Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries

Daimeng Wang*, Ajaya Neupane*, Zhiyun Qian*, Nael Abu-Ghazaleh*, Srikanth V. Krishnamurthy*
Edward J. M. Colbert†, and Paul Yu‡
*University of California Riverside. {dwang030, ajaya, zhiyunq, nael, krish}@cs.ucr.edu
†Virginia Tech. ecolbert@vt.edu
‡U.S. Army Research Lab (ARL). paul.l.yu.civ@mail.mil

***Abstract*—Operating systems use shared memory to improve performance. However, as shown in recent studies, attackers can exploit CPU cache side-channels associated with shared memory to extract sensitive information. The attacks that were previously attempted typically only detect the presence of a certain operation and require significant manual analysis to identify and evaluate their effectiveness. Moreover, very few of them target graphics libraries which are commonly used, but difficult to attack. In this paper, we consider the execution time of shared libraries as the side-channel, and showcase a completely automated technique to discover and select exploitable side-channels on shared graphics libraries. In essence, we first collect the cache lines accessed by a victim process during different key presses offline, and then use machine learning to infer the best cache lines (e.g., easily measurable, robust to noise, high information leakage) for a flush and reload attack. We are able to discover effective strategies to classify what keys have been pressed. Using this approach, we not only preclude the need for manual analyses of code and traces — the automated system discovered many previously unknown side-channels of the type we are interested in, but also achieve high precision in terms of inferring the sensitive information entered on desktop and Android platforms. We show that our approach infers the passwords with lowercase letters and numbers 10,000 - 1,000,000 times faster than random guessing. For a large fraction of PINs consisting of 4 to 6 digits, we are able to infer them within 20 and 80 guesses respectively. Finally, we suggest ways to mitigate these attacks.**

I. Introduction

Graphics are pervasively used in modern applications and many applications implement a graphical user interface (GUI) to improve user experience. Graphics rendering is complex and involves multiple processes. For example, on the Linux X architecture, graphics rendering involves many components spanning the kernel, the X-server, the application client, and the device driver. To shelter developers from this complexity, many operating systems provide graphics libraries with simple APIs for applications to render their GUIs.

In this paper, we scrutinize such graphics libraries as a target of side-channel attacks. It is noteworthy that these graphics libraries are provided by operating systems and loaded dynamically by applications and shared across user processes, i.e., different virtual pages are mapped to the same physical pages. This creates an opportunity for a malicious process to infer graphics-related activities of a victim process.

Our intuition of the attack is that the performance of graphics rendering is critical for user experience across a wide range of applications. Consequently, graphics libraries often optimize their execution logic for high performance. For example, when handling simpler graphical content, graphics libraries usually execute a different set of procedures than that for complex content. Even if the same set of subroutines are executed, the execution time can still differ for different inputs (e.g., different characters to render). This processing logic creates a side-channel that can allow attackers to infer a user's input since the execution times of these sensitive graphics operations are both input-dependent and measurable.

However, the practical realization of such side-channel attacks is not trivial, especially since graphics rendering is complex, large, and is characterized by interdependence across multiple processes. On Linux, for example, we find that graphics rendering could involve multiple shared libraries and millions of lines of code. Thus, if attempted blindly, it would take significant time and manual effort to determine where side-channels exist during the rendering process. Moreover, even upon finding such a side-channel, it is difficult to assess whether the leakage is sufficient to reliably recover the target information (e.g., in the presence of measurement noise). To further complicate attacks, measuring the execution time of the victim process regarding the shared graphics library is also challenging. Previous attacks often rely on the attacker *actively triggering* the sensitive procedure (e.g., encryption), and measuring the execution time as many times as they want [16], [30]. When attacking graphics-based applications such as the ones we consider, the attack process can only *passively observe* the execution, leaving fewer attack opportunities.

To this end, we propose a novel method to completely automate the end-to-end realization of practical attacks on graphics rendering. It not only automatically identifies the vulnerable instructions/subroutines whose execution times are input-dependent, but also yields an end-to-end exploit to infer a user's inputs with disturbingly high accuracies. The method measures the information gains from a set of execution times of the subroutines involved in rendering, and identifies those that yield high information gain (i.e., allow effective discrimination between subroutine executions [26]). We then apply a machine learning model that uses these discriminatory subroutines' executions to infer the user's input with high accuracy. We

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23221
www.ndss-symposium.org

demonstrate that the method exposes a leaky CPU side-channel that is practically exploitable and more effective than previously known side-channels of similar types (e.g., [23]). Unlike many previous studies (e.g., [13], [15], [25]), where the researchers relied on manual inspection of source code or instructions to identify vulnerabilities, we provide a systematic methodology that can completely automate the identification of such side-channels (all of which are previously unknown).

We upload the demo videos of our attack on an anonymous website [2]. Moreover, to facilitate the reproduction of the work and future research, we open source the complete source code of the attack at [3].

Our contributions: In brief, we make the following key contributions.

- *Exposing input-dependent execution-time side-channels in graphics libraries:* We systematically investigate this unique type of under-scrutinized side-channels in graphics libraries. By developing a novel and automated methodology, we discover previously unknown and exploitable graphics rendering side-channels on both Ubuntu and Android platforms.
- *Accurate subroutine execution time measurement:* We design and implement an end-to-end CPU cache side-channel attack to measure the execution time reliably in graphics rendering subroutines. We address the technical challenges associated with noises and the implementation of such attacks on the Android platform.
- *Evaluations on real-world applications:* We demonstrate that the discovered side-channel on common graphics libraries can be exploited to infer the passwords with lowercase letters and numbers 10,000 - 1,000,000 times faster than random guessing. For a large fraction of PINs consisting of 4 to 6 digits, we are able to infer them in under 20 and 80 guesses, respectively.

II. Background

This section presents the background necessary for understanding the details of the attack. Specifically, it overviews CPU caches, shared libraries, and CPU side-channel attacks.

A. CPU Caches

Programs often have temporal and spatial locality, i.e., the most recently accessed memory addresses, as well as nearby addresses, are often accessed in the near future. To exploit locality, modern architectures use CPU caches to store recently accessed memory. A CPU cache is often organized into multiple levels with different sizes and access speeds. For example, on Intel CPUs, there are commonly three levels of caches: L1, L2 and L3, with L1 being the fastest and the smallest and L3 being the largest and the slowest. On multi-core CPUs, lower levels of caches (L2 and L3) are often shared among multiple CPU cores.

Modern CPU caches are organized using a *set-associative* policy. This policy divides the cache into multiple cache sets and each cache set contains several cache lines. When the CPU accesses memory, the memory address is indexed into a cache set. The CPU checks all cache lines in this set to identify the presence of the cache line holding the memory address.

Inclusiveness. Lower levels of the cache (L2, L3) can be configured with different inclusion policies. The most common policies are inclusive, exclusive, "non-inclusive non-exclusive" (NINE). Lower levels of the cache are considered inclusive if all the memory blocks present in the upper levels of the cache are also present on the lower levels. The lower levels of the cache are considered exclusive of the higher levels of cache if all the memory blocks present on the higher level of cache are not present on the lower level of cache. If the presence of memory blocks is not strictly inclusive or exclusive with regards to the lower levels of cache, then it is considered NINE.

Prefetching. Programs often access their memory in a predictable order. For example, when a program executes code, the code stored at a lower address is often executed before code stored at a higher address. To further improve performance, modern architectures often implement a cache prefetcher that predicts future memory accesses and loads the predicted contents into the CPU cache before they are actually accessed. For example, Intel CPUs implements a streaming prefetcher which could prefetch up to 20 cache lines ahead of the cache line currently being accessed [5].

B. Shared Libraries

A program library is a collection of subroutines that are available for immediate use by other programs. Since the functionalities provided by these libraries are very commonly required, they are designed to be shared across multiple user programs. These libraries can be mapped to the address space of a user program by the linker when the user program prepares for execution. A library can also be loaded in the middle of an execution of a user program when it explicitly requests that the library be loaded. Regardless of the case, the contents in the library are mapped into the user program's address space.

Operating systems use shared memory to improve the memory utilization efficiency with regards to these libraries. For example, common libraries (`.so` on Linux, `.dll` on Windows) are often shared across all processes linking them. This means that these libraries are loaded into physical memory only once and remain there for the entirety of the OS session. For every process which loads a library, the library will be mapped to a different virtual address in the memory space of the process. However, when different processes access the same library, the same physical memory pages will be accessed. Interestingly, de-duplication can also allow the sharing of libraries across different virtual machines [41].

C. Flush+reload Side-channel Attack

When a process loads a memory block from the cache, the access time is relatively short. If the accessed memory block is not present in the CPU cache, the process will need to load that block from memory (or a lower level of the cache), which is slower. This creates a timing side-channel for attackers to infer the current state of CPU cache, and thus perform attacks on the victim process.

One of the most common forms of cache side-channel attacks is "flush+reload" [47]. Flush+reload attack usually targets shared libraries. The attacker first picks a memory block in the shared libraries and flushes it out of the CPU cache. On Intel CPUs, this can be done using the CLFLUSH instruction.

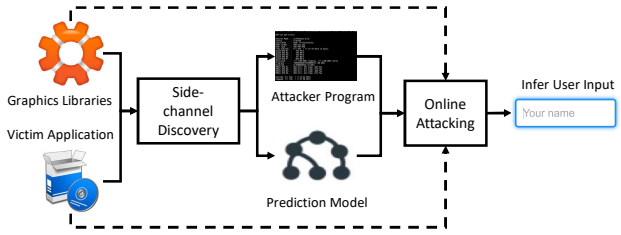


Fig. 1: Attack workflow.

If the victim process executes code that resides in the same memory block, it will load the memory block back into the CPU cache. Subsequently, the attacker accesses (reloads) the memory block, and checks whether it was loaded back into the CPU cache and thereby infers the victim's activity.

On some CPU architectures such as ARM, cache flush instruction is not available to user programs. However, an attacker can still "evict" a cache line by accessing a set of memory blocks that are mapped to the same cache set. Gruss et.al. [33] demonstrated that evict+reload attack can be as effective as flush+reload.

III. Attack Overview

In this section, we provide an overview of the proposed attack starting with the threat model and a high-level description of the underlying intuition. We then briefly describe the individual components of the attack.

A. Threat Model

The attacker's goal is to passively eavesdrop on the victim's CPU cache looking for sensitive information (e.g., PINs or passwords) the victim entered in an application, which we refer to as the victim application. The attacker is assumed to be local, i.e., the attack process is co-located with a victim process on the same physical device and operating system, (e.g., a piece of malware is installed on the victim's system). We assume that the physical device is a multi-core system, and the attack and victim process can run simultaneously on different cores. The attack process can create a few threads that run continuously in the background. No privileges or special permissions are required.

The attacker should also be aware of when a target victim application is launched and when it is in a sensitive state (e.g., login screen) so as to start the side-channel attack. Typically the login screens are shown automatically when the app is first launched. However, even if it is not, we can still infer their presence through attacks similar to prior work [18].

Finally, we assume that an attacker has access to shared graphics libraries used in desktop and mobile applications (these libraries come with the operating system). The attacker has the knowledge of the victim's CPU specification and has access to a device with the same CPU and operating system.

B. Intuition

By studying how graphics libraries work in general, we observe that when a part of the GUI of an application is updated, only the updated part will be rendered. For example, on-screen keyboard applications will often highlight the key being pressed. In this case, only the highlighted key will be rendered on the screen while other parts of the GUI remain the same. Another example is when a user types a character into an input box, the application will only render the typed character. This precise rendering is necessary for input inference.

Our second observation is that when performing text rendering, the graphics library often renders only the pixels representing the text while ignoring the background pixels. Because of this, the rendering of characters with fewer pixels such as "1" and "i", is considerably faster than rendering more complex characters such as "8" and "w".

Given these observations, if an attacker can measure the time it takes for a victim application to render its GUI, she could potentially use this as a side-channel to infer the user's input to the application. In practice, we envision that the attacker conducts offline profiling experiments to map different user inputs to execution times of subroutines related to rendering in shared graphics libraries. Later, she performs online attacks by leveraging the prepared mapping to associate measured execution times back to the user's input.

Without any privilege, an attack process cannot directly measure the program state of the victim. Fortunately, a flush+reload cache side-channel attack can be utilized to indirectly measure graphics rendering time through the shared graphics libraries. Note that previously studied flush+reload attacks [23], [33], [35], [36], [47], [48] were successful at checking only the presence and absence of data in the cache. In our attack, we take a step further to measure the execution time of subroutines.

Measuring the execution time of a shared library subroutine requires an attacker to locate at least a pair of instructions (and their corresponding cache lines). However, this can be challenging and tedious. First of all, graphics libraries are complex and graphics rendering often involves multiple libraries. For example, a typical desktop application on Ubuntu Linux will involve libraries including `libgdk-3.so`, `libcairo.so` and `libpixman.so` which have millions of instructions. Manually going through them is not scalable (especially considering that there are many platforms and library versions). Secondly, even if the attacker finds a good target pair to monitor, it is unclear whether it is reliable and effective in practice due to features such as the cache prefetcher. As a result, we need automated discovery and evaluation of good target cache lines to monitor.

C. Attack Workflow

In this work, we overcome the above challenges and show 1) how we automate the discovery and selection of viable instructions in graphics libraries and 2) how we automatically generate the working exploits of the discovered side-channels.

Our attack is divided into two phases: side-channel discovery (offline) and online attack. Figure 1 shows the general workflow of our proposed methodology. During the side-channel discovery phase the attacker selects a victim application and one (or more) shared graphics libraries to target. The goal of this phase is to (i) analyze the victim application's execution, (ii) discover execution-time side-channels inside the

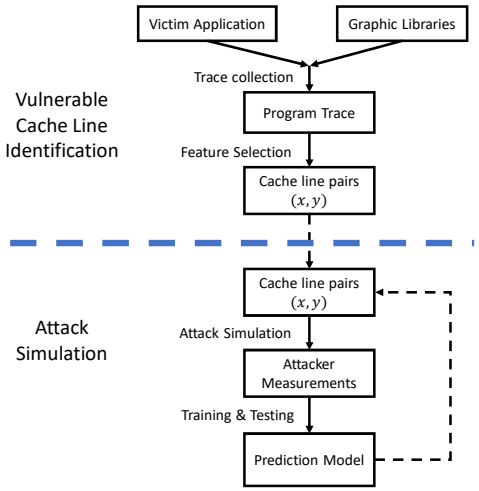


Fig. 2: Flow chart of side-channel discovery.

graphics libraries and, (iii) craft an attack program (malware) that exploits the discovered side-channel iv) generate a prediction model based on the side-channel for future attacks. Detailed descriptions of the side-channel discovery phase will be described in §IV.

To carry out the online attack on user's device, the attacker runs the malware program alongside the victim application, and performs the flush+reload cache side-channel attack on the vulnerable cache lines. The attacker then uses the prediction model to compute the most likely input from the victim. Finally, the attacker could combine the result with other information (such as a password dictionary) to improve the accuracy. We discuss the details of the design and implementation of the online attack in §V and §VI.

IV. Side-Channel Discovery

A. Overview

As discussed in §III-C, in the profiling phase, our goal is to find a pair of cache lines (x, y) (derived from instruction addresses) in the shared graphic libraries as targets for our flush+reload attack. Here we use the term "cache line" as a memory range in the shared library that occupies the same cache line block in the CPU cache (e.g., 0x7f00-0x7f3f). In order to make the attack effective, we need to find the vulnerable cache lines (x, y) such that the time difference between the first access of x to the first access of y (denoted as d_{xy}) "is dependent" on user input. If the attacker is able to reliably measure d_{xy} when users are inputting their PINs or passwords using a cache side-channel attack, they can map the measurements to the original input.

To increase the attack reliability, we need to reduce measurement noise. Measurement noise can originate from two different sources. First, the victim application's behavior combined with modern CPU cache features, (e.g. prefetcher) could significantly decrease the effectiveness of a flush+reload attack, which we refer to as **application noise**. Second, the background processes sharing the same libraries could also introduce noise to the flush+reload attack, which we refer as **system noise**. In side-channel discovery phase, an attacker tries to identify the cache lines least affected by these noises.

Algorithm 1 Feature Extraction and Selection

Input $\mathbb{T}$: Set of traces that are related to user input event.
Output $X = [(IG_{xy}, G_x, G_y)]$: Pairs of cache lines for flush+reload attack, ordered by information gain

$Result \leftarrow \emptyset$
$\mathbb{G} \leftarrow groupGenerate(\mathbb{T})$
for all $G_x, G_y \in \mathbb{G}$ **do**
  Compute d_{xy}, l_x, l_y
  if $d_{xy} > d_{threshold}$ & $l_x < l_{threshold}$ & $l_y < l_{threshold}$ **then**
    if $standard\_deviation(\{d_{xy}\}) \geq std_{threshold}$ **then**
      $IG_{xy} \leftarrow$ Compute information gain of (G_x, G_y)
      Add $(IG_{xy}, G_x, G_y))$ to $Result$
    end if
  end if
end for

Figure 2 captures the process we follow to find the vulnerable cache-line pairs that are suitable for attack. Below, we describe the sequence of steps needed.

1. Vulnerable Cache Line Identification

- The attacker runs the victim application multiple times with different user inputs and collects the program traces for graphics libraries used by the application.
- The attacker uses a feature extraction algorithm to identify potentially vulnerable cache lines from libraries that are least affected by the application noise. See §IV-B for more details.

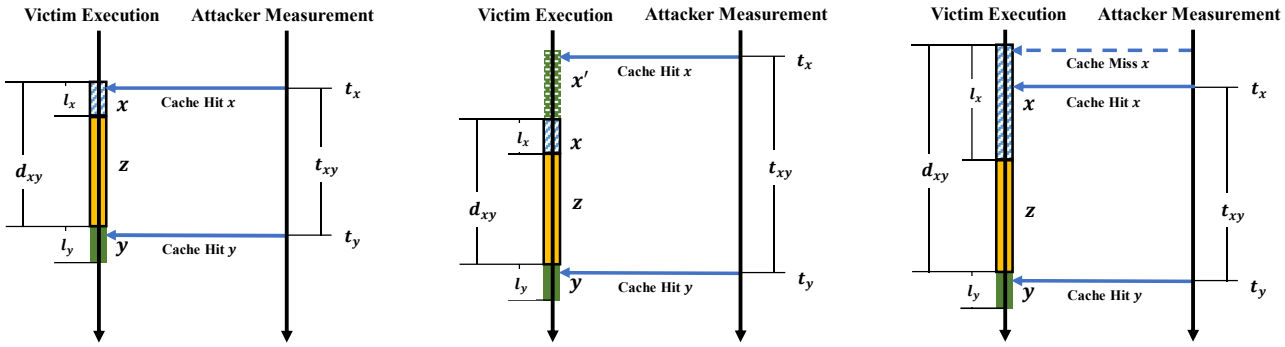
2. Attack Simulation

- For each pair of cache lines (x, y) identified, the attacker runs a simulated offline attack and collects the measurement times.
- The attacker builds a key-press prediction model using the collected measurement times (see §IV-C).
- If the performance of the prediction model is not better than a random guess (e.g., $< 20\%$ for numeric characters 0-9), the attacker selects another pair $\{x, y\}$ and starts over until all the selected cache line pairs are tested. The attacker then picks the cache line pairs with the best performance results. This assures that the chosen cache line pairs are least affected by system noise. See §IV-C for more details.

B. Vulnerable Cache Line Identification

We instrument the victim program and collect the program execution traces with regard to its "cache line accesses" under different user inputs. The cache line trace is a representation of a sequence of instructions being accessed and loaded into CPU instruction by the victim application. For example, on a machine with 64-byte cache lines, if a program executes instructions sequentially from addresses from 0x8020 to 0x80b0, the CPU will load cache lines 0x8000-0x803f, 0x8040-0x807f and 0x8080-0x80bf into the instruction cache. Given a cache line trace, we can easily compute d_{xy} for every possible combination of cache lines x and y.

Next, we design an algorithm to find a pair of cache lines (x, y) such that the distance between the appearances of x and y (denoted as d_{xy}) in the trace varies deterministically based on the user's input, thereby providing a potential side-channel. A graphics rendering operation might contain multiple side-channels, meaning that we might be able to find multiple pairs of (x, y). However, not all of these pairs can be used in a practical flush+reload attack to collect accurate measurements because of the following types of **application noise**.



(a) Successful attack scenario.

(b) Prefetcher noise due to cache line x' prefetching x. Attacker falsely thinks x is being executed by the victim.

(c) Measurement noise due to l_x too large while flush+reload miss a signal. Attacker's measurement t_x is more likely to be inaccurate.

Fig. 3: Attack scenario illustration. Victim executes x, z, y sequentially while execution time of z varies depend on user input.

- **Prefetcher Noise (Figure 3(b)).** The cache lines x and y might be prefetched by the prefetcher, when other memory blocks with slightly lower addresses are being fetched. As a result, we might get false hits with the flush+reload attack.
- **Measurement Noise (Figure 3(c).** x and y may be accessed by the victim application (or prefetched) multiple times during the rendering operation. There are chances that we might miss the first access of x or y (due to a cache line being evicted by victim or background processes before the attacker can reload it) and capture a later access. This will result in an inaccurate measurement with regards to the time interval between x and y.
- **Prediction Noise.** The flush+reload attack has a limited resolution. Therefore, d_{xy} must be large enough to be captured by flush+reload attack. For good prediction, d_{xy} should significantly differ across different user inputs while being consistent with the same input.

Figure 3(a) depicts an example of a victim application's execution trace and one possible attack scenario. In this scenario, the victim application executes code fragments in cache lines x, z, and y sequentially. The execution time of z varies depending on user input, thus creating a side-channel. The execution time of x and y are constant. Therefore, attacker can monitor the time where x and y first appears in CPU cache (flush+reload cache hit) as t_x, t_y respectively and measure $t_{xy} = t_y - t_x$ as a measurement of d_{xy} to infer user input.

However, as depicted in Figure 3(b), if the victim accesses cache line x' before x and accessing x' prefetches x into the CPU cache, the attacker might get a cache hit on x while in fact only the code in x' are being executed. This creates prefetcher noise that makes t_{xy} unable to measure d_{xy} accurately.

Moreover, the "flush" operation of "flush+reload" attack takes some time to complete. So there are chances (although rare) that the victim or some other background process may evict the cache line x or y before the attacker can "reload" the cache line and capture the signal. As a result, the attacker could possibly miss x where it is first loaded into the CPU cache and successfully capture it later on, as depicted in Figure 3(c). This creates a measurement noise whose level is determined by the "lifespan" of x or y (denote as l_x, l_y). If l_x and l_y are small, the attacker might miss the signals x or y completely but when it does capture a signal, the signal is guaranteed to be more accurate. On the other hand, if l_x and l_y are large, the attacker has a better chance at capturing the signals and measuring t_x and t_y, and yet the measured result could be very inaccurate. We find that inaccurate measurements are more detrimental to the attacker than missing measurements as the attacker might unknowingly use them for both training and in the actual attack. On the other hand, missing measurements can be compensated by methods such as having the attack process monitor multiple pairs of cache lines, as we will discuss in §V and §VI. As a result, our feature extraction algorithm favors a relatively small l_x and l_y.

In order to find a good pair (x, y) that achieves a good accuracy given the above constraints, we design and implement an algorithm to extract potential targets from program traces. Algorithm 1 captures this logic which we discuss in the subsequent paragraphs.

The first step in the algorithm is to find these cache line accesses in the shared library that are least affected by the CPU cache prefetcher (i.e. prefetcher noise). For this purpose, we put the cache lines into a set of no-conflict groups $\mathbb{G}$. We ensure that accessing a cache line from one group will have no prefetching effect on a different group. For example, according to Intel's optimization manual [5], the prefetcher can prefetch up to 20 consecutive cache lines. Therefore, we ensure that there is at least $20 * cache\_line\_size$ byte gap between cache lines of different groups.

With all groups $\mathbb{G}$ determined, we then compute d_{xy}, l_x and l_y for all pairs of groups G_x and G_y. We only select pairs of groups (G_x, G_y) where the lifespans of both groups l_x and l_y are below a threshold $l_{threshold}$. This ensures a minimum measurement noise level as discussed earlier.

To reduce the prediction noise, we filter out group pairs where d_{xy} is no greater than threshold $d_{threshold}$, as these are too small to be measurable given the limited resolution of the flush+reload attack. We also need to ensure that the d_{xy} is sufficiently different for different inputs. For this, we first use a coarse-grained filter by checking the standard deviation

of d_{xy} for different inputs. If it is less than a threshold, we exclude the associated pair.

To further reduce the prediction noise and select the best group pairs suitable for the flush+reload attack, we use "information gain" as a metric for selection [26]. The information gain captures the discriminatory value of a cache line pair by quantifying "how much information the cache line pair gives with respect to uniquely separating the users' inputs". In particular, it is a computation of the reduction in entropy. The cache line pairs that perfectly partition the user inputs will have the highest information gain. Our approach ranks the cache line pairs based on their information gain and selects the top-ranked cache line pairs; those that do not add much information will have a lower score and are removed.

Finally, the attacker will need to perform the flush+reload attack on two specific cache lines viz., $\{x, y\}$ instead of two groups $\{G_x, G_y\}$. Therefore, we pick x and y to be the first cache lines that appear in the trace, and belong to their respective groups.

C. Attack Simulation

We now have a list of features (cache line pairs) that are derived from the program trace. However, during the real attack, there can still be unpredictable **system noise**. Hence, the vulnerable cache lines identified in §IV-B might not perform well during the real attack. Therefore, as mentioned earlier in §IV-A, we need to run a simulated offline attack to find the cache line pairs least affected by the **system noise**.

To perform the attack simulation on a cache line pair (x, y) we create an attack process performing the flush+reload attack on one of the CPU cores while running the victim application on another core. Initially, the attack process continuously monitors cache line x and awaits the victim's activity. If the attack process observes a cache hit on cache line x, it records the time of the observation as t_x and immediately switches to monitoring cache line y. If the attacker successfully observes a cache hit on cache line y within a timeout threshold, it records the time of the observation as t_y. If both t_x and t_y are measured, it computes $t_{xy} = t_y - t_x$. Detailed implementations of flush+reload will be described in §V-B and §VI-B.

Model Construction. We use the collected measurements to build our key prediction model. We choose the *Random Forest* [14] algorithm for our key prediction model as these classifiers are robust to outliers, and resilient to irrelevant features [14]. We use sklearn [38] implementation of Random Forest with 100 estimators. The performance of the key prediction model was evaluated with 10-fold cross validation [31]. We compute the true positive rate TPR and the false positive rate FPR. TPR refers to the ratio of the total number of correctly identified instances to the total number of instances present in the classification model; the FPR refers to the ratio of the total number of negative instances incorrectly classified as a positive instance to the total number of actual negative instances. A model with a high TPR and a low FPR is considered good for classification tasks. We measure the performance metrics (of the machine learning model) using different cache-lines pairs as features and select the cache-line pairs which result in the highest TPR and a low FPR for the flush+reload attack.

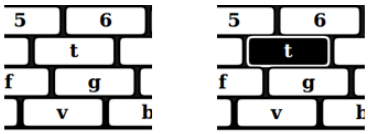


Fig. 4: Partial graphical user interface of Onboard keyboard and the highlighting effect during key-press.

TABLE I: Graphic library versions used in Onboard attack.

Library	Version	Description
libgtk-3	0.1800.9	Multi-platform toolkit for creating GUI.
libgdk-3	0.1800.9	A wrapper around the low-level functions provided by the underlying windowing and graphics systems.
libcairo	2.11400.6	Provides primitives for two-dimensional drawing.
libpixman-1	0.33.6	A low-level software library for pixel manipulation.
libfreetype	6.16.0	Render text onto bitmaps, and provides support for other font-related operation

In §V and §VI we describe the detailed implementations on two different platforms (Linux and Android) and demonstrate their effectiveness with real-world attacks.

V. Attack I: Ubuntu On-screen Keyboard

In this section, we demonstrate our attack on the default on-screen keyboard (used in Ubuntu) to extract a user's password. Such virtual keyboards are necessary in a touch screen scenario. Even without the touch screen, it is recommended that we use such virtual keyboards as a more secure alternative for entering private credentials [11] because it is less prone to various attacks such as keyloggers (e.g., [40], [51]). We evaluate our attacks on a desktop machine with 3.40 GHz Intel Core i7-4770 CPU, which has a 8MB L3 cache and 64 byte cache line size. The GPU on the machine is Nvidia GeForce GT 635 and the main display has a resolution of 1920x1080 pixels. Table I lists the versions of several graphic-related libraries. We study the "Onboard" input method editor (IME) that comes with Ubuntu Desktop 16.04. Figure 4 shows the graphical user interface of Onboard.

When a key is pressed by the user, a "highlight effect" of the pressed key will be rendered on the keyboard. The highlight effect includes (i) a color change of the border, (ii) a fill of the key (iii) a color change with regards to the character represented by the key. Since different characters are composed of different numbers of pixels, we suspect that there are side-channels within the highlight rendering process.

A. Side-Channel Discovery

For our offline trace collection, we use Intel PIN [9] to perform binary instrumentation of the graphics libraries listed in Table I (we identified them by checking the description of the every library loaded in the Onboard keyboard process). The instrumentation allows us to collect full-instruction traces of the keyboard process during a user's key press. We then convert the instruction trace to the corresponding cache line trace of the victim application. Such a trace is devoid of any form of cache pollution. (e.g., from PIN, background process, prefetcher, etc.). From the collected trace we find a very large number of cache lines being accessed by multiple graphics libraries. In `libcairo.so` alone we find 2591 cache lines, which can create around 6.7 million cache line pairs.

TABLE II: Top cache line pairs selected for Onboard IME attack

#	Cache Line	Library	Function Name
1	0x75a40	libcairo.so	_cairo_surface_create_scratch
	0x69e40	libcairo.so	_cairo_scaled_font_map_lock
2	0x69e40	libcairo.so	_cairo_scaled_font_map_lock
	0x41f40	libcairo.so	_cairo_intern_string
3	0x24440	libcairo.so	_cairo_clip_copy_with_translation
	0xbe000	libcairo.so	_cairo_ft_unscaled_font_lock_face
4	0x6b900	libcairo.so	_cairo_path_fixed_approximate_stroke_extents
	0x41700	libcairo.so	_intern_string_pluc
5	0x6a5c0	libcairo.so	_cairo_scaled_font_thaw_cache
	0x41700	libcairo.so	_intern_string_pluc

We then run our feature selection algorithm on the collected trace and attempt to discover side-channels. Using the procedure discussed in §IV-A, we first put all cache lines into groups to eliminate prefetcher noise. In the case of `libcairo.so`, the cache lines form a total of 150 groups, which can create 22350 cache line pairs. The algorithm then filters pairs of cache lines that are not suitable for our attack. The algorithm left us 1488 cache line pairs, ranked by information gain. Interestingly, we find that the majority of the identified cache lines are located at the beginning of functions. This is reasonable since the instructions of functions are organized contiguously in the memory address space. Cache lines located at the beginning of functions are much less likely to be affected by the prefetcher and thus have a better chance of being selected by our algorithm.

Interestingly, we find that out of the few graphics libraries only `libcairo.so` produces good cache line pairs. Upon closer inspection, it turns out that the measured execution time between cache lines pairs from GDK and GTK libraries are not consistent for the same key press. Cache lines corresponding to libfreetype.so are only accessed when a key is first pressed during the lifespan of the Onboard process (we'll discuss exploitation of Freetype library in §VI). Cache lines corresponding to libpixman.so are seldom accessed under the default Linux X server graphics architecture and only play a role under the Wayland architecture (an alternative to X server) as will be discussed in §VIII).

From the cache lines identified in `libcairo.so`, we further perform an offline simulation attack on the top 100 cache line pairs to filter out cache-line pairs that do not produce good results. For each lower-case letter and number, we collect 50 measurements by having one attacker thread running in the background monitoring the selected cache line pairs (as discussed in §IV-C). Next, we build our prediction model with Random Forest classifier on these measurements and evaluate it with a 10-fold cross validation [31]. We select the cache-line pairs and the prediction model with the highest true positive rate. Table II shows the top 5 cache-line pairs that performs best during attack simulation.

To understand the underlying cause of the input-dependent execution time, we choose to inspect the source code of the corresponding address pairs. According to Table II, cache line pair #1 and #2 clearly corresponds to two separate side-channels, one from 0x75a40 to 0x69e40 and another from 0x69e40 to 0x41f40. We find these two side-channels are both part of function `cairo_show_glyphs()`. This function is tasked with computing the rendering result of a given character (glyph) and send the rendering command to Linux X server. The function will first load the pre-computed font data (in which the input character will be rendered) and compute a scaled "pattern" matrix via a series of matrix transformations and multiplications, amounting to the first discovered input-dependent execution time (between 0x75a40 and 0x69e40). This operation involves over 100,000 instructions and its complexity depends on both the font used and the character to be rendered. Next, before contacting X server to render text on the screen, `cairo_show_glyphs()` will first render the computed "pattern" matrix on its own `cairo_surface_t` struct. This operation takes around 10,000 instructions to complete and its complexity also depends on the input character, thus creating the second side-channel (between 0x69e40 to 0x41f40). We find that cache line pair #4 and #5 also cover the second side-channel while cache line pair #3 covers both side-channels.

In theory, one cache-line pair is sufficient. In practice though, due to measurement noises (and the possibility of missing signals during flush+reload), we simultaneously monitor two cache-line pairs for redundancy. We further use them to train a new machine learning model for the actual attack. Empirically, we observe that the addition of more features (more cache line pairs) improves the prediction accuracy. However, monitoring too many cache line pairs results in much noisier measurements (as flush+reload itself as well as context switches create noises). Ideally, for the best attack resolution, an attacker thread should only monitor one cache line pair on a CPU core. On our desktop machine, we find that the measurement becomes noisy when we run the flush+reload attack to monitor more than two cache-line pairs. We select the top cache line pair #1 and #2 from Table II for our side-channel attack. Luckily, we also find that when a user presses a key on Onboard, there will be two rendering operations. The first operation will render the highlighted key while the second operation will reverse the highlight effect. Both operations will generate a signal measurable using the selected cache line pairs. Therefore, the attacker can capture 4 measurements for a single key press.

B. Flush+reload Evaluation

We first evaluate the flush+reload attack in a controlled environment. We create a controlled victim process that accesses a cache line x and collects the timestamp of the access as the ground truth. Meanwhile, the attack process tries to capture the time when x is accessed by the victim. To achieve this, the attack process "loads" a memory address in the cache line x. Then, the attacker thread executes the CLFLUSH instruction on the address just read. We find that we need to introduce a short delay after the CLFLUSH instruction and before the next load operation in order to capture the signal reliably. Similar to previous works [23], we use `sched_yield()` to introduce this delay.

The more delay we introduce, the better the chance that the flush+reload attack will be able to capture the signal. However, a larger delay also means a longer time for each round of flush+reload; this reduces our measurement resolution. Figure 5 demonstrates the effect of `sched_yield()`. In our attack, we set the number of `sched_yield()` calls to 3, which is the setup that can reliably capture the signal and maintain a reasonable measurement resolution at the same time.

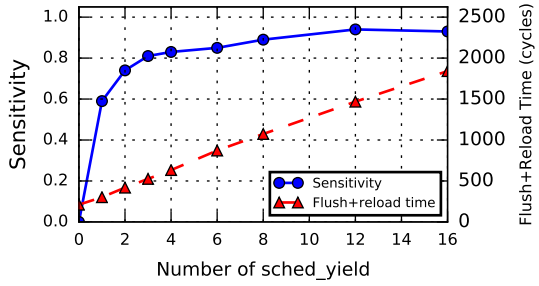


Fig. 5: Effect of $sched\_yield()$

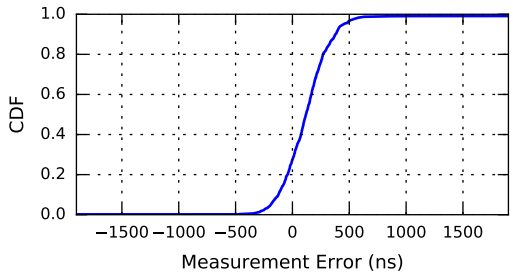


Fig. 6: CDF distribution of measurement error.

Figure 6 presents the cumulative distribution function (CDF) of the measurement error. We see that in 90% of the experiments, the attacker is able to measure the execution time of the target operation with an absolute error of less than 700 ns. Given that a single round of flush+reload takes around 500 ns, we conclude that our cross-process execution time measurement is accurate.

C. Password Inference Attack

We then run our attack to infer a user password inputted using the Onboard keyboard. We have two attacker threads each monitoring the selected cache line pairs (in Table II) respectively, while a user is inputting passwords. After the attacker thread collects the measurement data during the user's input, we use the trained prediction model generated via the offline simulation to predict the user passwords based on the measurements.

For demonstration, we only consider lower-case letters and numbers for the password field inputted using the Onboard keyboard. We choose the list of most common passwords as our dictionary [6]. This dataset contains 10,000 unique passwords, with 9984 of them composed of lower-case letters and numbers. Note that for all our attacks except for one augmented with dictionary, we assume the attacker has no knowledge of the passwords in the dataset.

1) Single Character Prediction Accuracy:

Single Login Attempt. First, we test our password prediction capability when we capture a single login attempt from the user (we might be able to observe multiple attempts over time). For each character, we perform the attack multiple times and test whether the attacker can correctly predict the character within a certain number of guesses. Figure 7 demonstrates the single-character prediction accuracy for numeric characters. We observe that we can reach 90% accuracy in 10 guesses for all numeric characters. Some characters (e.g., "2") can be predicted more accurately than characters like "0" and "4"; this can be helpful in inferring PINs. However, predicting lower-case letters are much more difficult. With 10 guesses we can only reach 70% accuracy. Some characters such as "u", "v", and "w" requires more than 30 guesses while characters like "i", "o", and "y" can be predicted pretty accurately in fewer guesses. When it comes to lower-case letters some of them look similar and the measured time difference is often overwhelmed by the noise. As we will show next, the limited measurement resolution and noise are the main reasons why the accuracy of observing a single login is not as high.

Repeated Login Attempts. A password is often reused or repeatedly inputted by the user. As a result, an attacker has the opportunity to obtain measurements of the user's repeated login attempts using the same password. This gives the attack more potency i.e., by combining multiple measurements together the attacker can make better predictions (e.g., because the noise can be corrected). Suppose the user inputs password $p = b_1 b_2 ... b_n$ N times. At the jth instance, suppose our prediction model generates confidence vectors $C_{j,1}, ..., C_{j,2}, C_{j,n}$. We can then combine these guesses by simply adding the prediction confidence values together. In other words, we let $C_{comb,k} = \Sigma_{x=1}^{N} C_{x,k}, k = 1, ..., n$ to be the aggregated confidence vector of these measurements.

We study the per-character prediction accuracy when we have 10 and 20 measurements of the same character, as shown in Figure 8. We can see that more measurements leads to higher prediction accuracy. With 10 measurements, all numeric characters and most lower-case letters can be predicted with 100% accuracy within 4 guesses. Characters like "a", "d" and "v" are a little harder to predict than other characters, yet the attacker can still predict them with 100% accuracy within 8 guesses. With 20 measurements, the prediction accuracy is slightly improved over 10 measurements. 17 out of 36 characters can be predicted perfectly in the first guess.

2) Multi-Character Prediction Accuracy: To guess the entire password correctly, we need every character to be guessed correctly. This means that the total number of guesses is bound by the prediction accuracy of the worst character. Specifically, if the worst character takes k guesses to achieve a 100% accuracy, then the total number of required guesses will be k^n where n is the number of characters in the password. This is because we cannot know a priori which character is the worst during guessing and will have to exhaust all possibilities.

In the best case when the attacker can observe 20 login attempts, the number of needed guesses is then 5^n, which is a drastic improvement from the original 36^n. Figure 9(a) further illustrates our results. We observe that having multiple measurements of the same password significantly improved our password-guessing effectiveness. On average we need 1000 times fewer guesses to infer a password compared to the case where we used a single input, and 1,000,000 times faster over a random guess. Finally, 40% of the passwords are guessed within 100 attempts.

Augmentation with Dictionary Attack. Password characters are often non-independent events. To better utilize this dependency between password characters, attackers often use dictionary attacks to reduce the search space. Being able to guess the password in fewer attempts is useful as an account may be locked after a few failed login attempts. Our attack can

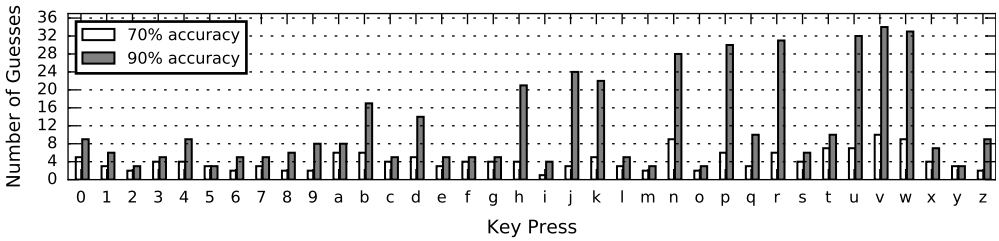


Fig. 7: Onboard IME: Single login attempt. Number of guesses required to reach 70% and 90% accuracy.

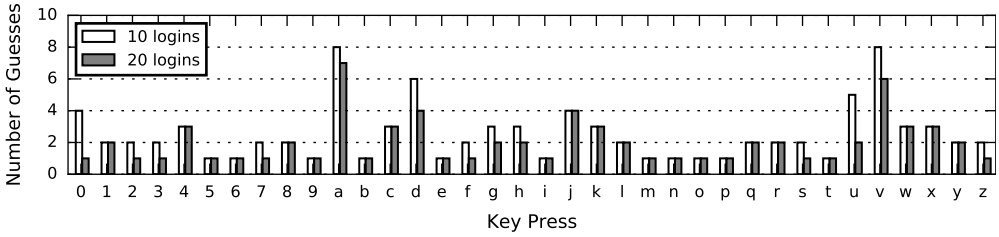
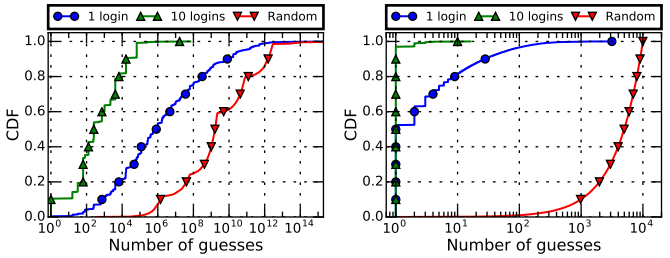


Fig. 8: Onboard IME: Repeated login attempts. Number of guesses required to reach 100% accuracy.

TABLE III: Example dictionary-assisted password guessing attack for password "hello".

Input	Confidence Vector (Partial)							
	e	h	i	j	l	o	s	y
h	0.0	0.39	0.0	0.23	0.0	0.0	0.0	0.03
e	0.21	0.0	0.0	0.0	0.0	0.0	0.0	0.03
l	0.0	0.0	0.05	0.0	0.37	0.0	0.07	0.0
l	0.0	0.0	0.05	0.0	0.37	0.0	0.07	0.06
o	0.0	0.0	0.0	0.0	0.0	0.15	0.0	0.0

Rank	Dictionary Words	Confidence Value
1	hello	$0.39 + 0.21 + 0.37 + 0.37 + 0.15 = 1.49$
2	jelly	$0.23 + 0.21 + 0.37 + 0.37 + 0.0 = 1.18$
3	hills	$0.39 + 0.0 + 0.37 + 0.37 + 0.0 = 1.13$
4	holly	$0.39 + 0.0 + 0.37 + 0.37 + 0.0 = 1.13$



(a) Password inference attack without dictionary.

(b) Password inference attack with password dictionary.

Fig. 9: Onboard IME: The cumulative distribution function for the number of guesses needed to infer passwords.

work very well in conjunction with dictionary attacks to boost its effectiveness. Let $p = b_1 b_2 ... b_n$ be the target password and $C_1, ..., C_n$ be the prediction confidence vectors (single-input or combined). Let $conf(a, C)$ be the confidence value of character a in confidence vector C.

For each n-character password $w = w_1 ... w_n$ from the dictionary, we can compute the confidence c_w of w being the correct password, to be $c_w = \Sigma_{i=1}^{n} conf(w_i, C_i)$. We then rank all possible n-character passwords based on the confidence values c_ws, and try them in order. If the correct guess has the kth-highest confidence score, we will need k guesses.

Table III shows an example of dictionary attack. When user inputs password "hello", the attacker would be able to obtain five different measurements each corresponding to one character in the password. The attacker will then use the prediction model to generate five confidence vectors $C_1, ..., C_5$, where each character in the password alphabet is given a confidence value. Next, the attacker look up all the passwords in the dictionary with a length of 5 and compute its confidence value. The attacker finally ranks all the 5-character passwords based on their confidence values and proceeds to use them as guesses in order. In this example, "hello" has the highest confidence compared to the other 5-length passwords in the dictionary. As a result, the attacker can guess "hello" in the first attempt. Our algorithm ranks the guesses based on confidence relating to each character. Note that we use the sum of individual confidences instead of product as we do not want to penalize a guess for one poorly predicted character. For example, in Table III, suppose the correct password is "jelly". Multiplying the confidences will result in a low rank of the word "jelly" for merely a poorly predicted "y".

We compare our password guessing approach with random brute-force guessing with dictionary. With the random approach, the attacker needs to look up all the passwords in the dictionary and guess them in random order. Therefore, the number of guesses required with the random approach is a random number between 1 and the total number of passwords in the dictionary (10,000 in our case).

Figure 9(b) demonstrates the result. With one input of the password, our approach can guess 50% of the passwords correctly in the first attempt and 80% the passwords within 10 guesses. On the other hand random guesses can hardly crack anything within 10 guesses. If attacker can measure 10 login attempts, 95% of the passwords can be cracked in the first guesses. We do acknowledge that this result is dependent on the size of dictionary we are considering.

TABLE IV: Cache line pairs selected for CapitalOne attack

#	Cache Line	Library	Function Name
1	0x176a80	libskia.so	SkScalerContext_FreeType_Base ::generateGlyphImage
	0xef440	libskia.so	SkMask::getAddr
2	0x109e40	libskia.so	SkGlyph::computeImageSize
	0xca8c0	libskia.so	SkAAClipBlitter:: SkAAClipBlitter

VI. Attack II: Android Application

In this section, we demonstrate our attack on two Android applications to extract a user's password and PIN, respectively. For this demonstration, we use a Nexus 6P running Android 8.0, as the victim's device. The victim apps are the CapitalOne banking and the Reliance Global Call [7] (an app similar to the Skype) app. With both apps, every time the app opens, it will require a user to input his/her username and password/pin. For security reasons, the exact password and pin are not normally rendered on the screen and often replaced with stars or dots. However, to prevent input errors, Android, by default, makes any input character visible for one second before masking it.

A. Side-Channel Discovery

For offline trace collection, we instrument the graphics libraries (e.g. `libskia.so`, `libfreetype.so`, etc.) from Android AOSP source code, recompiled these libraries and loaded them on to the victim device. The instrumentation targets various functions and records the timestamp each time an instrumented function is called. The instrumentation will introduce a small overhead but it does not thwart the feature selection algorithm.

We run our feature selection algorithm on the collected traces and identify top function pairs suitable for attack. We then translate the function names to cache line addresses and perform the offline simulation attack to filter out cache-line pairs that do not produce good results. Normally, the shared library is stripped and thus, not all function names to cache line translations will be straightforward. However, we find that graphics libraries such as `libskia.so` contain enough symbols for us to translate most of the selected functions. Similar to Onboard attack, we then select the top two cache-line pairs for our attack. Table IV shows the two cache-line pairs and the corresponding functions.

Upon analyzing the function calls made between the identified cache lines, as well as the source code of related graphics libraries, we find that the cache lines actually measure an exploitable side-channel in the *font translation* process from the standard Android Freetype library. Font translation is long in duration and happens whenever an application attempts to render any text (e.g., characters) for the first time. It works by translating the character representation into the graphical representation (glyphs). In Freetype library this is accomplished via `FT_Outline_Decompose()`. Depending on the font and the input character, `FT_Outline_Decompose()` will invoke a series of functions (such as `gray_set_cell()`, `gray_hline()`) to compute the translation result [4]. Each Android process keeps the translation result in its memory space. Thus, the next time the same character with the same font type and size is to be rendered, there is no need to execute font translation again. As a result, the attacker can only capture the first appearance of each character using the this side-channel. However, many sensitive applications, CapitalOne and Reliance Global Call included, require the user's login information upon start. At this point, the font translation has not been performed for most characters (especially the font type and size in the password box), leaving a window for attacker to extract sensitive information.

It is worth mentioning that the software keyboard app similar to those on Linux, is also an excellent target for this attack. However, the software keyboard is maintained in a dedicated long-running process and the font translation results are cached throughout the lifetime of the process. This means that most of the characters should have been translated by the time the attack is launched.

Note that there are also side-channels discovered on Android that get exercised every time when a key is pressed (regardless of whether it is a pressed for the first time). Unfortunately due to the limited measurement resolution on ARM, those are not practically exploitable.

B. Evict+reload Implementation

The ARM architecture poses several technical challenges. First of all, the ARM architecture does not include a CLFLUSH instruction. As a result, we cannot perform a flush+reload attack. Fortunately, we can still perform the evict+reload attack by creating a set of memory blocks (eviction set) that can evict the target cache line. Our implementation is similar to Gruss et.al.'s implementation [33].

Eviction is slower than the CLFLUSH instruction. On the LG Nexus 5X, each round of evict+reload takes 10μs to 13μs, while on our x86 desktop machine each round of flush+reload takes only around 0.5μs. As a result, the attack resolution of evict+reload is lower than that of flush+reload. This means that our attack will be less effective for faster graphic operations such as text rendering. The font translation process takes a long time to compute and thus, the evict+reload attack is able to exploit the timing side-channel associated with it.

The evict+reload attack is easier to implement when the attack process can read its page table and know the physical address of each memory block in the eviction set. However, later Android versions no longer grant user process read access to its page table. Still, Oren et.al., [35] demonstrated that evict+reload can still be performed without an attacker knowing the virtual-to-physical address mapping.

Another challenge in realizing the evict+reload attack on ARM is that many ARM CPUs do not have an instruction-inclusive shared L2 cache. In addition, Green et.al. [20] have shown that there are other features in the ARM CPU implementation that make an attack more difficult. Fortunately, most ARM CPUs are cache coherent. When a process accesses a cache line not currently cached in its own core, the CPU will try to fetch it from other cores in case other processes are accessing it. If successful, the resulting access will be much faster than access from memory. This feature has been exploited in recent works [33], [48] and we also rely on it.

However, the phones we tested (Nexus 5X and 6P) have an additional challenge in that the attacker and victim cannot have a shared L2 cache. Nexus 5X adopts big.LITTLE technology

[8]. It has 2 Coretex-A57 "big" cores sharing a single unified L2 cache and 4 Coretex-A53 "LITTLE" cores sharing 2 unified L2 caches. It appears that only the cores among big and LITTLE have the proper cache coherency protocol that can be exploited, which requires the attacker and victim to be on different classes of cores. Further, we find it interesting that the eviction takes significantly more time on the big cores (likely due to their cache replacement policy). As a result, we pin the attack threads on the "LITTLE" cores while leaving the victim on the "big" cores.

The net effect of this setup is that we are unable to evict the victim's L1 and L2 cache. As a result, when checking for the victim accesses relating to the cache line targeted by the evict+reload attack, the attacker will keep getting cache hits for a short while even when the victim is not accessing it anymore. Fortunately, our attack only focuses on capturing the target cache line's first appearance. As a result, the attack is not affected by the lack of victim-core eviction. Since L1 and L2 have limited sizes, shortly after the victim finishes executing the target function, the cache will be evicted by other functions of the victim automatically.

C. Password and Pin Inference

Password Inference. We attack a user who is launching a new CapitalOne application process and inputting a password from the common password list [6] (the same 10,000 password dataset as used in §V). During this process we have the attack threads running in the background and collecting measurements. Next, we use the trained prediction model to predict the user's key press based on the measurements collected.

In this attack, we measure the font translation time when rendering a character on the screen. This process only happens when the character is inputted for the first time (i.e., even if a character is inputted multiple times, the attacker can only perform its timing measurements when it is first inputted). In addition, the CapitalOne app renders "Username" and "Password" in respective input boxes on startup. Therefore, the attacker will not be able to measure the font translation time for characters "U", "P", "a", "d", "e", "m", "n", "o", "r", "s" and "w" (as they share the same font type and size with the actual password). Interestingly, the attack can still be performed with these restrictions. The attacker can easily use evict+reload attack to monitor when a key-press is happening and combine the result with the font translation timing measurement. If the attacker observes a key press but does not get a timing measurement, the attacker can infer that the inputted character is either in the list of pre-rendered characters or something that the user had previously inputted.

Table V shows an example attack scenario when "hello" is inputted. The steps are as follows:

- User inputs "h". "h" is not pre-rendered by the CapitalOne app and the attacker will be able to get a measurement. Suppose our prediction model guesses it as "0".
- User inputs "e", which is pre-rendered. The attacker will detect a key press but will have no measurement of the font translation time. The attacker will simply guess it as one of the rendered characters (including the pre-rendered characters and the inference made with regards to the first character i.e., "0"). Suppose we guess it as "e".

TABLE V: Example password guessing attack for password "hello".

Input	Potential Guesses	Guess	Description
h	0,p,h	0	Predicted by model.
e	a,d,e,m,n,o,r,s,w,0	e	Pre-rendered or same as 1st character.
l	l,1,7	l	Predicted by model
l	a,d,e,m,n,o,r,s,w,0,l	0	Pre-rendered or same as 1st/3rd character.
o	0,o	o	Cannot be "0" since its guessed as 1st character.

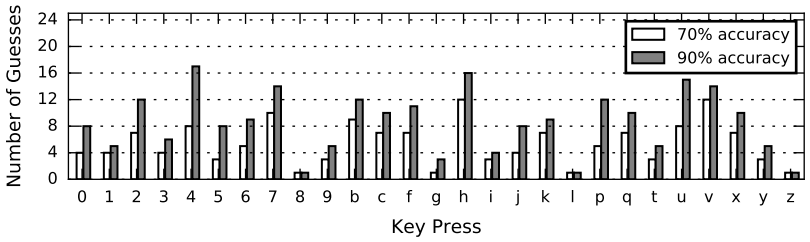
- User inputs "l", which is not pre-rendered. The attacker will be able to get a measurement. Suppose we guess it as "l".
- User inputs "l", which is already rendered during the previous keypress. The attacker will simply guess it as one of the rendered characters (including all pre-rendered characters and the guesses of the 1st and 3rd characters viz., "0" and "l"). Suppose we guess it as "0".
- User inputs "o", which is not pre-rendered. Attacker will be able to get a measurement and guess it to be either "0" or "o". However, since the belief is that "0" is already rendered, it is inferred that this character can only be "o".

Figure 10(a) shows the number of guesses required to predict individual characters. We omit the pre-rendered characters from this figure as the process of guessing them is simply a random selection from a list of known pre-rendered characters. We assume that the attacker can measure 10 login attempts from the user. As we discussed in §V, having more measurements improves the attacker's accuracy drastically. According to Figure 10(a), in general, the result is worse than the Linux Onbard keyboard attack primarily due to the limited measurement resolution of evict+reload. Still, most characters can be predicted with 90% accuracy within 10 guesses. Some characters such as "4", "7", "h", "u", "v" are harder to guess accurately as they are often confused with other characters. For example, "b" and "q" have a similar shape, thus their font translation times are close to each other. It's very difficult to distinguish them. On the other hand, characters such as "1", "8", "l" and "z" can be predicted very effectively due to their rather unique shape and font translation time.

We compute the number of guesses needed to infer a complete password using our attack model on the CapitalOne application, similar to the previously discussed Ubuntu Onboard keyboard attack §V-C. We compare our password inference capability with a random brute-force guessing in Figure 11(a). We see that using our attack model, the number of guesses needed to infer the password is 10,000 times less than the number of guesses needed with random guessing.

We also compare our password guessing capability in conjunction with a dictionary. As discussed in §V-C2, attackers often use dictionary attacks to reduce the search space. We present our results in Figure 11(b). When an attacker is able to capture one login attempt of the password, our approach can infer 30% of the passwords in the first guess and 70% the passwords within 10 guesses. With 10 login attempts captured, our approach can guess 60% of the passwords in the first guess and 90% the passwords within 10 guesses.

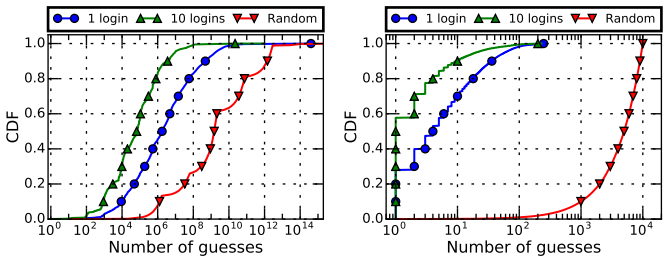
PIN Inference. In this section, we exploit the login process of the Reliance Global Call application [10] (a very popular



(a) Number of guesses needed to infer an input character correctly.

(b) Number of guesses needed to infer a digit.

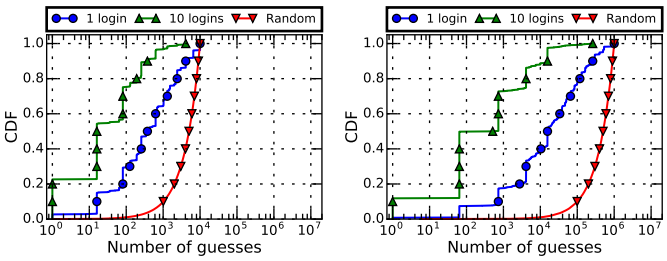
Fig. 10: Android: Number of guesses need to infer each character correctly.



(a) Password inference attack without dictionary.

(b) Password inference attack with password dictionary.

Fig. 11: Android: The cumulative distribution function for the number of guesses needed to infer passwords.



(a) 4-digit PIN

(b) 6-digit PIN

Fig. 12: Android: The cumulative distribution function for the number of guesses needed to infer the PINs of different lengths.

VoIP app), with a similar PIN input box and graphics libraries. Reliance asks users to enter a PIN every time the app opens. We generate random PINs of length 4 and 6 for us to infer.

Similar to the password inference attack, we measure the font translation time when rendering a digit on the screen using the same cache line pairs listed in Table IV. We assume that the attacker is able to capture the user inputting the PINs multiple times. These measurements are then fed to the prediction model generated in the offline simulation.

Figure 10(b) shows the number of guesses required to guess an individual digit with 100% accuracy. We observe that with 20 user logins, 4 of the digits can be inferred in one attempt and 8 of the digits can be inferred within 2 attempts with 100% accuracy. We next use our prediction model to guess the entire PIN of 4-digit and 6-digit lengths. From Figure 12(a) we see about 20% of the 4-digit PINs can be cracked in a single attempt, and 55% of the 4-digit PINs can be cracked in 20 attempts or less. From Figure 12(b) we notice that more than 50% of the 6-digit PINs can be inferred in less than 80 attempts. The overall attack success rates in both cases are five to six orders of magnitude better than the random brute-force attack. Consistent with previous results, we observe that the prediction rate improves as the number of observed login attempts increases.

D. Attacking Built-in Keyboards

To improve security, some banking apps have built-in keyboards for entering passwords/PINs. The goal of this design is to prevent malicious keyboard applications from recording user's input. Ironically, these keyboards are more vulnerable to our font-translation attack for the following reasons: 1) The banking app is not a long-running processes like a regular keyboard app. This means every time it launches anew, font translation need to be redone. 2) The keyboard often have a unique font type and size. There is no pre-rendered characters.

We perform our attack on CITIC mobile banking app [1], a popular banking app that has over 8 million downloads. The app uses its built-in keyboard for password entering. Whenever a key is pressed, a pop-up echo containing a enlarged version of the pressed key will be rendered. The password input box, however, will not render inputted character. The rendering procedure is different from CapitalOne and Reliance. Therefore, we perform side-channel discovery as previously discussed and selected the following pairs of functions: 1) `SkScalerContext_FreeType_Base::generateGlyphImage` (`libskia.so`) and `GpuPixelBuffer::map` (`libhwui.so`); 2) `gray_set_cell` (`libft2.so`) and `FontRenderer::cacheBitmap` (`libhwui.so`).

Similar to our CapitalOne attack, we assume that the attacker can measure 10 login attempts from the user. The measurements we obtain from CITIC however, are noisier than the CapitalOne. So we evaluated our key prediction model with Random Forest and Boosting [19] algorithms. Boosting outperforms Random Forest for CITIC app and we selected it to build a key-prediction model. Boosting is a machine learning ensemble algorithm that convert weak learners (high bias, low variance) to strong ones and is resistant to overfitting [19].

Figure 13 shows the character guessing accuracy for our attack on CITIC app. Comparing with Figure 10(a), we first find our attack on CITIC can capture all 26 lower-case characters. There's no pre-rendered characters like CapitalOne because the CITIC keyboard uses a unique font. Additionally, We find that the prediction accuracy for CITIC attack is better than CapitalOne attack, where the majority of the characters can be

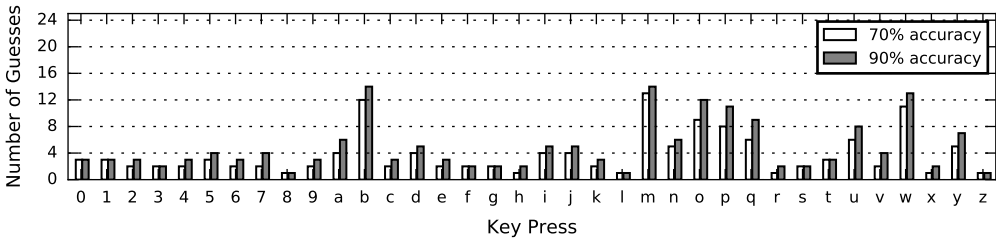


Fig. 13: CITIC mobile banking: Number of guesses needed to infer an input character correctly.

predicted with 90% accuracy within 4 guesses. This because the rendered echo in CITIC keyboard is much larger than the text echo displayed in CapitalOne password input box, making evict+reload more effective in capturing the stronger signal.

VII. Related Work

There has been an abundance of existing work on CPU cache side-channel attacks, most of which target encryption keys. For example, Tromer et al. [42] and Osvik et al. [36] demonstrate how the prime+probe attack could be used to break AES by locating memory accesses in the AES lookup table. Zhang et al., [50] [49] show that prime+probe attacks can even cross VM boundaries and perform cross-tenant attacks on PaaS (Platform as a service) clouds. In our work, we use the flush+reload attack, which was first introduced by Yuval et al. [47]. In their work, the authors demonstrate that the flush+reload attack can be used to attack encryption applications such as GnuPG. Gulmezoglu et al., [25] design and showcase an improved flush+reload attack on AES.

A closely related work by Gruss et al. [23] proposes a cache template attack which aims to discover input-dependent cache line accesses automatically in shared libraries. This methodology does not leverage unique characteristics of graphics libraries and therefore misses the opportunity to measure input-dependent execution time. Interestingly, we attempt to repeat the same experiment against the GDK library (also a graphics library) and are longer able to find input-dependent cache line accesses. Our attack differs in that we do not focus on unique memory accesses; instead, we rely on the difference in execution times to infer user inputs. Furthermore, we demonstrate that given a conducive application, we can achieve much better accuracy. On average we reduce the entropy per character for a random password from $log_2(36) = 5.16$ to ($log_2(10) = 3.32$ – $log_2 3 = 1.58$), while Gruss et al. only reduce the entropy from $log_2(26) = 4.7$ to $log_2(16) = 4$.

We perform our attack on ARM CPUs with an instruction non-inclusive last-level cache. This has been shown to be a minor hurdle for cache side-channel attacks. There have been solutions for both ARM and x86 that primarily leverage the cache coherency protocol among different last-level caches. For example, it was recently shown that the latest non-inclusive last-level cache employed by x86 CPUs can also be attacked [46]. Several researchers demonstrate the possibility of performing cache side-channel attacks on ARM. Zhang et al., [48] design and implement a return-oriented flush+reload attack on ARM. Both works utilize the cache coherency policy to monitor victim applications' instruction cache access and we adopted a similar methodology. Gruss et al., [33] perform a systematic study on cache side-channel attacks on the ARM architecture, discussing both the prime+probe and the flush+reload attacks. Our work builds on similar ideas. We had to also deal with the fact that there is no shared L2 cache between the attacker and victim.

In addition to the prime+probe and the flush+reload attacks, researchers also explore other potential side-channel attacks related to the CPU cache. Gruss et al., [22] propose the flush+flush attack, which utilizes the timing side-channel of CLFLUSH instruction under different cache states. Unfortunately, on our machine this attack did not work as reliably as the flush+reload attack. In addition, they also discover a timing side-channel on prefetch instructions [21] and utilize this side-channel to perform address translation towards breaking ASLR. Lee et al., [32] and Wang et al., [45] show that branch predictors also contain side-channels that can be used to attack secure systems such as SGX.

There are studies on the automated discovery of cache side-channels. For example, Gorka et al., [28] utilize dynamic taint analysis to locate cache side-channels in crypto libraries. Wang et al., [44] model the cache behavior and use symbolic execution in conjunction with their model to discover crypto-related vulnerabilities. These approaches discover only the presence and absence of a unique cache line access to decide if any side-channel is present. Here, we investigate a unique execution-time-based side-channel in shared libraries. Further, we not only automatically discover such side-channels but also generate and evaluate the exploit automatically.

There are other orthogonal research studies exploiting different types of side-channels (e.g., keystroke sounds [12], [27], [51]; electromagnetic waves [43]; vibrations [34] etc.). All these side-channel attacks need physical proximity to the target device. Researchers have also introduced new types of attacks to guess sensitive user input using motion sensors [17], [37] on smartphones or inter-keystroke timings [40]. However, the success of such attacks is dependent on individual users' typing habits. Unlike these attacks, our attack does not need access to a physical device nor is dependent on user behavior. Additionally, defenses mitigating inter-keystroke timing attacks [39] cannot prevent our attack.

VIII. Discussion and Future Work

Measurement challenges: Our attack is sensitive to measurement resolution and noise. For example, we found that with the Linux Wayland architecture, there is a side-channel in `libpixman.so` when it renders text for applications such as Gedit, the Gnome Terminal. We also find that `libskia.so` can perform text rendering pixel-by-pixel for Android applications (in addition to the font translation that is triggered only for the first time a key is rendered within the same

process). However, out measurement resolution is too low to perform a reliable attack on these operations. One interesting observation we had is that the larger the font size, the more time it takes for rendering. We will explore other opportunities where the measurement resolution is sufficient (e.g., larger fonts are used). Another direction is to integrate this attack with scheduler-based attacks [24], [29], [50] to slow down the victim process which in turn allows the measurement to be more precise. Finally, is is worth noting that most previous attacks against crypto libraries assume a large number of observations [50] (as the encryption can be triggered by the attacker) which makes their attack much easier compared to ours (from the measurement challenge perspective).

Capital letters and special characters: In reality, many passwords must include capital letters and special characters. Adding these characters directly to our prediction model would no doubt introduce confusion and reduce accuracy. Fortunately, often times these characters can only be entered by pressing special keys (e.g. shift, ?123) or perform special actions (e.g. long-presses). These operations will change the keyboard status (e.g. switch to special characters keyboard) and generate unique signals (e.g. redraw the keyboard) that can potentially be detected by the attacker. Therefore, attacker can train different prediction models for capital letters and special characters. Upon detecting a keyboard status change, attacker can then switch to the corresponding prediction model.

Mitigations: There are several steps one could take to help mitigate the side-channel attack that we discover. Since the attack relies on the flush+reload attack, disabling user access to the CLFLUSH instruction and high resolution timers will make the attack much more difficult. Although the attacker could still evict a cache line by accessing a set of memory blocks, it will be much slower and result in an attack with much lower resolution. Since performance is critical to our attack, this is likely to reduce the accuracy significantly. Disabling the high-resolution timer will also affect our attack. However, the attacker could choose to implement its own timer [33] and perform the attack as described.

A general solution to timing side-channels is to make the rendering constant time irrespective of the input at the cost of rendering performance. Nevertheless, even if one decides to implement this mitigation, one will need to overcome the challenge of locating the input-dependent subroutines. Here, our profiling model can significantly help in identifying these locations and thus can be useful for defense as well.

Finally, to prevent attacks on applications such as CapitalOne, a user can turn off the "Make password visible" option (which is on by-default) under the Android settings. This makes password inputting less convenient but prevents any text rendering operation for passwords. In addition, application developers can choose to forcibly pre-render all characters with the same font as the password, thus eliminating the font translation process during user's password input.

Extensions: In this paper, we focus on using our attack to discover side-channels in graphic libraries. In principle, our attack on input-dependent execution times could find previously unknown side-channels in all kinds of shared libraries. As a future work, we will consider extending our attack beyond graphic libraries such as crypto and audio processing libraries, hardware drivers, etc. In addition, currently we are mainly studying applications on Linux and Android. We can also port our attack to other platforms such as Windows and MacOS.

Additionally, its worth noting our attack implicitly obtains the inter-keystroke timing for free via flush+reload. This allows us to combine our attack with existing inter-keystroke timing attacks [40] to further improve its effectiveness, which we will consider in future studies.

Finally, we currently only use our intuition to exploit a general type of feature - measuring the execution time between two addresses. There might exist other type of features (e.g. execution order, time series of multiple addresses, etc.) in the program execution trace that could potentially be identified using more sophisticated techniques such as deep learning. This is another interesting direction for future studies.

IX. Conclusions

In this paper, we discover a previously unknown type of potent side-channel that allows an attacker to use the flush+reload attack to perform cross-process timing measurements on sensitive functionalities inside shared graphic libraries. The attack facilitates the inference of a user's keystrokes when the typed keys are rendered on the screen. The attack hinges on utilizing machine learning techniques to discover execution-time based side-channels inside graphic libraries. We have completely automated the discovery of such side-channels and even the generation of exploits. We validate that our attack is viable on real-world applications on multiple platforms and demonstrate its high accuracy in predicting user input in practice, which affects a large user population of the considered applications. Finally, we suggest ways to mitigate this exploit.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback on this paper. This work was partially supported by the U.S. Army Research Laboratory Cyber Security Collaborative Research Alliance under Cooperative Agreement Number W911NF-13-2-0045. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to re-produce and distribute reprints for Government purposes, notwithstanding any copyright notation hereon.

References

[1] "Citic bank app," https://shouji.baidu.com/software/11576280.html.

[2] "Demo videos of attack." https://sites.google.com/view/swtwmyc/home.

[3] "Github repository," https://github.com/CacheAttackGraphics/KeystrokeAttack.

[4] "Glyph hell: An introduction to glyphs, as used and defined in the freetype engine." http://chanae.walon.org/pub/ttf/ttf_glyphs.htm.

[5] "Intel 64 and ia-32 architectures optimization reference manual)," https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[6] "Most common passwords list." http://www.passwordrandom.com/most-popular-passwords/page/1.

[7] "Reliance global call," https://www.relianceglobalcall.com/.

[8] "Technologies — big.little arm developer." https://developer.arm.com/technologies/big-little.

[9] "Pin - a dynamic binary instrumentation tool," https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, 2012.

[10] "Reliance global call," https://www.relianceglobalcall.com/features/free-international-app-to-app-calls, 2012.

[11] "10 ways to stay safe from cyber attacks," https://medium.com/all-technology-feeds/10-ways-to-stay-safe-from-cyber-attacks-6449219ceb54, 2017.

[12] D. Asonov and R. Agrawal, "Keyboard acoustic emanations," in *IEEE Symposium on Security and Privacy*, 2004.

[13] N. Benger, J. Van de Pol, N. P. Smart, and Y. Yarom, "ooh aah... just a little bit: A small amount of side channel can go a long way," in *International Workshop on Cryptographic Hardware and Embedded Systems*, 2014.

[14] L. Breiman, "Random forests," *Machine learning*, 2001.

[15] B. B. Brumley and R. M. Hakala, "Cache-timing template attacks," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2009.

[16] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03, 2003.

[17] L. Cai and H. Chen, "Touchlogger: Inferring keystrokes on touch screen from smartphone motion." *HotSec*, 2011.

[18] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into your app without actually seeing it: UI state inference and novel android attacks," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[19] Y. Freund, R. E. Schapire *et al.*, "Experiments with a new boosting algorithm," in *Icml*, vol. 96. Citeseer, 1996, pp. 148–156.

[20] M. Green, L. Rodrigues-Lima, A. Zankl, G. Irazoqui, J. Heyszl, and T. Eisenbarth, "Autolock: Why cache attacks on ARM are harder than you think," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[21] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016.

[22] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds. Cham: Springer International Publishing, 2016.

[23] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15, 2015.

[24] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games – bringing access-based cache attacks on aes to practice," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11, 2011.

[25] B. Gülmezoğlu, M. S. Inci, G. Irazoqui, T. Eisenbarth, and B. Sunar, "A faster and more realistic flush+ reload attack on aes," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*, 2015.

[26] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of machine learning research*, 2003.

[27] T. Halevi and N. Saxena, "A closer look at keyboard acoustic emanations: random passwords, typing styles and decoding techniques," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012.

[28] G. Irazoqui, K. Cong, X. Guo, H. Khattri, A. K. Kanuparthi, T. Eisenbarth, and B. Sunar, "Did we learn from LLC side channel attacks? A cache leakage detection tool for crypto libraries," *CoRR*, 2017.

[29] M. Kayaalp, D. Ponomarev, N. Abu-Ghazaleh, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016.

[30] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology — CRYPTO '96*, N. Koblitz, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996.

[31] R. Kohavi *et al.*, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Ijcai*, 1995.

[32] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[33] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[34] P. Marquardt, A. Verma, H. Carter, and P. Traynor, "(sp) iphone: decoding vibrations from nearby keyboards using mobile phone accelerometers," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011.

[35] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox - practical cache attacks in javascript," *CoRR*, 2015.

[36] A. Osvik, Dag, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers Track at the RSA Conference*. Springer, 2006.

[37] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang, "Accessory: password inference using accelerometers on smartphones," in *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, 2012.

[38] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, 2011.

[39] M. Schwarz, M. Lipp, D. Gruss, S. Weiser, C. Maurice, R. Spreitzer, and S. Mangard, "Keydrown: Eliminating keystroke timing side-channel attacks," in *NDSS*, 2018.

[40] D. X. Song, D. Wagner, and X. Tian, "Timing analysis of keystrokes and timing attacks on ssh," in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, ser. SSYM'01, 2001.

[41] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory deduplication as a threat to the guest os," in *Proceedings of the Fourth European Workshop on System Security*, 2011.

[42] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, 2010.

[43] M. Vuagnoux and S. Pasini, "Compromising electromagnetic emanations of wired and wireless keyboards," in *Proceedings of the 18th USENIX Security Symposium*, 2009.

[44] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "Cached: Identifying cache-based timing channels in production software," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[45] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, 2017.

[46] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *2019 IEEE Symposium on Security and Privacy*.

[47] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack," in *USENIX Security Symposium*, 2014.

[48] X. Zhang, Y. Xiao, and Y. Zhang, "Return-oriented flush-reload side channels on arm and their implications for android devices," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016.

[49] Y. Zhang, A. Juels, K. Reiter, Michael, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[50] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, 2012.

[51] L. Zhuang, F. Zhou, and J. D. Tygar, "Keyboard acoustic emanations revisited," *ACM Transactions on Information and System Security (TISSEC)*, 2009.