KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks

Michael Schwarz*, Moritz Lipp*, Daniel Gruss*, Samuel Weiser*,
Clémentine Maurice†, Raphael Spreitzer*, Stefan Mangard*
{michael.schwarz, moritz.lipp, daniel.gruss, samuel.weiser, raphael.spreitzer, stefan.mangard}@iaik.tugraz.at
clementine.maurice@irisa.fr
*Graz University of Technology, Austria †Univ Rennes, CNRS, IRISA, France

***Abstract*—Besides cryptographic secrets, software-based side-channel attacks also leak sensitive user input. The most accurate attacks exploit cache timings or interrupt information to monitor keystroke timings and subsequently infer typed words and sentences. These attacks have also been demonstrated in JavaScript embedded in websites by a remote attacker. We extend the state-of-the-art with a new interrupt-based attack and the first Prime+Probe attack on kernel interrupt handlers. Previously proposed countermeasures fail to prevent software-based keystroke timing attacks as they do not protect keystroke processing through the entire software stack.**

We close this gap with *KeyDrown*, a new defense mechanism against software-based keystroke timing attacks. *KeyDrown* injects a large number of fake keystrokes in the kernel, making the keystroke interrupt density uniform over time, *i.e.*, independent of the real keystrokes. All keystrokes, including fake keystrokes, are carefully propagated through the shared library to make them indistinguishable by exploiting the specific properties of software-based side channels. We show that attackers cannot distinguish fake keystrokes from real keystrokes anymore and we evaluate *KeyDrown* on a commodity notebook as well as on Android smartphones. We show that *KeyDrown* eliminates any advantage an attacker can gain from using software-based side-channel attacks.

I. Introduction

Modern computer systems leak sensitive user information through side channels. Among software-based side channels, information can leak, for example, from the system or microarchitectural components such as the CPU cache [12] or the DRAM [43]. Historically, side-channel attacks have exploited these information leaks to infer cryptographic secrets [31], [41], [58], whereas more recent attacks even target keystroke timings and sensitive user input directly [17], [40], [43].

In general, keystroke attacks aim to monitor when a keyboard input occurs, which either allows inferring user input directly or launching follow-up attacks [50], [60]. In particular, mobile devices may expose this information through sensor data, but practical mitigations [48] have already been proposed. Furthermore, restrictions (on the procfs) have already been implemented in Android O [14], [25] and are likely to be upstreamed to the main Linux kernel. Consequently, attackers are left with side channels to obtain keystroke timings. Especially microarchitectural attacks allow monitoring memory accesses with a granularity of single cache lines, and thus also allow recovering keystroke timings with a high accuracy.

Keystroke timing attacks are hard to mitigate, compared to side-channel attacks on cryptographic implementations. Indeed, attacks on cryptographic implementations can be mitigated with changes in the algorithms, such as making execution paths independent of secret data. On the contrary, user input travels a long way, from the hardware interrupt through the operating system and shared libraries up to the user space application. In order to detect a keystroke, an attacker just needs to probe a single spot in the keystroke path for activity.

In the general case, keystrokes are non-repeatable low-frequency events, *i.e.*, if the attacker misses a keystroke, there is no way to repeat the measurement. However, an attacker that explicitly targets a password field can record more timing traces when the user enters the password again. While these traces have variations in timing, due to the variance of the typing behavior, it allows an attacker to combine multiple traces and to perform a more sophisticated attack. This makes attacks on password fields even harder to mitigate.

State-of-the art defense mechanisms [14], [25], [48] only restrict access to the system interfaces providing interrupt statistics [10], [60], and do not address all the layers involved in keystroke processing. Therefore, these defenses do not prevent all software-based keystroke timing attacks. We first demonstrate two novel side-channel attacks to infer keystroke timings, that work on systems where previous keystroke timing attacks are mitigated [14], [25]. The first attack uses the `rdtsc` instruction to determine the execution time of an interrupt service routine (ISR), which is then used to determine whether or not the interrupt was caused by the keyboard. The second attack uses Multi-Prime+Probe on the kernel to determine when a keystroke is being processed in the kernel.

Based on these investigations and state-of-the-art attacks, we identify three essential requirements for successful elimination of keystroke timing attacks on the entire software stack. In the presence of the countermeasure:

† During the work the author was affiliated with Graz University of Technology, Austria

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23027
www.ndss-symposium.org

1) Any classifier based on a single-trace side-channel attack may not provide any advantage over a random classifier.
2) The number of side-channel traces a classifier requires to detect all keystrokes correctly must be impractically high.
3) The implementation of the countermeasure may not leak information about its activity or computations.

Based on the identified requirements, we present *KeyDrown*, a new defense mechanism against keystroke timing attacks exploiting software-based side channels. *KeyDrown* covers the entire software stack, from the interrupt source to the user space buffer storing the keystroke, both on x86 systems and on ARM devices. We cover both the general case where an attacker can only obtain a single trace, and the case of password input where an attacker can obtain multiple traces. *KeyDrown* works in three layers:

1) To mitigate interrupt-based attacks, *KeyDrown* injects a large number of fake keyboard interrupts, making the keystroke interrupt density uniform over time, *i.e.*, independent of the real keystrokes. Prime+Probe attacks on the kernel module are mitigated by unifying the control flow and data accesses of real and fake keystrokes such that there is no difference visible in the cache or in the execution time.
2) To mitigate Flush+Reload and Prime+Probe attacks on shared libraries, *KeyDrown* runs through the same code path in the shared library for all, fake and real, keystrokes.
3) To mitigate Prime+Probe attacks on password entry fields, *KeyDrown* updates the widget buffer for every fake and real keystroke.

We evaluate *KeyDrown* on several state-of-the-art attacks as well as our two novel attacks. In all cases, *KeyDrown* eliminates any advantage an attacker can gain from the side channels, *i.e.*, the attacker cannot deduce sensitive information from the side channel.

We provide a proof-of-concept implementation, which can be installed as a Debian package compatible with the latest long-term support release of Ubuntu (16.04). It runs on commodity operating systems with unmodified applications and unmodified compilers. *KeyDrown* is started automatically and is entirely transparent to the user, *i.e.*, requires no user interaction. Although our countermeasure inherently executes more code than an unprotected system, it has no noticeable effect on keystroke latency. Finally, we also define what *KeyDrown* cannot protect against, such as word completion lookups or immediate forwarding of single keystrokes over the network.

Contributions. The contributions of this work are:

1) We present two novel attacks to recover keystroke timings, that work in environments where previous attacks fail [14], [25].
2) We identify three essential requirements for an effective countermeasure against keystroke attacks.
3) We propose *KeyDrown*, a multi-layered solution to mitigate keystroke timing attacks.[1]
4) We evaluate *KeyDrown* and show that it eliminates all known attacks.

[1]The code and a demo video are available in a GitHub repository: https://github.com/IAIK/keydrown.

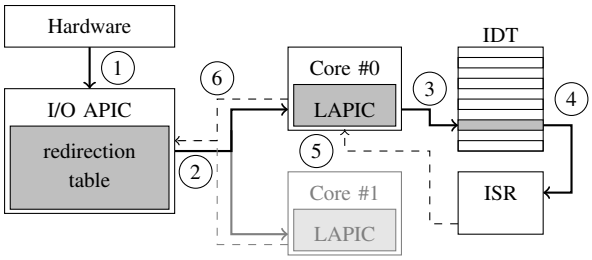

Fig. 1. Linux interrupt handling on x86.

Outline. The remainder of the paper is organized as follows. In Section II, we provide background information. In Section III, we introduce our novel attacks and define requirements a defense mechanism has to provide to successfully mitigate attacks. In Section IV, we describe the three layers of *KeyDrown*. In Section V, we demonstrate that *KeyDrown* successfully mitigates keystroke timing attacks. In Section VI, we discuss limitations and future work. We conclude in Section VII.

II. Background

In this section, we provide background information on interrupt handling as well as on software-based side channels that leak keystroke timing information.

A. Linux Interrupt Handling

Interrupt handling is one of the low-level tasks of an operating system and thus highly architecture and machine dependent. This section covers the general design of how interrupts and their handling within the Linux kernel work on both x86 PCs and ARMv7 smartphones.

1) Interrupts on x86 and x86_64: Figure 1 shows a high-level overview of interrupt handling on a dual-core x86 CPU. Interrupts are handled by the Advanced Programmable Interrupt Controller (APIC) [22]. The APIC receives interrupts from different sources: locally and externally connected I/O devices, inter-processor interrupts, APIC internal interrupts, performance monitoring interrupts, and thermal sensor interrupts. On multi-core systems, every CPU core has a local APIC (LAPIC) to handle interrupts. All LAPICs are connected to one or more I/O APICs which handle the actual hardware interrupts. The I/O APICs are part of the chipset and provide multi-core interrupt management by distributing the interrupts to the LAPICs as described in the ACPI system description tables [37].

Interrupt-generating hardware, such as the keyboard, is connected to an I/O APIC pin (1). The I/O APIC uses a redirection table to redirect hardware interrupts and the raised interrupt vector to the destination LAPIC (2) [21]. In the case of multiple configured LAPICs for one interrupt, the I/O APIC chooses a CPU based on task priorities in a round-robin fashion [6].

The LAPIC receiving the interrupt vector fetches the corresponding entry from the Interrupt Descriptor Table (IDT) (3) which is set up by the operating system. The IDT contains an offset to the Interrupt Service Routine (ISR) for every interrupt vector. The CPU saves the current CPU flags and jumps to the interrupt service routine (4) which then handles the interrupt.

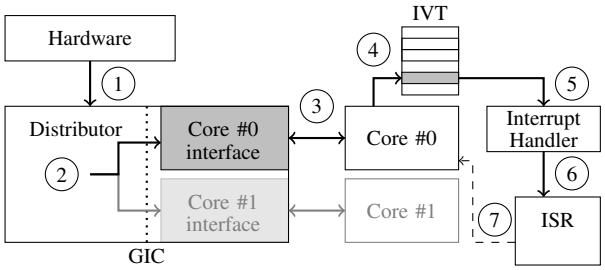

Fig. 2. Linux interrupt handling on ARM.

After processing, the interrupt service routine acknowledges the interrupt by sending an end-of-interrupt (EOI) to the LAPIC (⑤). It then returns using the `iret` instruction to restore the CPU flags and to enable interrupts again. The LAPIC forwards the EOI to the I/O APIC (⑥) which then resets the interrupt line to enable the corresponding interrupt again.

2) Interrupts on ARM: Figure 2 shows a high-level overview of interrupt handling on a dual-core ARMv7 CPU. On ARM, interrupts are handled by the General Interrupt Controller (GIC). The GIC is divided into two parts, the distributor, and a CPU interface for every CPU core [3]. Every interrupt-generating device is connected to the distributor of the GIC (①). The distributor (②) schedules between CPU interfaces according to the interrupt's affinity mask.

When a CPU interface receives an interrupt, it signals it to the corresponding CPU core (③). The core reads the interrupt number from the interrupt acknowledge register to acknowledge it. If the interrupt was sent to multiple CPU interfaces, all other CPU cores receive a spurious interrupt, as there is no more pending interrupt.

When receiving an interrupt, the CPU finishes executing the current instruction, switches to IRQ mode, and jumps to the IRQ entry of the Interrupt Vector Table (IVT) (④). The IVT contains exactly one instruction to jump to a handler function (⑤). In this handler function, the OS branches to the Interrupt Service Routine (ISR) corresponding to the interrupt number (⑥).

When the CPU is done servicing the interrupt, it writes the interrupt number to the End Of Interrupt register (⑦) to signal that it is ready to receive this interrupt again [2].

B. Microarchitectural Attacks

CPU caches are a small and fast type of memory, buffering frequently used data to speed-up subsequent accesses. There are typically three levels of caches in modern x86 CPUs, and two levels in modern ARM CPUs. The last-level cache is typically shared across cores of the same CPU, which makes it a target for cross-core side-channel attacks. On Intel x86 CPUs, the last-level cache is divided into one slice per core. The smallest unit managed by a cache is a cache line (typically $64\,\mathrm{B}$). Modern caches are set-associative, *i.e.*, multiple cache lines are considered a set of equivalent storage locations. A memory location maps to a cache set and slice based on the physical address [20], [34], [59].

Flush+Reload. Flush+Reload [18], [58] is a technique that allows an attacker to monitor a victim's cache accesses at a granularity of a single cache line. The attacker flushes a cache line, lets the victim perform an operation, and then reloads and times the access to the cache line. A low timing indicates that the victim accessed the cache line. While very accurate, it can only be performed on shared memory, *i.e.*, shared libraries or binary code. Flush+Reload can neither be performed on dynamic buffers in a user program nor on code or data in the kernel. Gruss et al. [17] presented cache template attacks as a technique based on Flush+Reload to automatically find and exploit cache-based leakage in programs.

Prime+Probe. Prime+Probe [31], [41], [42] is a technique that allows an attacker to monitor a victim's cache accesses at a granularity of a cache set. The attacker primes a cache set, *i.e.*, fills the cache set with its own cache lines. It then lets the victim perform an operation. Finally, it probes its own cache lines *i.e.*, measures the access time to them. This technique does not require any shared memory between the attacker and the victim, but it is difficult due to the mapping between physical addresses and cache sets and slices. As Prime+Probe only relies on measuring the latency of memory accesses, it can be performed on any part of the software stack. It is possible to perform Prime+Probe on dynamically generated data [30] as well as kernel memory [41]. Preventing Prime+Probe attacks is difficult due to the huge attack surface and the fact that Prime+Probe uses only innocuous operations such as memory accesses on legitimately allocated memory, as well as timing measurements.

DRAMA. Besides the cache, the DRAM design also introduces side channels [43], *i.e.*, timing differences caused by the DRAM row buffer. A DRAM bank contains a row buffer caching an entire DRAM row ($8\,\mathrm{KB}$). Requests to the currently active row are served from this buffer, resulting in a fast access, whereas other requests are significantly slower. DRAM side-channel attacks do not require shared memory and work across CPUs of the same machine sharing a DRAM module.

C. Keystroke Timing Attacks

Keystrokes from Keystroke Timing. Keystroke timing attacks attempt to recover what was typed by the user by analyzing keystroke timing measurements. These timings show characteristic patterns of the user, which depend on several factors such as keystroke sequences on the level of single letters, bigrams, syllables or words as well as keyboard layout and typing experience [44]. Existing attacks train probabilistic classifiers like hidden Markov models or neural networks to infer known words or to reduce the password-guessing complexity [49], [50], [60].

Most keystroke timing attacks exploit the inter-keystroke timing, *i.e.*, the timing difference between two keystrokes, but according to Idrus et al. [19] combinations of key press and key release events could also be exploited. Pinet et al. [44] report inter-keystroke interval values between $160\,\mathrm{ms}$ and $200\,\mathrm{ms}$ for skilled typists. Lee et al. [27] define the values depending on whether a text sequence was trained or entered for the first time, resulting in inter-keystroke intervals between $125\,\mathrm{ms}$ and $215\,\mathrm{ms}$ with a variance between $43\,\mathrm{ms}$ and $106\,\mathrm{ms}$, again for trained and untrained text sequences.

Keystroke Timing from Software. A direct software side channel for keystroke timings is provided through OS interfaces, such as instruction pointer and stack pointer information leaked through `/proc/stat`, and interrupt statistics leaked through `/proc/interrupts` [60]. As the instruction pointer and stack pointer information became too unpredictable, Jana and Shmatikov [23] showed that CPU usage yields much more reliable information on keystroke timings. Diao et al. [10] demonstrated high-precision keystroke timing attacks based on `/proc/interrupts`. However, these attacks are not possible anymore in Android O [14], [25], as access to these resources has been restricted.

Vila et al. [53] recovered keystroke timings from timing differences caused by the event queue in the Chrome browser. Based on the native attack we present in Section III-B, Lipp et al. [29] implemented the same attack in JavaScript. They recovered keystroke timings and identified user-typed URLs. They also showed that users can be distinguished based on this attack.

Gruss et al. [17] demonstrated that Flush+Reload allows distinguishing specific keys or groups of keys based on key-dependent data accesses in shared libraries. Ristenpart et al. [46] demonstrated a keystroke timing attack using Prime+Probe with a false-negative rate of $5\,\%$ while measuring 0.3 false positive keystrokes per second. Pessl et al. [43] showed that it is possible to use DRAM attacks to monitor keystrokes, e.g., in the address bar of Firefox. However, this attack only works if the target application performs a massive amount of memory accesses to thrash the cache reliably on its own.

III. Keystroke Timing Attacks & Defenses

Due to the amount of code executed for every keystroke, there are many different side channels for keystroke timings. In this section, we introduce our two novel attacks and compare them to state-of-the-art keystroke timing attack vectors, in order to understand the requirements for effective countermeasures. Finally, we derive three requirements for countermeasures to be effective against keystroke timing attacks.

The requirements are defined based on precision and recall of side-channel attacks. The *precision* is the fraction of true positive detected keystrokes in all detected keystrokes. If the precision is low, the side channel yields too many false positives to derive the correct keystroke timings. The *recall* is the fraction of true positive detected keystrokes in all real keystrokes. If the recall is low, *i.e.*, the side channel misses too many true positives, inter-keystroke timings are corrupted too. A standard measure of accuracy is the *F-score*, *i.e.*, the geometric mean of precision and recall. An F-score of 1 describes a perfect side channel. An F-score of 0 describes that a side channel provides no information at all.

Note that there is only a limited number of keystroke time frames that can be reliably distinguished by an attacker, due to the typing speed and the variance of inter-keystroke timing (cf. Section II-C). A keystroke timing attack providing nanosecond-accurate timestamps is actually only providing the binary information in which time frames a keystroke occurred. Hence, we can compare side-channel-based classifiers to binary decision classifiers for these time frames.

TABLE I. State-of-the-art Software-based Keystroke Timing Attacks and their Targets.

	Kernel	Shared library	User process
Interface-based	✓ [10], [23], [60]	✗	✗
Timing-based	✓ **ours**	✗	✗
Flush+Reload	✗	✓ [17]	✗
Prime+Probe on L1	✓ [46]	✓ [46]	✓ [46]
Prime+Probe on LLC	✓ **ours**	✓ **ours**	✓ **ours**
DRAMA	✗	✗	✓ [43]

An *always-zero oracle* which never detects any event has an F-score of 0. An *always-one oracle* which "detects" an event in every possible time frame, *i.e.*, a large number of false positives, no false negatives, and no true negatives, is a channel which provides zero information. Similarly, a *random-guessing oracle*, which decides for every possible time frame whether it "detects" an event based on an apriori probability, also provides zero information. For 8 keystrokes and 100 possible time frames per second, the F-score for the always-one oracle is 0.15 which is strictly better than the F-score of the random-guessing oracle (0.14). An attacker relying on any side-channel-based classifier with a lower F-score could achieve better results by simply using an always-one oracle, *i.e.*, in such a case it would not make sense to use the side-channel-based classifier in the first place. In the remainder of the paper, we assume that an attacker wants to find the real 8 keystrokes in 100 possible time frames per second.

This attack model does not have the concept of processes or windows. Indeed, this is an accurate representation, as side-channel attacks on keystroke timings are system-wide attacks on shared code, cache sets, or other shared parts of the microarchitecture. This makes them very powerful but also provides us a means to defeat them, *i.e.*, an attacker cannot distinguish real keyboard input in one process or window from fake keyboard input in another process or window.

A. Keystroke Timing Attack Surface

Keystroke processing involves computations on all levels of the software stack. Hence, targeted solutions like Cloak cannot provide complete protection in this case [15]. The keyboard interrupt is handled by one of the CPU cores, which interrupts the currently executed thread. A significant amount of code is executed in the *operating system kernel* and the keyboard driver until the preprocessed keystroke event is handed over to a user space *shared library* that is part of the user interface. The shared library distributes the keystroke event to all user interface elements listening for the event. Finally, the shared library hands over the keyboard input to the active *user space process* which further processes the input, e.g., store a password character in a buffer. This abundance of code and data that is executed and accessed upon a keystroke provides a multitude of possibilities to measure keystroke timings.

B. New Attack Vectors

Software side channels through `procfs` interfaces can be mitigated by restricting access to them [10], [60]. However, such restrictions do not prevent keystroke timing attacks. We demonstrate two new attacks to infer keystroke timings: the first one exploits interrupt timings to detect keystrokes, and the second one relies on Prime+Probe to attack a kernel module. Table I compares the novel attacks we describe in the following

Algorithm 1: Recording interrupt timing

```
for i ∈ {1, ..., N} do
    tsc[i] ← rdtsc();
    if tsc[i] − tsc[i − 1] > threshold then
        events[i] ← tsc[i];
        diff[i] ← tsc[i] − tsc[i − 1];
```

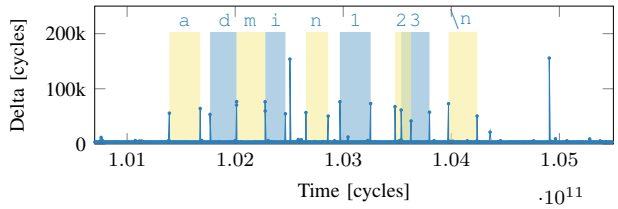

Fig. 3. Measured delta between continuous `rdtsc` calls while entering a password. Keystroke events interrupt the attacker and thus cause higher deltas. Background color illustrates the keystroke ground truth. Periodic interrupts at 1.025 and 1.049 have a different interruption time.

with the state-of-the-art attack vectors (cf. Section II-C) in terms of attack techniques and the exploited attack surface.

Low-Requirement Interrupt Timing Attack. We propose a new *timing-based* attack that only requires unprivileged sandboxed code execution on the targeted platform and an accurate timing source, e.g., the `rdtsc` instruction or a counter thread. The basic idea is to monitor differences in the execution time of acquiring high-precision time stamps, e.g., the `rdtsc` instruction, as outlined in Algorithm 1. While small differences between successive time stamps allow us to infer the CPU utilization, larger differences indicate that the measurement process was interrupted. In particular, I/O events like keyboard interrupts lead to clearly visible peaks in the execution time, due to the interaction of the keyboard ISR with hardware and the subsequent processing of keystrokes. Modern operating systems have core affinities for interrupts, which generally do not change until the system is rebooted, and core affinities for threads. Hence, once a thread runs on the core for the keyboard interrupt, it will continuously be interrupted by every keyboard interrupt, making this attack surprisingly reliable. By starting multiple threads an attacker can first run on all cores and after detecting which thread receives keyboard interrupts, terminate all threads but the one that is running on the right core.

Note that this attack does not benefit at all from attacker process and victim process running on the same core. The keyboard interrupt is scheduled based on its core affinity and not based on the core affinity of any victim thread. Hence, the attack works best if the attacker has a lot of computation time on the interrupt-handling core, but not the victim core.

Figure 3 illustrates these observations in a timing trace recorded while the user was typing a password. The bars indicate actual keystroke events, which almost perfectly match certain measurement points. Based on this plot, we can clearly distinguish keyboard interrupts (around 60 000 cycles) from other interrupts. For example, rescheduling interrupts can be observed with a difference of about 155 000 cycles. In this attack, we achieve a precision of 0.89 and a recall of 1, resulting in an F-score of 0.94, which means a significant advantage over an always-one oracle of +537.4 %.

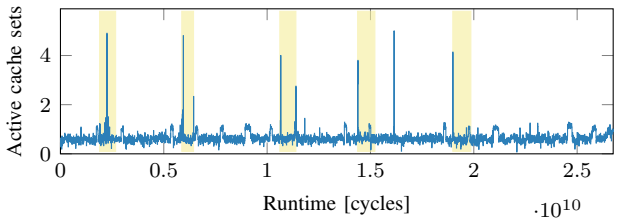

Fig. 4. Multi-Prime+Probe attack on password input. Keystrokes cause higher activity in more cache sets. Background color illustrates the keystroke ground truth.

In our attack, we targeted the laptop keyboard of a Lenovo ThinkPad T460s (*i.e.*, a PS/2 keyboard), and touchscreens of multiple smartphones (cf. Appendix). The attack might not work on USB keyboards, as they are typically configured for polling instead of interrupts. However, the defense mechanism we present in Section IV protects USB keyboards as well.

A preliminary version of our attack was the basis for an implementation without `rdtsc` in JavaScript embedded in websites [29]. The authors used this attack to detect the URL typed by the user with a high accuracy and even distinguish different users typing on the same machine. Showing that the attack even works in this much more constrained environment underlines the practicality of our attack. It is not influenced by foreground, background, or sandboxed operation.

Multi-Prime+Probe Attack on the Kernel. Our second attack relies on Prime+Probe to attack the keyboard interrupt handler within the kernel. More specifically, we target the code in the keyboard interrupt handler that is executed each time a key is pressed. Thereby, keystroke events can be inferred by observing cache activity in the cache set used by the keyboard interrupt handler.

To find the cache sets that are accessed by the keyboard interrupt handler, we first need to find the physical addresses where the code is located. We can use the TSX-based side channel by Jang et al. [24] to locate the code within the kernel. Kernel Address-Space-Layout Randomization was not enabled by default until Ubuntu 16.10. Thus, an attacker can also just use known physical addresses from an attacker-controlled system.

To reduce the influence of system noise, we developed a new form of Prime+Probe attack called Multi-Prime+Probe. Multi-Prime+Probe combines the information from multiple simultaneous Prime+Probe attacks on different addresses. Figure 4 shows the result of such a Multi-Prime+Probe attack on the keyboard interrupt handler. In a post-processing step, we smoothed the Multi-Prime+Probe trace with a 500 µs sliding window. The keystroke events cause higher activity in the targeted cache sets and thus produce clearly recognizable peaks for every key event. Despite doubts that such an attack can be mounted [16], our attack is the first highly accurate keystroke timing attack based on Prime+Probe on the last-level cache. More specifically, we achieve a precision of 0.71 and a recall of 0.92, resulting in an F-score of 0.81, which is significantly better than state-of-the-art Prime+Probe attacks.

C. Requirements for Elimination of Keystroke Timing Attacks

As demonstrated in the previous section, we are able to craft new attacks with fewer requirements than state-of-the-art attacks. Hence, countermeasures against keystroke timing attacks must be designed in a generic way, in all affected layers of the software stack, covering known and unknown attacks.

Attack Model. We assume that an attacker can run an unprivileged program on the target machine, with a recently updated system. As sensor-based attacks [7] are already addressed in [48], and Android O [14], [25] also mitigates various procfs attacks, we consider them out of scope for this paper.

The attacker is able to continuously monitor a side channel to obtain traces for all user input. We assume the (hypothetical) countermeasure against keystroke timing attacks was already installed when the attacker gained unprivileged access to the machine. Consequently, the attacker cannot obtain keystroke timing templates and thus cannot perform a template attack.

We assume that an attacker can generally obtain only a single trace for any user input sequence, but multiple traces for password input. In contrast to side-channel attacks on algorithms, which can be repeated multiple times, user input sequences are generally not (automatically) repeatable, and thus an attacker cannot obtain multiple traces. An exception are phrases that are repeatedly entered in the same way, such as login credentials and especially passwords. A countermeasure must address both cases.

To effectively eliminate keystroke timing attacks, we identify the 3 following requirements a countermeasure must fulfill.

***R1*: Minimize Side Channel Accuracy.** As user input sequences are in general not (automatically) repeatable, keystroke timing attacks require a high precision and high recall to succeed. To be effective, a countermeasure must reduce the F-score enough so that the attacker does not gain any advantage from using the side channel over an always-one oracle. More specifically, the F-score of the side-channel based classifier may not be above the F-score of the always-one oracle (0.15). Ristenpart et al. [46] reported a false-negative rate of $5\,\%$ with 0.3 false positives per second. At an average typing speed for a skilled typist of 8 keystrokes per second [44], the F-score is thus 0.96, which is an advantage over an always-one oracle of $+545.3\,\%$. Gruss et al. [16], [17] reported false-negative rates $\leq 8\,\%$ with no false positives, resulting in an F-score of > 0.96, which is an advantage over an always-one oracle of $+546.9\,\%$. Thus, we assume a countermeasure is effective if it reduces the F-score of side channels significantly, such that using the side channel gives an advantage over an always-one oracle of $\leq 0.0\,\%$.

***R2*: Reduction of Statistical Characteristics in Password Input.** In the case of a password input, we assume that an attacker can combine information from multiple traces, *i.e.*, exploit statistical characteristics. A countermeasure is effective if the attacker requires an impractical number of traces to reach the F-score of state-of-the-art attacks, *i.e.*, higher than 0.95.

Specifically, if the side-channel attack requires more traces than can be practically obtained, we consider the side-channel attack not practical. Studies [8], [9], [11], [47], [54] estimate that most users have 1–5 different passwords and enter 5 passwords per day on average. It is also estimated that 56% of users change their password at least once every 6 months. Thus, even if we assume that we attack a user with a single password that is entered 5 times per day, the expected number of measurement traces that an attacker is able to gather after 6 months is 913. Assuming that attackers might come up with new side-channel attacks, a generous security margin must be applied. We consider a countermeasure effective if it requires more than 1825 traces, *i.e.*, traces for a whole year, to reach an F-score of 0.95.

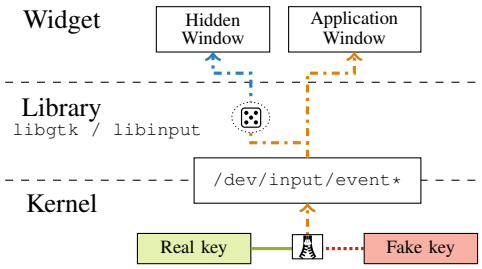

Fig. 5. Multi-layered design of *KeyDrown*.

***R3*: Implementation Security.** *R1* and *R2* define how the countermeasure must be designed to be effective. However, the implementation itself can indirectly violate *R1* or *R2* by leaking side-channel information on computations of the countermeasure itself. Consequently, an attacker may be able to filter the true positive keystrokes. We thus require that the countermeasure may not have distinguishable code paths or data access patterns to guarantee that it is free from leakage.

If the implementation does not leak by itself, an attacker is only left with the low F-scores from *R1* and *R2*. If all requirements are met, classical password recovery attacks like brute force and more sophisticated attacks using Markov n-grams [33], [38], probabilistic context-free grammars (PCFG) [52], [55], or neural networks [36], are more practical than a side-channel attack in the presence of the countermeasure.

In the following section, we describe the design of a countermeasure that fulfills all three requirements.

IV. *KeyDrown* Multi-Layer Design

We designed *KeyDrown* as a multi-layered countermeasure.[2] Each layer builds up on the layer beneath and adds additional protection. Figure 5 shows how the layers are connected to each other. The first layer implements a protection mechanism against interrupt-based attacks and timing-based attacks by artificially injecting interrupts. Any real keyboard interrupt only replaces one fake keyboard interrupt within a multitude of fake interrupts, *i.e.*, it perfectly blends in the stream of random fake keyboard interrupts. The implementation ensures that it makes the keystroke interrupt density uniform over time and thus, independent of the real interrupts. Figuratively speaking, plotting the number of keystroke interrupts over time will yield a line which has no deviations at the points in time where real keystrokes occur.

[2]The code and a demo video are available in a GitHub repository: https://github.com/keydrown/keydrown.

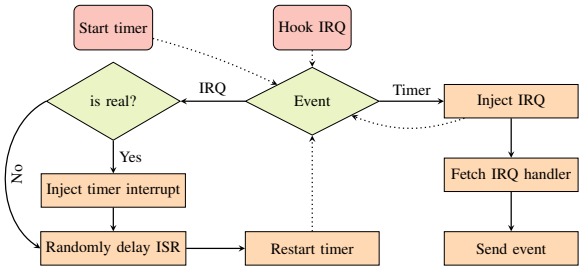

Fig. 6. General flowchart of the kernel module.

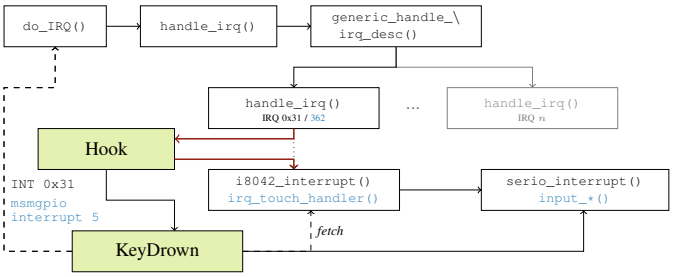

Fig. 7. Linux kernel module design for x86 and the Snapdragon SoC. Snapdragon specific functions are marked in blue.

KeyDrown exploits that keystroke timing side channels do not provide the information which process or window is receiving the keystroke. These side channels are system-wide attacks on shared code, shared cache sets, or other shared parts of the microarchitecture. While this makes them very powerful (cross-core, cross-user attacks), it is also the basis for our defense mechanism. An attacker cannot distinguish real keyboard input in one process or window from fake keyboard input in another process or window. *KeyDrown* exploits this technicality and sends the fake keyboard input through the entire software stack into a special process and window. All keystrokes, *i.e.*, real keystrokes and fake keystrokes, are passed to the library in a way which is indistinguishable for an attacker. The only difference is the key code value as well as the target process and window, which both cannot be obtained in keystroke timing side channels.

The second layer protects the library handling the user input against Flush+Reload attacks, including cache template attacks, and Prime+Probe attacks. For every keystroke event received from the kernel, a random keystroke is sent to a hidden window. The library cannot distinguish between real and fake keystrokes and thus both have the same execution path. Note that this also triggers screen redraw events, hence, the screen-redraw interrupt side channel is also covered by *KeyDrown*.

In the third layer, the actual password entry field is protected against Prime+Probe attacks by accessing the underlying buffer whenever a real or a fake keystroke is received.

Combining the three layers, the system-wide set of cache lines that are touched by the code paths through the entire software stack for real and fake keystrokes, are identical. As there is no difference, this voids any advantage an attacker could have gained from a cache side channel.

A. First Layer

Basic Concept. Figure 6 shows the program flow for the kernel part of *KeyDrown* for both x86 and ARM. We use a non-periodic one-shot timer interrupt with a random delay to inject a fake keystroke.[3] This leads to a uniform random distribution of keystrokes over time.

The kernel module handles two types of events: Hardware interrupts from the input device, and the timer interrupts. If the kernel module receives one of our timer interrupts, it injects a keyboard interrupt. If it receives a keyboard interrupt, it injects a non-periodic one-shot timer interrupt. Thus, for real and fake keystrokes both interrupts occur. To minimize the effect of the real keyboard interrupt on the interrupt density, the upcoming non-periodic one-shot timer interrupt is canceled. Note that the time between the fake keyboard interrupt and the user pressing a key was also a random delay. *KeyDrown* acts as if this random delay was planned for the fake keyboard interrupt all along. That is, the real keyboard interrupt takes the place of our fake keyboard interrupt. Hence, the real keyboard interrupt has no additional influence on the keystroke interrupt density function. This guarantees that overall, the keystroke interrupt density remains uniform and real keystrokes cannot be distinguished from fake keystrokes.

For the fake keystrokes, the kernel uses a typically unused key value. The kernel does not have varying code paths and data accesses based on the key value, hence, the same code is executed for both real and fake keystrokes. In both cases, the keystroke handler is delayed by a small random delay to hide timing differences from interrupt runtimes. Finally, all keystrokes are passed to the library through the same data structures (cf. Figure 5). Consequently, the attacker cannot use a Prime+Probe or Multi-Prime+Probe attack on the kernel to distinguish real and fake keystrokes.

Implementation Details. The first layer of *KeyDrown* is implemented as a Linux kernel module that aims to prevent interrupt-based attacks on keystrokes. We do not require a custom kernel or any patches to the Linux kernel itself, but only the Linux kernel header files for the running kernel. All functionality is implemented in one generic kernel module that can be loaded into any Linux kernel from version 3.4 to 4.10, the newest release at the time of writing. The interrupt hardware and handling mechanism is compatible with all personal computers; thus, there is no further limitation on PC hardware or Linux distributions.

Figure 7 shows the implementation details of the *KeyDrown* kernel module. The non-periodic one-shot timer interrupts are implemented using the Linux platform-independent high-resolution timer API [32]. On Linux, a driver can register an interrupt handler for a specific interrupt which is called whenever the CPU receives the interrupt. The interrupt service routine `do_IRQ` calls the general `handle_irq` function which subsequently calls `generic_handle_irq_desc` to execute the correct handler for every interrupt. To receive all hardware interrupts, we change the input device's interrupt handler to a function within our kernel module. Afterwards,

[3]Timer interrupts are often known as periodic interrupts triggering regular operations, e.g., scheduling. However, on modern systems there are significantly more features to timer interrupts, such as non-periodic one-shot timers [22]. One-shot timers are architectural features that can be used through legitimate kernel interfaces and have no side effects on any system timers.

we forward the interrupt to the actual input device driver (*i.e.*, `i8042_interrupt` on x86, and `irq_touch_handler` on the Nexus 5 (ARM)). Every time the kernel receives one of the non-periodic timer interrupts or a real hardware interrupt, we restart the non-periodic timer with a new random delay to maintain the uniform random distribution over time.

The kernel module triggers a hardware interrupt for every non-periodic timer interrupt. On x86, we can simply execute the `int` assembly instruction with the corresponding interrupt number. This spurious keyboard interrupt travels up until the point where the keyboard driver tries to read the scancode from the hardware. As the driver does not execute the entire `i8042_interrupt` function for spurious interrupts, we access the remaining function to fetch it into the cache as if it was executed. In contrast, for real keys, we access the code that injects the keys to fetch it into the cache as if it was executed. From an attacker's point of view, there is no difference in cache activity between a data fetch and a code fetch, *i.e.*, a Prime+Probe attack cannot determine the difference.

We inject a scancode of a typically unused key, such as F16 or a Windows multimedia key using the standard `serio_interrupt` interface. Thus, from this point on the only difference between real and fake keystrokes is the scancode. Finally, all scancodes are sent to the upper software layers and run through the same execution path.

On the ARM platform, hardware interrupts and device drivers are hardware dependent. We decided to implement our proof-of-concept on the widespread Qualcomm Snapdragon Mobile Station Modem (MSM) SoC [45].

ARM processors generally do not provide an assembly instruction to generate arbitrary interrupts from supervisor mode. Instead, we have to communicate with the interrupt controller directly. The Snapdragon MSM SoC implements its own intermediate I/O interrupt controller. All interrupt generating hardware elements are connected to this interrupt controller and not directly to the GIC. Therefore, if we want to inject an interrupt, we write the interrupt state of the touchscreen interrupt via memory mapped I/O registers to the MSM I/O interrupt controller. The remaining execution path is analogous to the x86 module. When the driver aborts due to a spurious interrupt, we fetch the `irq_touch_handler` to produce the same cache footprint as if it was executed. We inject an out-of-bounds touch event using the `input_event`, `input_report_abs`, and `input_sync` functions, which is then handed to the upper layers.

B. Second Layer

Basic Concept. The second layer countermeasure ensures that the control flow within the key-handling library is exactly the same for both real and injected keystrokes. The fundamental idea of the second layer is that real and injected keystrokes should have the same code paths and data accesses in the library. We rely on the events injected in the first layer to propagate them further through the key-handling library. The injected keys sent by our first layer are valid, but typically unused keys, thus they travel all the way up to the user space to the receiving user space application. However, these unused keys might not have the exact same path within the library.

Gruss et al. showed that an attacker can build cache template attacks based on Flush+Reload [17] to detect keystrokes and even distinguish groups of keys. This cache leakage can also be measured with Multi-Prime+Probe. Both attacks exploit the cache activity of certain functions that are only called if a keystroke is handled, *i.e.*, varying execution paths and access patterns [17]. We mitigate these attacks by duplicating every key event (cf. Figure 5) running through multiple execution paths and access sequences simultaneously. The key value of the duplicated key event is replaced by a random key value, and the key event is sent to a hidden window. Hence, the two key events, the real and the duplicated one, are processed simultaneously by the remainder of the library and the two applications. This introduces a significant amount of noise on cache template attacks on the library layer.

The real key event at this point may still be a fake keystroke from the kernel. However, we duplicate the key event in order to trigger key value processing and key drawing in the library and the hidden window for both fake and real keystrokes. Consequently, we cannot distinguish real and fake keystrokes on the library layer using a side channel anymore.

Implementation Details. One of the most popular user interface libraries for Linux is *GTK+* [57]. The *GTK+* library handles the user input for many desktop environments and is included in most Linux distributions [51]. As we cannot hide cache activity, we generate artificial cache activity for the same cache lines that are active when handling real user inputs.

The kernel provides all events, such as keyboard inputs, through the `/dev/input/event*` pseudo-files to the user space. The X Window System uses these files to provide all events to the *GTK+* event queue.

On x86, the second layer is a standalone *GTK+* application. On system startup, we create a hidden window containing a text field. The application uses `poll` to listen to the `/dev/input/event*` interface to get notified whenever a keyboard event occurs. This allows *KeyDrown* to have a very low performance overhead, as the application is not using CPU time as long as it is waiting inside the `poll` function. Whenever we receive a keystroke event from the kernel, we create an additional *GTK+* keystroke event with a random key that is associated with the text field of the hidden window. For every keystroke — regardless of whether it is a printable character or not — that comes from the kernel, the same path is active within the library. Thus, an attacker cannot distinguish an injected keystroke from a real keystroke anymore.

The second layer has no knowledge of an event's source. Thus, it cannot violate *R3*, as the information whether a keystroke is real or injected is not present in the second layer.

On Android, the handling of input events is considerably simpler. The injected events travel directly to the foreground application without going to any non-Android library. Thus, all events have exactly the same execution path, and it is only necessary to drop our fake event immediately before the registered touch event handler is called. To not leak any information through the non-executed touch handler, we access the cache lines in the same way as if the touch handler was executed.

C. Third Layer

Basic Concept. While the first layer protects against interrupt-based attacks and the second layer prevents attacks on the library handling the user inputs, the buffer that stores the actual secret is not protected and can still be monitored using a Prime+Probe attack. The fake keystrokes sent by the kernel are unused key codes, which do not have any effect on the user interface element or the corresponding buffer. We mitigate cache attacks on this layer by generating cache activity on the cache lines that are used when the buffer is processed for any key code received from the kernel. More specifically, we access the buffer every time the library receives a keystroke event from the kernel. This ensures that the buffer is cached for both real and fake keystrokes.

An attacker who mounts a Flush+Reload attack against the library, or a Prime+Probe attack directly on the buffer, sees cache activity for both real and injected events. This is also the case for cache template attacks, as the injected events induce a significant amount of noise in both the profiling and the exploitation phase. Therefore, the third layer protects against attacks that are mounted against the Android keyboard as shown by Lipp et al. [30], or Multi-Prime+Probe attacks directly on the input field buffer (cf. Section III-B).

Implementation Details. In *GTK+*, the `GtkEntry` widget implements the `GtkEditable` interface, which describes a text-editing widget, used as a single-line text and password entry field. By setting its *visibility* flag, entered characters are shown as a symbol and, thus, hidden from the viewer.

Implementing the countermeasure directly in the *GTK+* library would require rebuilding the library and all of its dependencies. As this is highly impractical, we chose a different approach: `LD_PRELOAD` allows listing shared objects that are loaded before other shared objects on the execution of the program [28]. By using this environment variable, we can overwrite the `gtk_entry_new` function that is called when a new object of `GtkEntry` should be created. In our own implementation, we register a key press event handler for the new entry field. This event handler is called on both real and injected keys and accesses the underlying buffer.

On Android, the basic concept is the same. It is, however, implemented as part of the keyboard and not the library. The keyboard relies on the `inotifyd` command to detect touch events provided by the kernel. If a password entry field is focused, the keyboard accesses the password entry buffer on every touch event by calling the key handling function with a dummy key. This ensures that both the buffer as well as the keyboard's key handling functions are active for every event.

V. Evaluation

We evaluate *KeyDrown* with respect to the requirements *R1*, *R2*, *R3* as well as discuss the performance of our implementation. We evaluate the x86 version of *KeyDrown* on a Lenovo ThinkPad T460s (Intel Core i5-6200U) and the ARM version on both an LG Nexus 5 (ARMv7) and a OnePlus 3T (ARMv8). A large comparison table can be found in the appendix. As the results are very similar for all architectures, we provide the results for the LG Nexus 5 (ARMv7) and the OnePlus 3T (ARMv8) in the appendix. We evaluate four different side channels with and without *KeyDrown*: `procfs`, `rdtsc`, Flush+Reload (including cache template attacks), and Prime+Probe on the last-level cache. We discuss Prime+Probe attacks on the L1 cache and DRAMA side-channel attacks. Table II gives an overview of all known and new attacks and whether *KeyDrown* prevents them.

TABLE II. Overview which attacks work (●), partly work (◐) and do not work (○) with enabled (✓) and disabled (✗) *KeyDrown*.

	Android < 8		Android ≥ 8		Linux	
KeyDrown	✗	✓	✗	✓	✗	✓
Interface-based [10], [23], [60]	●	◐	○	○	●	○
Interrupt-based (`rdtsc`, [53])	●	○	●	○	●	○
Prime+Probe on L1 [46]	●	○	●	○	●	○
Prime+Probe on LLC	●	○	●	○	●	○
Multi-Prime+Probe	●	○	●	○	●	○
Flush+Reload [17]	●	○	●	○	●	○
DRAMA [43]	●	◐	●	◐	●	◐

To evaluate *KeyDrown*, we chose a uniform key-injection interval $[0\,\mathrm{ms}, 20\,\mathrm{ms}]$. Note that this is not a constant interrupt rate but quite the opposite. Any real keystroke replaces the currently scheduled key injection. Real keystrokes are much rarer and when splitting time into $20\,\mathrm{ms}$ intervals, the distribution of real keystrokes in these $20\,\mathrm{ms}$ intervals is uniform, identical to the uniform distribution of our key-injection delay. Hence, based on the time a keystroke arrives there is no side channel leaking whether it was a fake one, or a real one. This leads to a uniform interrupt density function with 100 events per second, independent of the real keystrokes.

As described in Section III, we compare our results to an always-one oracle and a random-guessing oracle. A random-guessing oracle, which chooses randomly — without any information — for every $10\,\mathrm{ms}$ interval whether there was a keystroke based on an apriori probability, would achieve an F-score of 0.14. The always-one oracle performs slightly better, as it has a higher true positive rate of $100\,\%$, but it also has a false positive rate of $100\,\%$, *i.e.*, the oracle neither uses nor provides any information. The F-score of the always-one oracle is 0.15 and thus, higher than the F-score of a random-guessing oracle. If a side channel yields an F-score of this value or below, the attacker gains no advantage over the always-one oracle from this side channel.

For all evaluated attacks, we provide the precision of the attack with and without *KeyDrown*, based on the best threshold distinguisher we can find. *KeyDrown* does not influence the recall, as it does not reduce the number of true positives and it also does not increase the number of real keystrokes. However, we provide the recall for all attacks with a recall below 1. The harmonic mean of precision and recall — the F-score — gives an indication how well the countermeasure works. We provide the advantage over the always-one oracle as a direct indicator on whether it makes sense to use the side channel or not.

A. Requirement R1

We evaluate *KeyDrown* with respect to *R1*, the elimination of single-trace attacks. *R1* defines that a side channel may not provide any advantage over an always-one oracle, *i.e.*, the advantage measured in the F-score must be $\leq 0.0\,\%$. We show that *KeyDrown* fulfills this requirement by mounting state-of-the-art attacks with and without *KeyDrown*. Table III summarizes the F-scores for all attacks with and without

TABLE III. F-SCORE WITHOUT AND WITH *KeyDrown* AND ADVANTAGE OVER ALWAYS-ONE ORACLE FOR STATE-OF-THE-ART ATTACKS. *KeyDrown* ELIMINATES ANY SIDE-CHANNEL ADVANTAGE.

Side Channel	no *KeyDrown*	(Δ always-one)	*KeyDrown*	(Δ always-one)
`procfs`	1.00	(+575.0 %)	0.15	(+0.0 %)
`rdtsc`	0.94	(+537.4 %)	0.14	(−3.8 %)
Flush+Reload	0.99	(+569.3 %)	0.09	(−40.2 %)
LLC Prime+Probe	0.81	(+440.0 %)	0.11	(−27.7 %)

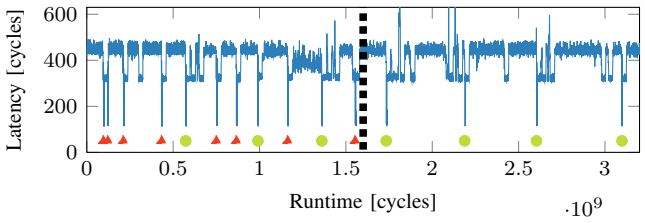

Fig. 8. Flush+Reload attack on address `0x381c0` of `libgdk-3.so`. Injected keystrokes (▲) and real events (●) are not distinguishable when *KeyDrown* is active (before dotted line).

KeyDrown. In all cases, *KeyDrown* eliminates any advantage that can be gained from the side channel, when considering single-trace attacks only. In some cases, the numerous false positives and false negatives lead to an even worse F-score.

Flush+Reload. Flush+Reload allows an attacker to monitor accesses to memory addresses of a shared library with a very high accuracy. Figure 8 shows the result of such an attack against the `gdk_keymap_get_modifier_mask` function at address `0x381c0` of *libgdk-3.so* (v3.20.4 on Ubuntu Linux), the shared library isolating *GTK+* from the windowing system. This function is executed on every keystroke to retrieve the hardware modifier mask of the windowing system. The attacker measures cache hits on the monitored address whenever a key is pressed and, thus, can spy on the keystroke timings very accurately. While *KeyDrown* is active, the attacker measures additional cache hits on every injected keystroke and cannot distinguish between real and fake keystrokes. When *KeyDrown* is not active, the attack is successful.

For other addresses found using cache template attacks, we made the same observation. Without *KeyDrown*, both profiling and exploiting vulnerable addresses is possible. With *KeyDrown*, we still find all addresses that are loaded into the cache upon keystrokes, however, as we cannot distinguish between real and fake keystrokes we cannot exploit this anymore. Without *KeyDrown*, the precision is 1.00 and the F-score is 0.99, which is a +569.3 % advantage over an always-one oracle. If *KeyDrown* is active, the precision is lowered to 0.05 and, thus, the resulting F-score is 0.09, which is a (negative) advantage of −40.2 % over the always-one oracle.

Prime+Probe. If an attacker cannot use Flush+Reload, a fallback to Prime+Probe is possible. The disadvantage of a Prime+Probe attack on the last-level cache is the amount of noise that increases the false-positive rate. Prior to this work, there was no successful keystroke attack using Prime+Probe on the last-level cache. We perform the Multi-Prime+Probe attack presented in Section III-B to attack keystroke timings.

Figure 9 shows the results of inferring keystrokes by detecting the keyboard interrupt handler's cache activity using Multi-Prime+Probe. We monitored 5 cache sets in parallel for a higher noise robustness. Without *KeyDrown*, the precision is

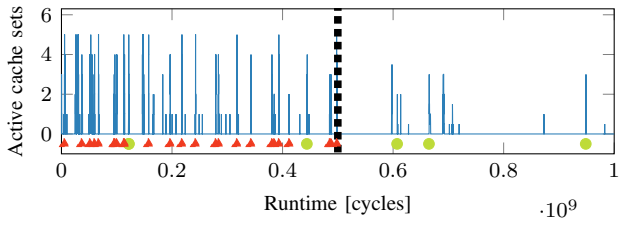

Fig. 9. Multi-Prime+Probe attack on the 5 cache sets from `0x2514250` to `0x2514390` of `i8042_interrupt`. Injected keystrokes (▲) and real events (●) are not distinguishable with *KeyDrown* (before dotted line).

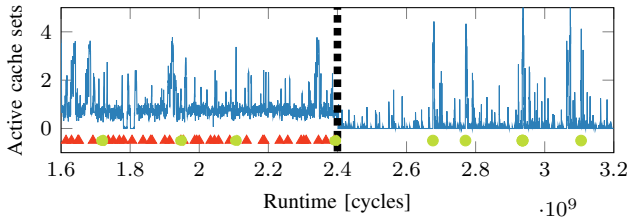

Fig. 10. Multi-Prime+Probe attack on the 5 cache sets corresponding to a password field's buffer within a demo application. Injected keystrokes (▲) and real events (●) are not distinguishable with *KeyDrown* (before dotted line).

already at a quite low value of 0.71 with a recall of only 0.92, yielding an F-score of 0.81, which is an advantage over an always-one oracle of +440.0 %. Memory accesses to one of the cache sets by any other application cannot be distinguished from a cache set access by the keyboard interrupt handler, causing a high number of false positives. If we enable *KeyDrown*, the precision drops to 0.06, as the attacker additionally measures the noise generated by the injected keystrokes. The F-score is then 0.11, which is a (negative) advantage over an always-one oracle of -27.7 %.

Figure 10 shows the results of mounting a Multi-Prime+Probe attack on the buffer of a password field within a *GTK+* application. Although there is more noise visible in the traces, we achieve the same precision and F-score as for the attack on the kernel module when *KeyDrown* is disabled. If we enable *KeyDrown*, the precision drops to 0.05, which is a bit lower than the precision on the kernel, resulting in an F-score of 0.10, which is again no advantage over an always-one oracle.

Interrupts. *KeyDrown* also protects against interrupt-based attacks, including our new timing-based attack. For the attacks based on the `procfs` interface [10], [23], we measure an average reading interval of 980 cycles. With our new attack based on `rdtsc`, we can measure every 95 cycles on average, resulting in a probing frequency one order of magnitude higher.

Figure 11 and Figure 12 illustrate the effect of our countermeasure on the `procfs`-based interrupt attack and the `rdtsc`-based attack, respectively. Without *KeyDrown*, we achieve a precision of 1.00 for the `procfs`-based attack and a precision of 0.89 for the `rdtsc`-based attack, resulting in an F-score of 1.00 and 0.94 respectively. Enabling *KeyDrown* reduces the precision to 0.08 and 0.07 respectively. Thus, the resulting F-score is 0.15, which is exactly the same as the always-one oracle, for the `procfs`-based attack, and 0.14 for the `rdtsc`-based attack, which is a (negative) advantage over an always-one oracle of -3.8 %.

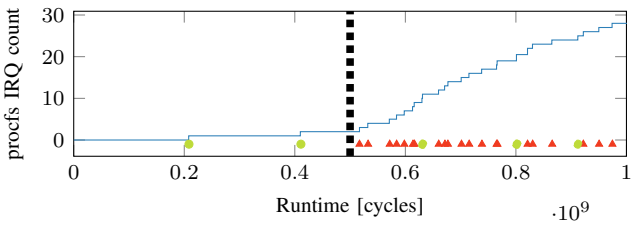

Fig. 11. `procfs`-based attack. Injected keystrokes (▲) and real events (●) are not distinguishable when *KeyDrown* is active (after dotted line).

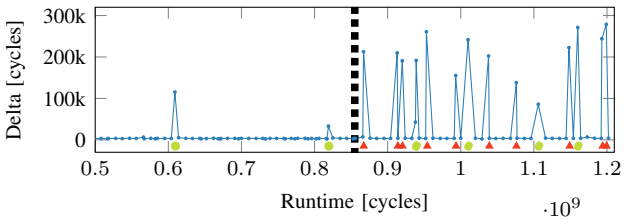

Fig. 12. `rdtsc`-based attack. Injected keystrokes (▲) and real events (●) are not distinguishable when *KeyDrown* is active (after dotted line).

B. *Requirement* R2

KeyDrown reduced the F-score of all state-of-the-art attacks such that using the side channel gives an advantage over an always-one oracle of $\leq 0.0\,\%$. An attacker might still be able to combine multiple traces from the same user and build a binary classifier, if the user predictably and repeatedly types the same character sequence. Such a classifier may achieve a higher precision and a higher F-score, as long as there is actually meaningful information in the corresponding traces. However, there is a practical limit on the number of traces an attacker can gather from the user, which *R2* estimates to be 1825 traces.

In our attack scenario, we model a powerful attacker who can take advantage of the following properties:

1) **Noise-free side channel:** The used side channel is noise-free, *i.e.*, only real and fake keystrokes are recorded, no other system noise.
2) **Perfect (re-)alignment:** The attacker can detect when a password input starts with a variance as low as the variance of a single inter-keystroke interval. Additionally, the attacker has an alignment-oracle providing perfect re-alignment for the traces after each guessed keystroke. This leads to the same variance for every key instead of an accumulated variance.
3) **Known length:** The attacker knows the exact length of the password and expects exactly as many keystrokes.

This attacker is far stronger than any practical attacker.

We generate simulated traces that fulfill the properties above and calculate the average of the perfectly (re-)aligned traces. As our attacker knows the length n of the password, he finds the n most likely positions where a Gaussian distribution with the known inter-keystroke interval variance matches. If the expected value μ of each Gaussian curve is within the variance of the real keystroke, we assume that the number of traces was sufficient to extract the positions of the real keystrokes.

We set the simulated typing variance to $\pm 40\,\mathrm{ms}$ which is a bit less than the value reported by Lee et al. [27] for trained text sequences. In total, we generated $300\,000$ simulated traces, each containing 8 keystrokes within $2\,\mathrm{s}$. From this set of simulated traces, we evaluated how many randomly chosen traces we have to combine to extract the correct positions of the keystrokes. We found that an attacker requires an average of 2458 traces to extract the correct positions. This is significantly more than the 1825 traces deemed to be secure in *R2*.

C. *Requirement* R3

As *KeyDrown* fulfills *R1* and *R2*, we can be assured that the underlying technique is a working countermeasure. However, as the implementation of a countermeasure itself can leak information, we need to ensure that *KeyDrown* does not create a new software-based side channel in order to satisfy *R3*.

First Layer. The first layer runs in the kernel and can thus only be attacked using Prime+Probe. Figure 7 shows that, in general, we have the same execution flow and data accesses. For the few deviations, we prevent any potential cache leakage from non-executed code paths by performing the same memory accesses as if they were executed. As an attacker cannot distinguish if a cache activity is caused by an execution or a memory read, the module's cache activity does not leak additional information to an attacker. We investigated the cache activity on the cache sets used by the *KeyDrown* kernel module in a Prime+Probe attack and found no leakage from our module.

Second Layer. To make use of the same noise as in the first layer, the second layer listens to the `/dev/input/event0` pseudo-file containing all keyboard events. This file is not world-readable but only readable by members of the `input` group. Thus, this layer runs as a separate *keydrown* user with default limited privileges and additional access to this file.

As the second layer is a user space binary, an attacker could theoretically mount a Flush+Reload attack against the second layer. However, attacking the second layer does not result in any additional information. The second layer does not know whether an event is generated from a real or an injected keystroke. For every event, a random printable character is sent to the hidden window. Thus, the execution path for printable characters is always active, and the attacker cannot learn any additional information from attacking the second layer. The same is also true for Prime+Probe, even a successful attack does not provide additional information. We investigated the cache activity of the *KeyDrown* shared library parts and the *KeyDrown* user space binary using a template attack and did not find any leakage.

Third Layer. The third layer builds upon the second layer, and thus the same argumentation as for the second layer holds. An attacker cannot distinguish real and injected keystrokes in the second layer as all events are merged within the kernel. As the third layer relies on the same source as the second layer, there is also no leakage from the third layer. Thus, any attack on the third layer does not give an attacker any advantage over any other attack. We investigated the cache activity of the control flow and data accesses up to the point where the input is stored in the buffer in a Prime+Probe attack and found no leakage.

D. Performance

On the x86 architecture, we evaluate the performance impacts of running our *KeyDrown* implementation on standard Ubuntu 16.10. We use *lmbench* [35], a set of micro benchmarks for performance analysis of UNIX systems, and *PARSEC 3.0* [5], a benchmark suite intended to simulate a realistic workload on multicore systems.

The lmbench results for the latency benchmarks show a performance overhead of $6.9\,\%$. However, as the execution time of the lmbench benchmarks is in the range of microseconds to nanoseconds, the overhead does not allow for definite conclusions about the overall system performance. Still, we can see that the injected interrupts have only a small impact on the kernel performance.

To measure the overall performance, we run the PARSEC 3.0 benchmark with different numbers of cores. The average performance overhead over all measurements for any number of cores is $2.5\,\%$. For workloads that do not use all cores, the performance impact is only $2.0\,\%$ for one core and $2.5\,\%$ for two cores. Only if the CPU is under heavy load, we observe a higher performance impact of $3.1\,\%$ when running the benchmarks on all cores.

On ARM, we evaluate the battery consumption of *KeyDrown*. We measure the power consumption in three different scenarios, always over the timespan of $5\,\mathrm{min}$. First, if the screen is off, our fake interrupts are completely disabled, and thus, *KeyDrown* does not increase the power consumption if the mobile phone is not used. Second, if the screen is turned on, but the keyboard is not shown, *KeyDrown* increases the power consumption slightly by $3.9\,\%$. Third, if the keyboard is shown, the power consumption with *KeyDrown* increases by $15.6\,\%$. However, as most of the time, the keyboard is not shown, *KeyDrown* does not have great impacts on the overall power consumption. In total, *KeyDrown* reduces the battery life time of an average user by $4.6\,\%$.[4]

Note that all the performance measurements were done using the proof-of-concept. We expect that the proof-of-concept can be considerably improved in terms of performance overhead and battery usage by not injecting the fake interrupts all the time but only while the user is actually entering text.

E. Other Attacks

While we already demonstrated that the most powerful side-channel attacks are mitigated, we discuss three other attacks subsequently. The Prime+Probe side channel results from the victim program evicting a cache line of the attacker. As the last-level cache is inclusive, any eviction from the last-level cache also evicts this line from the L1 cache. However, if a cache line is evicted from the L1 cache it may still be in the last-level cache. In this case, the attacker would miss the eviction and thus the targeted event. In our evaluation, we find that the recall is very close to 1 in all cases. This means that we do not miss any events. Hence, there is no additional information that an attacker could gain from a Prime+Probe attack on the L1 cache. Consequently, evaluating Prime+Probe on the last-level cache is sufficient to conclude that Prime+Probe on the L1 cache does not leak additional information.

The DRAMA side-channel attack presented by Pessl et al. [43] results from a massive number of secret-dependent memory accesses that lead to heavy cache thrashing, *i.e.*, the victim program accesses lots of memory locations that are mapped to the same cache lines. It is therefore unclear whether or not *KeyDrown* protects against DRAMA. In particular, it does not protect against the specific attack against keystrokes in the Firefox address bar (cf. Section VI). However, we observe that *KeyDrown* adds significant amounts of noise to the attack.

To our surprise, we found that *KeyDrown* also mitigates the keystroke timing attack based on the event queue of the Chrome browser by Vila et al. [53] (USENIX Sec'17). They state that the leakage is due to the time it takes Chrome to enqueue and dispatch every keystroke event. However, we investigated their attack and were able to reproduce it on MacOS systems reliably, but not on other operating systems, indicating that this effect is not purely Chrome-specific, but also has other influences. We believe that their attack exploits multiple effects in combination: the Chrome event queue and the interruption by the hardware interrupts as in our `rdtsc`-based attack, which is additionally amplified by the significantly higher I/O latency caused by the atypical MacOS design for interrupt handling [1].[5]

A preliminary version of our `rdtsc`-based interrupt timing attack was the basis for the same attack in JavaScript [29]. They were able to identify the user typed URL, and distinguish different users based on this attack. As they report, *KeyDrown* successfully mitigates their attack in JavaScript as well.

KeyDrown has a significant effect on the attack by Jana et al. [23], exploiting CPU utilization spikes. The fake keystrokes introduce similar small CPU utilization spikes making their attack impractical. Similarly, *KeyDrown* triggers screen redraws through the hidden window (cf. Section IV). Hence, *KeyDrown* also makes the screen-redraw-based attack by Diao et al. [10] impractical.

VI. Limitations and Future Work

KeyDrown mitigates software-based side-channel attacks on keystrokes and keystroke timings in general. This includes even the application layer without changing an existing application if either:

- the input is processed only after the user finished entering the text (e.g., pressing a button on a login form), and there is no immediate action when a key is pressed (e.g., as with password fields or simple text input fields),
- or the application is designed to remove side-channel information.

[4]For an average user, with a screen-on-time of 145 minutes, 2617 touch actions [56], and 1 charge per day (21.7 hours standby time) [26], an average typing speed of 20 words per minute [4] and hence, 100 characters per minute [39], we can assume a keyboard-shown time of 26 minutes per battery charge. For modern devices, screen-on consumes approximately 33 times more battery than standby [13].

[5]Interrupt handlers on MacOS only enqueue the task to handle an interrupt in a queue, taking almost zero time. This queue is processed by an interrupt service thread, doing the actual interrupt handling. This additional step increases the total computation time compared to traditional interrupt handling. As the attack is not influenced by which thread does the actual interrupt handling, this increased interruption time amplifies the side channel.

Otherwise, the application layer might still leak timing information when performing intense computations for every single keystroke, e.g., autocomplete or live search features [43].

Song et al. [50] demonstrated keystroke timing attacks performed by a malicious observer on the same network. Zhang et al. [60] speculated that this attack vector could also be exploited through `/proc/net`, which might still be available in Android O. However, this is not a local software-based attack but a side channel for a remote attacker. Hence, dedicated countermeasures beyond *KeyDrown* should be implemented to prevent this attack.

Some software-based side channel attacks may be unaffected by *KeyDrown*, e.g., the sensor-based attacks exploiting the accelerometer [7], but these attacks can be thwarted by introducing noise [48].

KeyDrown protects against software-based attacks on keystrokes as well as touch events. However, swipe movements are not protected as their interrupt rate is too high. While this is not a problem in the case of a password input — if a password can be swiped and thus pasted from a dictionary, there is little to protect — it is future work to investigate how to extend *KeyDrown* to protect swipe movements.

Furthermore, our novel side channels emphasize the necessity to deploy *KeyDrown* widely. Multi-Prime+Probe attacks provide a significantly higher accuracy than previous Prime+Probe attacks on dynamic memory and kernel memory. It is likely that Multi-Prime+Probe works similarly in cloud systems and thus allows highly accurate attacks like keystroke timing attacks across virtual machine boundaries.

Our current proof-of-concept is not optimized for usability. For most of the system, the real and fake keystrokes are indistinguishable, the keystrokes are just led to different windows. Known limitations are that fake keystrokes interrupt key repetition and may interfere with input methods in modern computer games. However, these limitations can be overcome by adapting how key repetition is implemented.

VII. Conclusion

Keystrokes are processed on many different layers of the software stack and are thus not entirely covered by previously proposed defense mechanisms. In this article, we presented *KeyDrown*, a novel defense mechanism that mitigates keystroke timing attacks. *KeyDrown* injects a large number of fake keystrokes on the kernel level and propagates them — through all layers of the software stack — up to the user space application. A careful design and implementation of this countermeasure ensures that all software routines involved in the processing of a keystroke are loaded, irrespective of whether a real or a fake keystroke is processed. Thereby, *KeyDrown* mitigates interrupt-based attacks, Prime+Probe attacks, and Flush+Reload attacks on the entire software stack. With *KeyDrown*, an attacker cannot distinguish fake from real keystrokes in practice anymore. Our evaluation shows that *KeyDrown* eliminates any advantage an attacker can gain from side channels, *i.e.*, $\leq 0.0\,\%$ advantage over an always-one oracle, thus, it successfully mitigates keystroke timing attacks.

Acknowledgment

We would like to thank our anonymous reviewers for their valuable feedback and Johannes Winter for insights on ARM interrupt handling. This work has been supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644052 (HECTOR). This work was partially supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments".

References

[1] "Performance considerations," Apple Inc., 2013. [Online]. Available: https://developer.apple.com/library/content/documentation/Darwin/Conceptual/KernelProgramming/performance/performance.html

[2] ARM, "Application Note 176 – How a GIC works," 2007. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0176c/ar01s03s02.html

[3] ——, "ARM Generic Interrupt Controller Architecture version 2.0," 2013.

[4] P. Bao, J. Pierce, S. Whittaker, and S. Zhai, "Smart phone use by non-mobile business users," in *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, 2011.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[6] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly Media, Inc., 2005.

[7] L. Cai and H. Chen, "TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion," in *USENIX Workshop on Hot Topics in Security – HotSec*, 2011.

[8] CSID, "Consumer Survey: Password Habits," 2012. [Online]. Available: http://www.csid.com/wp-content/uploads/2012/09/CS_PasswordSurvey_FullReport_FINAL.pdf

[9] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The tangled web of password reuse." in *NDSS'14*, 2014.

[10] W. Diao, X. Liu, Z. Li, and K. Zhang, "No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis," in *S&P'16*, 2016.

[11] S. Gaw and E. W. Felten, "Password management strategies for online accounts," in *SOUPS'06*, 2006.

[12] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware," *Journal of Cryptographic Engineering*, pp. 1–27, 2016.

[13] N. Gondhia, "Samsung galaxy s7 battery life review," 2016. [Online]. Available: http://www.androidauthority.com/samsung-galaxy-s7-battery-life-review-683968/

[14] Google, "Android o prevents access to /proc/stat," Jun. 2017. [Online]. Available: https://issuetracker.google.com/issues/37140047

[15] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *USENIX Security Symposium*, 2017.

[16] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA'16*, 2016.

[17] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security Symposium*, 2015.

[18] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice," in *S&P'11*, 2011.

[19] S. Idrus, E. Cherrier, C. Rosenberger, and P. Bours, "Soft Biometrics for Keystroke Dynamics: Profiling Individuals While Typing Passwords," *Computers & Security*, vol. 45, pp. 147–155, 2014.

[20] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache Attacks Enable Bulk Key Recovery on the Cloud," in *CHES'16*, 2016.

[21] Intel, "82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC)," 1996.

[22] ——, "Intel® 64 and IA-32 Architectures Software Developer′s Manual, Volume 3 (3A, 3B & 3C): System Programming Guide," 2014.

[23] S. Jana and V. Shmatikov, "Memento: Learning Secrets from Process Footprints," in *S&P'12*, 2012.

[24] Y. Jang, S. Lee, and T. Kim, "Breaking Kernel Address Space Layout Randomization with Intel TSX," in *CCS'16*, 2016.

[25] N. Kralevich, "Honey, i shrunk the attack surface," in *Black Hat 2017 Briefings*, 2017.

[26] D. G. B. Lab, "Global app power consumption report 2016, h1," 2016. [Online]. Available: https://medium.com/@DU_Global_Battery_Lab/e7f9b845bed

[27] P.-M. Lee, W.-H. Tsui, and T.-C. Hsiao, "The Influence of Emotion on Keyboard Typing: An Experimental Study Using Auditory Stimuli," *PLOS ONE*, vol. 10, pp. 1–16, 2015.

[28] "ld.so(8) Linux Programmer's Manual," Linux man-pages project, 2016. [Online]. Available: http://man7.org/linux/man-pages/man8/ld.so.8.html

[29] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C. Maurice, and S. Mangard, "Practical keystroke timing attacks in sandboxed javascript," in *ESORICS'17*, 2017, (to appear).

[30] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *USENIX Security Symposium*, 2016.

[31] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P'15*, 2015.

[32] LWN, "The high-resolution timer API," Jan. 2006. [Online]. Available: https://lwn.net/Articles/167897/

[33] J. Ma, W. Yang, M. Luo, and N. Li, "A study of probabilistic password models," in *S&P'14*, 2014.

[34] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse Engineering Intel Complex Addressing Using Performance Counters," in *RAID'15*, 2015.

[35] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX ATC'96*, 1996.

[36] W. Melicher, B. Ur, S. M. Segreti, S. Komanduri, L. Bauer, N. Christin, and L. F. Cranor, "Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks," in *USENIX Security Symposium*, 2016.

[37] Microsoft, "Acpi system description tables," Jul. 2016. [Online]. Available: https://msdn.microsoft.com/en-us/windows/hardware/drivers/bringup/acpi-system-description-tables#madt

[38] A. Narayanan and V. Shmatikov, "Fast dictionary attacks on passwords using time-space tradeoff," in *CCS'05*, 2005.

[39] P. Norvig, "English letter frequency counts: Mayzner revisited," 2013. [Online]. Available: http://norvig.com/mayzner.html

[40] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications," in *CCS'15*, 2015.

[41] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *CT-RSA*, 2006.

[42] C. Percival, "Cache missing for fun and profit," in *Proceedings of BSDCan*, 2005.

[43] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security Symposium*, 2016.

[44] S. Pinet, J. C. Ziegler, and F.-X. Alario, "Typing is writing: Linguistic properties modulate typing execution," *Psychon Bull Rev*, vol. 23, no. 6, pp. 1898–1906, Apr. 2016.

[45] Qualcomm, "Snapdragon mobile processors and chipsets," Jan. 2017. [Online]. Available: https://www.qualcomm.com/products/snapdragon

[46] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds," in *CCS'09*, 2009.

[47] R. Shay, S. Komanduri, P. G. Kelley, P. G. Leon, M. L. Mazurek, L. Bauer, N. Christin, and L. F. Cranor, "Encountering stronger password requirements: User attitudes and behaviors," in *SOUPS'10*, 2010.

[48] P. Shrestha, M. Mohamed, and N. Saxena, "Slogger: Smashing Motion-based Touchstroke Logging with Transparent System Noise," in *WiSec'16*, 2016.

[49] L. Simon, W. Xu, and R. Anderson, "Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards," *Proceedings on Privacy Enhancing Technologies*, 2016.

[50] D. X. Song, D. Wagner, and X. Tian, "Timing Analysis of Keystrokes and Timing Attacks on SSH," in *USENIX Security Symposium*, 2001.

[51] The GTK+ Team, "GTK+ Features," 2016. [Online]. Available: https://www.gtk.org/features.php

[52] R. Veras, C. Collins, and J. Thorpe, "On semantic patterns of passwords and their security impact," in *NDSS'14*, 2014.

[53] P. Vila and B. Köpf, "Loophole: Timing attacks on shared event loops in chrome," in *USENIX Security Symposium*, 2017.

[54] R. Wash, R. Rader, R. Berman, and Z. Wellmer, "Understanding password choices: How frequently entered passwords are re-used across websites," in *SOUPS'16*, 2016.

[55] M. Weir, S. Aggarwal, B. d. Medeiros, and B. Glodek, "Password cracking using probabilistic context-free grammars," in *S&P'09*, 2009.

[56] M. Winnick and J. Mons, "Mobile touches: a study on humans and their tech," 2016. [Online]. Available: https://blog.dscout.com/mobile-touches

[57] X.org Foundation, "xorg documentation," 10 2014. [Online]. Available: https://www.x.org/wiki/Documentation/

[58] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014.

[59] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the Intel Last-Level Cache," *Cryptology ePrint Archive, Report 2015/905*, pp. 1–12, 2015.

[60] K. Zhang and X. Wang, "Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems," in *USENIX Security Symposium*, 2009.

APPENDIX

We compare the accuracy of four different side channels with and without *KeyDrown* (`procfs`, `rdtsc`, Flush+Reload, and Prime+Probe on the last-level cache) on three different architectures: a Lenovo ThinkPad T460s (Intel Core i5-6200U), an LG Nexus 5 (ARMv7), and a OnePlus 3T (ARMv8). Table IV summarizes the F-scores for all attacks with and without *KeyDrown*. *KeyDrown* prevents keystroke timing attacks in all cases when considering single-trace attacks only.

TABLE IV. F-SCORE WITHOUT AND WITH *KeyDrown* FOR STATE-OF-THE-ART ATTACKS.

Device	Side Channel	unprotected	*KeyDrown*
ThinkPad T460s	`procfs`	1.00	0.15
LG Nexus 5	`procfs`	1.00	0.15
OnePlus 3T	`procfs`	1.00	0.15
ThinkPad T460s	Interrupt-timing (`rdtsc`)	0.94	0.14
LG Nexus 5	Interrupt-timing	0.94	0.14
OnePlus 3T	Interrupt-timing	0.99	0.15
ThinkPad T460s	Flush+Reload	0.99	0.09
LG Nexus 5	Flush+Reload	0.99	0.02
OnePlus 3T	Flush+Reload	0.93	0.10
ThinkPad T460s	Prime+Probe on LLC	0.81	0.11
LG Nexus 5	Prime+Probe on LLC	0.80	0.11
OnePlus 3T	Prime+Probe on LLC	0.89	0.07

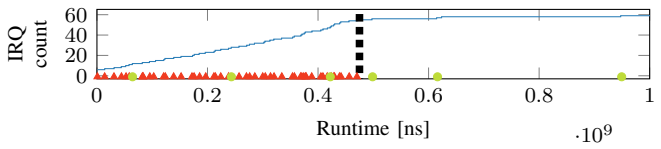

Fig. 13. `procfs`-based attack on the Nexus 5. Injected keystrokes (▲) and real events (●) are not distinguishable with *KeyDrown* (before dotted line).

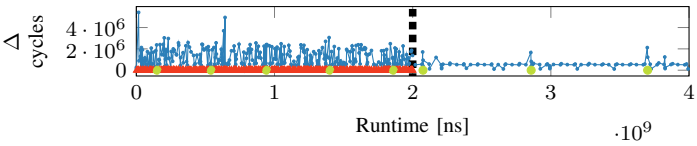

Fig. 14. Timing-based attack on the Nexus 5. Injected keystrokes (▲) and real events (●) are not distinguishable with *KeyDrown* (before dotted line).

We performed our experiments on the touchscreen soft-keyboard of the Nexus 5. With *KeyDrown*, the precision is lowered to 0.01 and, thus, the resulting F-score of 0.02 means a $\leq -86.5\,\%$ advantage over an always-one oracle.

Figure 13 and Figure 14 show a `procfs`-based interrupt attack and a timing-based attack, both on the Nexus 5. Without *KeyDrown*, we achieve a precision of 1.00 for the `procfs`-based attack and 0.89 for the timing-based attack, resulting in an F-score of 1.00 and 0.94 respectively. Enabling *KeyDrown* reduces the precision to only 0.08 and 0.07 respectively. Thus, the resulting F-score is 0.15 for the `procfs`-based attack, and 0.14 for the timing-based attack, which is an advantage of $\leq 0.0\,\%$ over an always-one oracle.

Figure 15 shows the results of inferring keystrokes by detecting the touchscreen interrupt handler's cache activity using Multi-Prime+Probe on the Nexus 5. We monitored 5 cache sets in parallel for noise robustness. Without *KeyDrown*, the precision is 0.71 with a recall of only 0.92, as an access to one of the cache sets by any other application cannot be distinguished from a cache set access by the touchscreen interrupt handler, resulting in a high number of false positives. If we enable *KeyDrown*, the precision drops to 0.06, as the attacker additionally measures the noise generated by the injected keystrokes. Thus, the F-score is 0.11.

We performed our experiments on the OnePlus 3T touch-screen soft-keyboard. Figure 16 shows a Flush+Reload attack on `libflinger.so`. Without *KeyDrown*, the precision is 0.88 and the F-score is thus 0.93. If *KeyDrown* is active, the precision is lowered to 0.05 and, thus, the resulting F-score of 0.10 means a $\leq -32.5\,\%$ advantage over an always-one oracle.

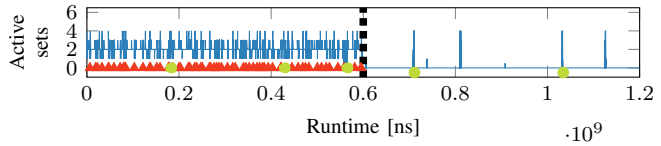

Fig. 15. Multi-Prime+Probe attack on the 5 cache sets from `0x382659be` to `0x38265abe` of `touch_irq_handler` on the Nexus 5. Injected keystrokes (▲) and real events (●) are not distinguishable when *KeyDrown* is active (before dotted line).

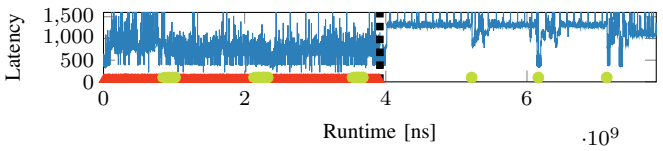

Fig. 16. Flush+Reload attack on address `0x28ec0` of `libflinger.so` on the OnePlus 3T. Injected keystrokes (▲) and real events (●) are not distinguishable when *KeyDrown* is active (before dotted line).

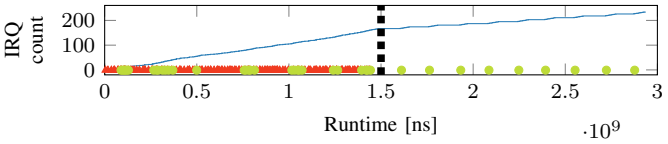

Fig. 17. `procfs`-based attack on the OnePlus 3T. Injected keystrokes (▲) and real events (●) are not distinguishable with *KeyDrown* (before dotted line).

Figure 17 and Figure 18 show a `procfs`-based interrupt attack as well as a timing-based attack, both on the OnePlus 3T. The attack has a precision of 1.00 (F-score of 1.00) and 0.99 (F-score of 0.99) respectively. Enabling *KeyDrown* reduces the precision to only 0.08 (F-score is 0.15) and 0.07 (F-score is 0.15) respectively, which is a $0.0\,\%$ advantage over an always-one oracle.

Figure 19 shows the results of inferring keystroke timings by detecting the touchscreen interrupt handler's cache activity using Multi-Prime+Probe on the OnePlus 3T. We monitored 5 cache sets in parallel for a higher noise robustness. Without *KeyDrown*, the precision is already at a quite low value of 0.80 with a recall of only 1.00, as access to one of the cache sets by any other application cannot be distinguished from a cache set access by the touchscreen interrupt handler. Thus, this attack has a high number of false positives. If we enable *KeyDrown*, the precision drops to 0.10, as the attacker additionally measures the noise generated by the injected keystrokes. Thus, the F-score is 0.07, which is a $\leq -52.7\,\%$ advantage over an always-one oracle.

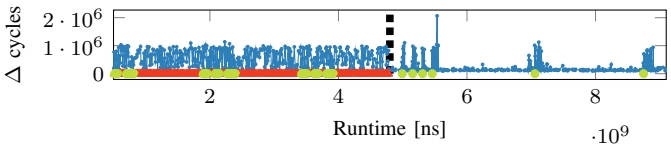

Fig. 18. Timing-based attack on the OnePlus 3T. Injected keystrokes (▲) and real events (●) are not distinguishable with *KeyDrown* (before dotted line).

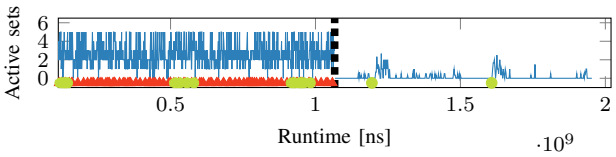

Fig. 19. Multi-Prime+Probe attack on the 5 cache sets from `0x3fc0355c28` to `0x3fc0355d68` of `msm_gpio_irq_handler` of the OnePlus 3T. Injected keystrokes (▲) and real events (●) are not distinguishable when *KeyDrown* is active (before dotted line).