security workshop*, pages 103–108, 2010.

[5] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J.-K. Tsay. Efficient padding oracle attacks on cryptographic hardware. In *Advances in Cryptology — CRYPTO 2012*, pages 608–625. 2012.

[6] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS#1. In *Advances in Cryptology — CRYPTO '98*, pages 1–12, 1998.

[7] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *4th USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, 1993.

[8] A. Bortz and D. Boneh. Exposing private information by timing web applications. In *16th International Conference on World Wide Web*, pages 621–628, 2007.

[9] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *IEEE Symposium on Security and Privacy*, pages 45–60, 2009.

Copyright @ 2021 Authors

[10] J. P. Crespo. Personal communication, June 2014.

[11] G. Doychev, D. Feld, B. Köpf, and L. Mauborgne. CacheAudit: A tool for the static analysis of cache side channels. In *USENIX Security Symposium*, 2013.

[12] S. Esser. Lesser known security problems in PHP applications. In *Zend Conference*, 2008.

[13] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *7th ACM Conference on Computer and Communications Security*, pages 25–32, 2000.

[14] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *2011 IEEE Symposium on Security & Privacy*, pages 490–505, 2011.

[15] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a minute! A fast, cross-VM attack on AES. Cryptology ePrint Archive, 2014.

[16] T. Jager, S. Schinzel, and J. Somorovsky. Bleichenbacher's attack strikes again: breaking PKCS#1 v1.5 in XML encryption. In *Computer Security — ESORICS 2012*, pages 752–769. 2012.

[17] S. Kamkar. phpwn: Attacking sessions and pseudo-random numbers in PHP. In *Blackhat USA*, 2010.

[18] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security Symposium*, 2012.

[19] P. Li, D. Gao, and M. K. Reiter. Mitigating access-driven timing channels in clouds using StopWatch. In *43rd IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, June 2013.

[20] R. P. Mahowald, C. W. Olofson, M.-C. Ballou, M. Fleming, and A. Hilwa. Worldwide competitive public Platform as a Service 2013-2017 forecast (Doc 243315). IDC Inc., November 2013.

[21] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Information Security and Cryptology*, pages 156–168, 2005.

[22] Y. Nagami, D. Miyamoto, H. Hazeyama, and Y. Kadobayashi. An independent evaluation of web timing attack and its countermeasure. In *3rd International Conference on Availability, Reliability and Security*, pages 1319–1324, 2008.

[23] Y. V. Natis. Gartner research highlights Platform as a Service (ID: G00259659). Gartner Inc., 3 February 2014.

[24] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[25] 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.

[26] OASIS. Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0. http://docs.oasis-open.org/security/saml/v2.0/saml-bindings-2.0-os.pdf.

[27] OASIS. Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0. http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf.

[28] 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.

[29] R. Owens and W. Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *30th IEEE International Conference on Performance, Computing and Communications*, pages 1–8, November 2011.

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

[31] H. Raj, R. Nathuji, A. Singh, and P. England. Resource management for isolation enhanced cloud services. In *2009 ACM workshop on Cloud computing security*, pages 77–84, 2009.

[32] 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.

[33] K. Suzaki, K. Iijima, T. Yagi, and C. Artho. Memory deduplication as a threat to the guest OS. In *4th European Workshop on System Security*, April 2011.

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

[35] V. Varadarajan, T. Ristenpart, and M. Swift. Scheduler-based defenses against cross-VM side-channels. In *23st USENIX Security Symposium*, 2014.

[36] B. C. Vattikonda, S. Das, and H. Shacham. Eliminating fine grained timers in Xen. In *3rd ACM workshop on Cloud computing security workshop*, pages 41–46, 2011.

[37] W3Techs. Usage of content management systems for websites. http://w3techs.com/technologies/overview/content_management/all.

[38] Y. Yarom and K. Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. http://eprint.iacr.org/2013/448, 2013.

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

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

[41] Y. Zhang and M. K. Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *2013 ACM Conference on Computer and Communications Security*, pages 827–838, 2013.

Copyright @ 2021 Authors