RefreshChannels: Exploiting Dynamic Refresh Rate Switching for Mobile Device Attacks

Gaofeng Dong
University of California, Los Angeles
Los Angeles, California, USA
gfdong@g.ucla.edu

Jason Wu
University of California, Los Angeles
Los Angeles, California, USA
jaysunwu@g.ucla.edu

Julian de Gortari Briseno
University of California, Los Angeles
Los Angeles, California, USA
julian700@g.ucla.edu

Akash Deep Singh
University of California, Los Angeles
Los Angeles, California, USA
akashdeepsingh@g.ucla.edu

Justin Feng
University of California, Los Angeles
Los Angeles, California, USA
jfeng10@ucla.edu

Ankur Sarker
University of California, Los Angeles
Los Angeles, California, USA
as4mz@virginia.edu

Nader Sehatbakhsh
University of California, Los Angeles
Los Angeles, California, USA
nsehat@ee.ucla.edu

Mani Srivastava*
University of California, Los Angeles
and Amazon
Los Angeles, California, USA
mbs@ucla.edu

ABSTRACT

Mobile devices with dynamic refresh rate (DRR) switching displays have recently become increasingly common. For power optimization, these devices switch to lower refresh rates when idling, and switch to higher refresh rates when the content displayed requires smoother transitions. However, the security and privacy vulnerabilities of DRR switching have not been investigated properly. In this paper, we propose a novel attack vector called `RefreshChannels` that exploits DRR switching capabilities for mobile device attacks. Specifically, we *first* create a covert channel between two colluding apps that are able to stealthily share users' private information by modulating the data with the refresh rates, bypassing the OS sandboxing and isolation measures. *Second*, we further extend its applicability by creating a covert channel between a malicious app and either a phishing webpage or a malicious advertisement on a benign webpage. Our extensive evaluations on five popular mobile devices from four different vendors demonstrate the effectiveness and widespread impacts of these attacks. *Finally*, we investigate several countermeasures, such as restricting access to refresh rates, and find they are inadequate for thwarting `RefreshChannels` due to DDR's unique characteristics.

*Mani Srivastava holds concurrent appointments as a Professor of ECE and CS (joint) at the University of California, Los Angeles and as an Amazon Scholar. This paper describes work performed at the University of California, Los Angeles and is not associated with Amazon.

MOBISYS '24, June 3–7, 2024, Minato-ku, Tokyo, Japan

ACM ISBN 979-8-4007-0581-6/24/06
https://doi.org/10.1145/3643832.3661864

CCS CONCEPTS

• **Security and privacy** → **Mobile platform security**; **Side-channel analysis and countermeasures**; **Operating systems security**; **Browser security**.

KEYWORDS

Mobile Devices, Security and Privacy, Covert Channel, Dynamic Refresh Rate

ACM Reference Format:
Gaofeng Dong, Jason Wu, Julian de Gortari Briseno, Akash Deep Singh, Justin Feng, Ankur Sarker, Nader Sehatbakhsh, and Mani Srivastava. 2024. `RefreshChannels`: Exploiting Dynamic Refresh Rate Switching for Mobile Device Attacks. In *The 22nd Annual International Conference on Mobile Systems, Applications and Services (MOBISYS '24), June 3–7, 2024, Minato-ku, Tokyo, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3643832.3661864

1 INTRODUCTION

Mobile devices have revolutionized the way we work, travel, shop, and stay connected. The prevalence of mobile devices in all aspects of our lives means that they have access to vast amounts of sensitive information about us – from precise location to credit card information, making them a ripe target for malicious apps to steal and share such data [1–3]. Mobile OSs mitigate this issue by employing app *sandboxing* [4], which isolates different apps and their resources. Apps can still communicate with each other with technologies such as inter-process communications (IPC), but these methods are closely monitored by the OS and require the user's explicit approval or consent.

Unfortunately, malicious apps can utilize *covert channels* to discreetly transmit information without the system or user's knowledge [5–8]. These apps may be maliciously implanted by the adversary, or may also be benign but contain third-party malicious libraries. For example, Reardon et al. [9] discovered such vulnerabilities in third-party libraries provided by well-known companies.

With covert channels, a user's private data such as personal preferences, GPS coordinates, credit card numbers or persistent identifiers like IMEI can be shared with other parties while bypassing legal and security regulations [10–12]. Apart from apps, malicious webpages can also utilize covert channels to circumvent anti-tracking protections in browsers [13, 14]. For instance, Snyder et al. [15] use web covert channels to transmit a 35-bit string, which is sufficient for attackers to uniquely identify the 7.9 billion people on Earth.

Over the years, active measures have been taken to suppress known covert channels. For example, modern mobile OSs such as Android and iOS require permissions for an app to access sensitive sensors [16–18]. Moreover, the W3C, the Web's standardization body, has also recommended disabling sensor access on cross-origin iframes to limit attacks on webpages [19]. However, the introduction of new features often leads to unforeseen exploits and vulnerabilities that can circumvent existing countermeasures [7, 20–23]. In this work, we investigate a novel vulnerability surrounding the newly employed Dynamic Refresh Rate (DRR) switching technology in recent mobile devices, and show that the existing countermeasures are ineffective at mitigating this vulnerability.

To simultaneously achieve a good user experience and long battery life, DRR dynamically adjusts the refresh rate depending on the screen content – higher rates for dynamic content that requires smoothness and lower rates for static content when the device is idling. We find that both Android and iOS, the two most popular mobile OSs, provide permission-less access to refresh rates [24, 25], which can be exploited by malicious apps or websites. This vulnerability exists because running apps need to be able to suggest a preferred refresh rate, as well as monitor the current rate, for DRR to be effective. As DRR techniques are adopted by many mainstream smartphone and tablet vendors such as Apple and Samsung, it is crucial to thoroughly investigate the underlying security and privacy risks.

In this work, we demonstrate two novel privacy invasive attacks using the DRR switching technique, as shown in Fig. 1. *Firstly*, we construct an inter-app covert channel in Section 3 that two apps can use to communicate and bypass the app sandboxing and isolation measures imposed by the OS. We evaluate its effectiveness using five popular mobile devices including smartphones and tablets from four distinct vendors, i.e., Samsung, Google, Lenovo, and OnePlus. However, the refresh rate changes from normal user activities will interfere with our channel. Therefore, we identify the interference sources, propose an anti-interference scheme, and evaluate it with ten popular apps. *Secondly*, despite the lack of direct ways for webpages to access refresh rates, we build another covert channel between a malicious app and a phishing webpage or a malicious advertisement on a benign webpage, as discussed in Section 4. We assess this attack on four widely used browsers, i.e., Chrome, Firefox, Opera, and Samsung Internet. This broadens the flexibility and applicability of the first attack. *Third*, we discuss the challenges of mitigating these attacks and propose potential countermeasures in Section 5. As an attack vector, DRR can be exploited to conduct more attacks, as discussed in Section 6.

To the best of our knowledge, this is the first work showcasing DRR switching as a new attack vector, which we refer to as `RefreshChannels`. In summary, the contributions of this paper are:

- We investigate DRR switching strategies on mobile devices and explore numerous ways to modulate and monitor the refresh rate under differing scenarios.
- We demonstrate two proof-of-concept attacks that exploit DRR to circumvent the system's sandboxing and isolation measures to share data. The first attack builds a covert communication channel between two apps without triggering the system's alerts or users' awareness. Then, we show that even a webpage or a malicious advertisement on a benign webpage can covertly communicate with an app in the second attack.
- We implement the attacks[1] and evaluate them with extensive experiments using five popular mobile devices and four popular browsers.
- We discuss the challenges in mitigating these attacks and propose several countermeasures.

2 BACKGROUND

In this section, we present the background of DRR in Android OS, introduce the basic methods to affect and monitor the refresh rates, and provide an overview of the attacks. Android OS is the most popular open-sourced mobile OS, covering 72.37% of mobile OS market shares as of January 2023 [26]. Though we focus on Android mobile devices due to the greater reach, this work extends to iOS devices with DRR since they have similar DRR switching strategies. More details and some preliminary results of iOS devices will be discussed in Section 6.

Terminology. We first introduce the concepts of the refresh rate and frame rate, two terms frequently used in our work. The *refresh rate* refers to the frequency at which the display refreshes, often given in Hz. On the other hand, the *frame rate* refers to the actual frames per second (fps) that the computing hardware provides to be displayed on the screen. The system attempts to match the frame rate and refresh rate to optimize user experience.

Android graphics rendering. At a high level, the rendering pipeline consists of three main components: Image Stream Producers, Image Stream Consumers, and the Hardware Abstraction Layer (HAL) [27]. Android apps primarily utilize *Surfaces* as image stream producers, with *SurfaceFlinger* as the image stream consumer. The timing of the pipeline depends on the vertical synchronization (VSYNC) events, which indicates the time the display starts to refresh the display pixels [24]. Normally, the rendering must be completed within one VSYNC period (inverse of frame rate) to produce a new frame for the display. Many apps like games or video players have their own custom rendering pipelines, which means they can have their own preferred refresh rates. Therefore, it is necessary for apps to suggest their preferred refresh rates to the system.

Affecting and monitoring the refresh rate. To enable DRR, Android provides several APIs for apps to access the refresh rates [24, 28] directly. We utilize *surface.setFrameRate()* to set the refresh rate through a Surface. The Surface attempts to modify the VSYNC period, and if successful, *both* the refresh rate and the frame rate will change since the Surface will generate frames based on the new VSYNC. However, sometimes Surface-based methods don't work for some devices. In such cases, we turn to *Window*, which is

[1]The code will be available at https://github.com/nesl/RefreshChannels

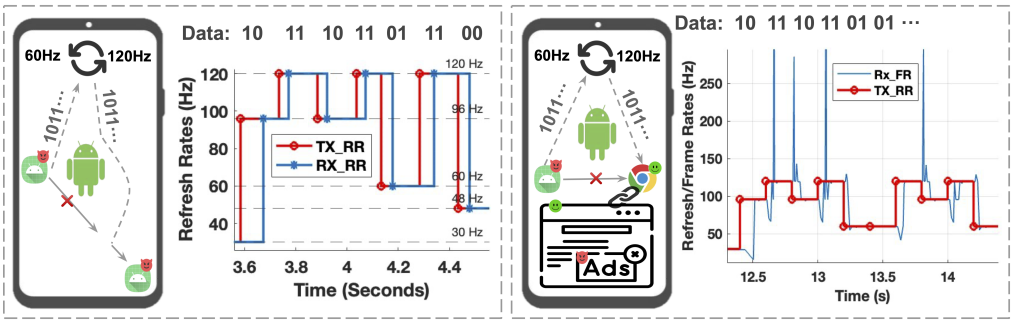

Figure 1: Two attack scenarios are demonstrated in this work. (a) Building an inter-app covert channel: the accompanying graph shows the refresh rates set by the transmitter (in red) and observed by the receiver (in blue). (b) Building a covert channel between an app and a webpage or an advertisement on the webpage: the accompanying graph shows the refresh rate set by the transmitter (in red) and the frame rate observed by the receiver (in blue).

a structure internally backed by a Surface. Each Window has a set of refresh rates, which can be chosen by a preferred mode.

To monitor the refresh rate, apps can use *AChoreographer_registerRefreshRateCallback* to register a callback to be run when the display refresh rate changes [29]. This provides the VSYNC period in nanoseconds, which can be used to infer the refresh rate. When the refresh rate is inaccessible or unresponsive, we infer the refresh rate using frame rates that are available through *Choreographer.FrameCallback* [30].

Overview: Apart from the user and OS, the primary parties involved in mobile devices are apps and webpages. These entities can influence the refresh rate and act as a transmitter (*Tx*) by either changing screen content or directly specifying the refresh rate through the API. Similarly, they can also read the refresh/frame rate with the same API and act as a receiver (*Rx*). With several possible *Tx* and *Rx* combinations across the parties, we broadly classify the potential covert channel attacks into two scenarios: intra-party covert channels (app to app) and inter-party covert channels (webpage to app and vice versa). We refer to these as Attack Scenarios 1 and 2, respectively. Furthermore, we also showcase the potential for DRR to leak private user touch information in Section 6.

3 ATTACK SCENARIO 1: BUILDING INTER-APP COVERT CHANNELS

The lack of appropriate access controls enables malicious apps to freely access refresh rates. We exploit this vulnerability to design a covert channel between two apps. Then, we evaluate it with five mobile devices from four mainstream vendors. Since channel interference can occur as the refresh rate changes over normal usage, we identify the interference sources, propose an anti-interference scheme, and evaluate it with ten apps.

3.1 Threat Model

Following previous works [8–12], we assume there are two apps on the victim's device that try to establish a covert communication channel bypassing the system's monitoring to share private user data. These malicious apps could enter the victim's system by masquerading as innocent apps or be downloaded through phishing links. Moreover, the apps can be benign and created by legitimate entities, yet may contain third-party libraries with malicious code from attackers. This security flaw has even been discovered in libraries developed by large companies [9]. As shown in Fig. 1 (a), one app will act as the transmitter, *Tx*, while the other acts as the receiver, *Rx*. *Tx* has the user's private data such as geolocation data or persistent identifiers like IMEI. However, it is barred from sharing this data with other parties due to legal or security concerns, forcing it to resort to covert channels. This private data can be encoded into binary bits, and then modulated on refresh rate changes via `RefreshChannels`.

Our attack utilizes apps' unrestricted ability to directly or indirectly modulate and monitor refresh rates to build the covert channel. To enable DRR, the Android OS allows apps unrestricted ability to suggest and read refresh rates. The actual frequency at which the refresh rate can be modified depends on the system, the hardware, and the specific method used to induce the change. Therefore, we design different schemes and evaluate them on different mobile devices. While apps do not need permissions to access refresh rates, mobile OSs restrict background app activities. Since bypassing background restrictions is not the focus of this work, we discuss several methods used in previous works in Section 6.

3.2 Channel Design and Implementation

3.2.1 Test devices and refresh rate information. We utilize five popular mobile devices with DRR, as shown in Table 1. To showcase the widespread impact of our attack across different hardwares, these devices encompass both smartphones and tablets and originate from four distinct vendors. The table showcases all the available refresh rates that can be set and monitored by apps. The test devices are operating on Android 12, the most current version of Android at the time of purchase, and we maintain the same version throughout the entire work to ensure consistent results.

3.2.2 Channel design. Since different devices have a varying number of available refresh rates, we design several basic modulation

Table 1: Devices and their available refresh rates.

Device	Refresh Rates (Hz)
Samsung Galaxy S22 Ultra	10, 24, 30, 48, 60, 96, 120
OnePlus 10 Pro	60, 90, 120
Google Pixel 6 Pro	60, 120
Samsung Galaxy Tab S7	60, 120
Lenovo Tab P12 Pro	60, 120

schemes tailored to specific device characteristics. On a high level, the modulation schemes transmit a basic packet of information consisting of a *SYNC* signal to begin transmission, a fixed-length *DATA* chunk, and an optional *END* signal to end the transmission. We encode the '0' and '1' bits as two distinct refresh rates, allowing us to send information by modulating between the two refresh rates. For devices with only two refresh rates, in order to differentiate the *SYNC* signal from the *DATA* signal, we utilize a unique pattern to indicate the *SYNC* signal. For devices with multiple refresh rates, we employ a more robust scheme where *SYNC* and *END* have distinct refresh rates from the '0' and '1' data bits.

Algorithm 1 Covert Channel Transmitter

Input: *SYNC* refresh rate(s) rr_{sync}, (*END* refresh rate rr_{end}), data $DATA$, bit '0' refresh rate rr_0, bit '1' refresh rate rr_1, interval I

1: Send *SYNC* signal by setting the refresh rate(s) rr_{sync}.
2: **for** data bit b in $DATA$ **do**
3: **if** b == '0' **then**
4: Set refresh rate to rr_0
5: **else if** b == '1' **then**
6: Set refresh rate to rr_1
7: **end if**
8: Wait for time I
9: **end for**
10: Send *END* signal by setting the refresh rate rr_{end} ▷ device-dependent

Algorithm 1 describes how the transmitter sends one data chunk. After the *SYNC* signal is sent in Line 1, the *DATA* is transmitted by sending one bit every I milliseconds, as shown in Lines 2-9. Here the interval I is the pulse width, which is chosen based on the bit error rate to maximize the channel capacity. Data transmission concludes with the *END* signal in Line 10 if the device has the *END* signal. For devices with only two refresh rates, 120 Hz (rr_1) is used to represent bit '1' while 60 Hz (rr_0) represents bit '0', and the *SYNC* pattern is the signal '10101'. This pattern is virtually impossible to encounter during normal usage since a benign app or the user must precisely switch the refresh rate at a fixed interval I four times in a row to mimic the *SYNC*. For devices with more than two refresh rates, such as the Samsung S22 Ultra, we modulate the *SYNC*, *END*, and *DATA* bits using different refresh rates. Since 24, 60, and 120 Hz are very common refresh rates that occur frequently during normal use, they are not suitable for the *SYNC* or *END* signals as they might cause mistriggers. In this basic scheme of Samsung S22 Ultra, we use 30 Hz (rr_{sync}) as the *SYNC* signal, 10 Hz (rr_{end}) as the *END* signal, 96 Hz (rr_0) as bit '0', and 120 Hz (rr_1) as bit '1'. For OnePlus, 90 Hz acts as the *SYNC*, 60 and 120 Hz as bit '0' and '1' respectively.

To improve the bandwidth, we can map a given refresh rate to multiple bits if the device has a sufficient number of refresh rates. The Samsung S22 Ultra contains seven refresh rates, so we propose an improved scheme where one refresh rate represents two bits. Specifically, 30 and 10 Hz are still the *SYNC* and *END* signal, but 48, 60, 96, and 120 Hz now represent bits '00', '01', '10' and '11'. This scheme can improve the bandwidth with higher channel resource utilization.

Algorithm 2 Covert Channel Receiver

Input: *SYNC* refresh rate(s) rr_{sync}, *END* refresh rate rr_{end} (or data length L), bit '0' refresh rate rr_0, bit '1' refresh rate rr_1, interval I

Output: data $DATA$

1: $DATA \leftarrow$ "", $synced \leftarrow$ `False`, $tLast \leftarrow$ currentTime, $rrLast \leftarrow 0$
2: **procedure** RRCallback(VSYNC period $vsyncPeriod$, Timestamp t)
3: Refresh rate $rr \leftarrow 1/vsyncPeriod$
4: **if** $rrLast == rr_0$ **then**
5: $bLast \leftarrow$ '0'
6: **else if** $rrLast == rr_1$ **then**
7: $bLast \leftarrow$ '1'
8: **end if**
9: $numPrevBits \leftarrow round((t - tLast)/I)$
10: Append $numPrevBits$ bit $bLast$ to $DATA$
11: **if** not $synced$ **then**
12: $dLast \leftarrow$ last $length(rr_{sync})$ characters of $DATA$
13: **if** $dLast == SYNC$ (or $rr == rr_{sync}$) **then** ▷ device-specific
14: $synced$ = True
15: $DATA \leftarrow$ ""
16: **end if**
17: **else if** $len(DATA) == L$ (or $rr == rr_{end}$) **then** ▷ device-specific
18: Save $DATA$
19: $synced$ = False
20: $DATA \leftarrow$ ""
21: **end if**
22: $tLast \leftarrow t$
23: $rrLast \leftarrow rr$
24: **end procedure**
25: `register(RRCallback)`

Algorithm 2 describes the overall structure of the receiver. The *RRCallback*() function is registered as a callback that triggers every time the refresh rate changes. It receives the VSYNC period and timestamp as parameters, which can be used to get the current refresh rate rr. Since it is only triggered at a refresh rate *change*, sending the same bit multiple times will not trigger the callback since rr will remain constant. As a result, at every refresh rate change, we decode the *previous* refresh rate $rrLast$ to data bit $bLast$, as shown in Lines 4-8. Then, Lines 9-10 calculate the number of previous bits by estimating how many intervals $rrLast$ was held over, and append it to *DATA*. Next, if *Rx* hasn't synchronized with *Tx*, it will keep checking in Lines 11-16. Note that if a device like Samsung S22 Ultra has a separate refresh rate rr_{sync} for the *SYNC* signal, it can directly compare to the refresh rate instead, as shown in Line 13. After synchronization, *Rx* will save the data when the length equals the predefined data length L, or when the refresh rate is the predefined *END* signal (Lines 17-21). Only the data received after the synchronization and before the end are saved as final *DATA* in Line 18. Otherwise, it will be cleared in Line 15 or 20.

3.2.3 Implementation. We build a pair of simultaneously operating *Tx* and *Rx* apps for each device with a previously determined time interval. As described in Section 2, the *Tx* app will use *surface.setFrameRate()* to set the refresh rate. This API works in all test devices except for the Oneplus 10 Pro, where it has no effect. A possible explanation could be the Oneplus phone not having support for these relatively new APIs. Instead, we utilize the *Window*

based method. The *Rx* app can register a callback using *AChoreographer_registerRefreshRateCallback* that triggers whenever the refresh rate changes. All the tested devices support this method of monitoring the refresh rate.

3.3 Evaluations

To show the effectiveness of the covert channel, we evaluate it using the five mobile devices in Table 1. For each device, we adjust the interval of the *Tx* and *Rx* apps to get the best capacity based on their raw bandwidth and bit error rate.

3.3.1 Capacity evaluation. In order to choose the best interval (I), we use channel capacity (C) as the metric, which indicates the upper bound on the information that can be reliably transmitted in a noisy channel. To simplify the calculation while retaining a good estimation of the capacity, we assume the channel is a binary symmetric channel (BSC) [31–33]. C can be calculated using the following equations:

$$C = \frac{n}{I} \times (1 - H(P_e)), \quad (1)$$

$$H(P_e) = -P_e \times log_2(P_e) - (1 - P_e) \times log_2(1 - P_e), \quad (2)$$

where n is the number of bits represented by one refresh rate, I is the interval, $H()$ is the binary entropy function, and P_e is the bit probability of error obtained by dividing the number of erroneous bits by the total number of transmitted bits (10,000). $\frac{n}{I}$ is the raw bandwidth BW.

The bit error probability or bit error rate (BER) is used to quantify errors induced by noise, e.g., inaccurate timings. As the raw bandwidth increases, BER also increases because the refresh rate has less time to be properly set and read. Fig. 2 shows the capacity across the five test devices. In Table 2, we list the maximum and zero-BER capacities for each device. The '1Rate2Bits' scheme, where 1 refresh rate represents 2 bits, of the Samsung phone achieves the highest capacity of 30.8 bps, while the Lenovo tablet has the worst capacity of 5.6 bps. This is still enough to send short private user data like IMEI, GPS coordinates, or credit-card numbers [7–9, 11, 12, 34]. Shepherd et. al built covert channels using sensor multiplexing [8], and obtained a maximum bandwidth at zero-BER of 9.62 bps across all tested devices and sensors. Our best zero-BER capacity is 22.2 bps, more than twice that of their channels.

The '1Rate2Bits' scheme has an improvement of 25% in best capacity over the '1Rate1Bit' scheme, where 1 refresh rate represents 1 bit. Since the two-bit scheme involves switching between four different refresh rates instead of two, there is additional complexity in setting the bit interval. The interval required to transition between refresh rates varies significantly. For example, it may require 50 ms to switch from 48 Hz to 60 Hz but 130 ms from 48 Hz to 120 Hz. Therefore, the BSC model may not be well-suited for the '1Rate2Bits' scheme, and setting different intervals for different refresh rate transitions could optimize channel utilization. We leave this as future work to explore.

3.3.2 Interference. Apart from noise-related errors, measured by BER, we also identify some interference sources that may directly affect refresh rates. We first analyze their effects on our channel and then perform a study with ten popular apps to evaluate the proportion of time with active interference.

Table 2: The maximum and zero-BER capacities of the covert channel on different devices.

Device	Capacity (bps)	
	Maximum	Zero-BER
Samsung S22 Ultra (1Rate2Bits)	30.8	22.2
Samsung S22 Ultra (1Rate1Bit)	24.7	14.3
Google Pixel 6 Pro	20.5	11.1
Samsung Tab S7	13.6	11.1
Oneplus 10 Pro	10.5	5.6
Lenovo Tab P12 Pro	5.6	5.6

Brightness. Smartphones fix their idle refresh rate at 120 Hz in low-brightness environments. Despite this, our channel can still modulate the refresh rates as usual, indicating that the APIs have priority over brightness-related settings.

Touches. There are two types of touches a user can perform: touches that cause the screen content to change and touches that do not affect any content, such as a simple screen tap. We will henceforth refer to the former as "dynamic touches" and the latter as "static touches". For both types of touches, the refresh rate will increase to 120 Hz. However, we observed that static touches have no effect on our channel. In Fig. 3, the four grey blocks indicate four static touches while the *Tx* and *Rx* apps are communicating. The refresh rates received by *Rx* (blue lines) are identical to the ones set by *Tx* (red lines), which indicates that data was transmitted correctly in the presence of static touch interference.

Dynamic content across different apps. However, dynamic touches like pressing a button or scrolling will interrupt the channel, causing the refresh rate to jump to 120 Hz for a brief period. Other dynamic content such as pop-up notifications and animated advertisements will also briefly set the refresh rate to 120 Hz and 60 Hz respectively.

Interference from dynamic content varies across different apps due to their unique usage patterns and animations. For example, a user may touch the screen less frequently when using e-book apps compared to browsing various options on food delivery apps. Similarly, the prevalence of dynamic content interference also varies in different apps, e.g., more animated content in social media apps while less in e-book apps. Therefore, we choose ten from the most popular apps (Table 3) based on AndroidRank [35] to evaluate how interference from different app usage would affect the channel concurrently running in the background.

We test these apps using the Samsung phone, but the results should apply to other devices since the interference is predominantly caused by dynamic touches and content, which is influenced by app and user behavior rather than the device itself. The only exception is Lenovo tablet, which maintains the refresh rate set by *Tx* regardless of interference. Thus, although Lenovo tablet has the worst capacity among the tested devices, its robustness to interference compensates with more available transmission time.

We record ten minutes of refresh rate data during normal app usage, while the *Tx* app alternates the refresh rate between 48 and 96 Hz. When interference exists, the refresh rate will be forced to 60

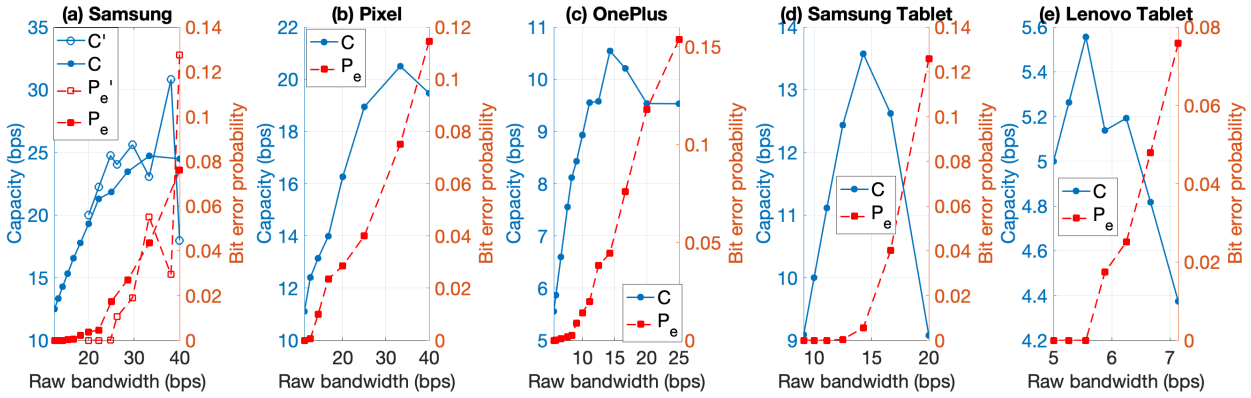

Figure 2: The capacities C and bit error probabilities P_e of the five test devices. In (a), C and P_e are for the '1Rate1Bit' scheme while C' and P'_e are for the '1Rate2Bits' scheme.

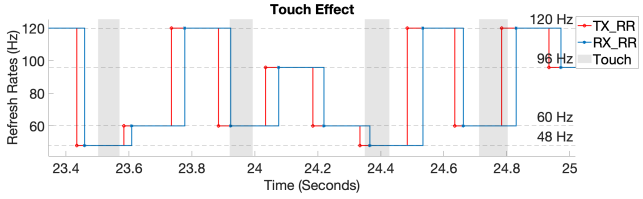

Figure 3: Robustness of *Tx* refresh rate to static touch

or 120 Hz. Otherwise, we read the 48 or 96 Hz set by *Tx*. Therefore, the percentage of 48 and 96 Hz shows the available time for the covert channel to send data.

Table 3: Percentages of available time for the covert channel.

App Name	Percentage
Google Play Books	96%
Google News	82%
Reddit	70%
Amazon Shopping	65%
Gmail	61%
Play Store	60%
Twitter	58%
Telegram	54%
Messenger	49%
Uber Eats	48%

Table 3 showcases that the percentage of available time for the ten apps ranges from 48% for Uber Eats to 96% for Google Play Books, with an overall average of 64%. This indicates that *Tx* can find ample available time to send data, even when the victim is actively using other apps that may interfere with the channel.

3.3.3 Anti-interference scheme and evaluation. Apart from showing there is enough available time for the covert channel despite interference, we also provide an anti-interference design. We evaluate its throughput based on simulation.

The basic idea is that *Tx* can monitor the same refresh rates using the same methods as *Rx*. Therefore, whenever *Tx* reads a different refresh rate than it attempted to set, indicating the presence of interference, it will try to send the packet again until the data is correctly transmitted. The state machine of the anti-interference '1Rate1Bit' transmitter is shown in Fig. 4. It's possible to further improve the scheme using methods like error correction, which can be explored in the future. The symbols used here are the same as those in Algorithms 1 and 2. The three main states are *S-SYNC*, *S-DATA* and *S-END*, where *Tx* tries to send *SYNC*, *DATA* and *END* signals respectively. The scheme of *Rx* is the same as Algorithm 2. Each time the *SYNC* signal is received, *DATA* is cleared. *DATA* is only saved upon receiving the *END* signal

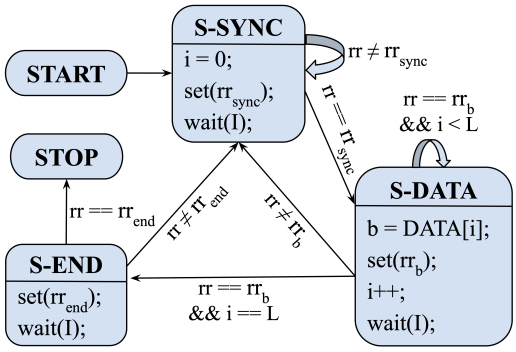

Figure 4: State machine of the anti-interference scheme

To evaluate the throughput T under interference, we also need to consider the interval I and packet length L'. As shown in Section 3.3.1, the optimal I of Samsung phone is 30 ms with a bit error probability P_e of 4.36% and capacity of 24.7 bps. The optimal L' is affected by the frequency of the interference, i.e., it's better to choose a shorter L' when interference happens frequently. But if it's too short, the control bits will occupy a significant portion of the bandwidth. Therefore, it becomes an optimization problem to find the optimal packet length L' to maximize the throughput T given the interference.

To evaluate the effect of changing L', the interference should be kept the same when testing different L'. However, it's impossible

for the user to interact with the apps in exactly the same manner to generate the same interference for different L'. Here, we choose to use the refresh rate data from the same experiment as described in Table 3 and their timestamps to simulate the interference pattern of a user interacting with the ten different apps, and then calculate the throughput based on this simulation. Specifically, with the timestamp of each refresh rate, we can calculate each duration D of the continuous available time that is not interrupted by 60 or 120 Hz. We denote all available durations as Set $\mathbb{D}$. With a data length L and two control bits (*SYNC* and *END*) per packet, the total packet length L' would be $L + 2$. Therefore, the number of data bits that can be transmitted in D is $L * \lfloor \frac{D}{(L+2)*I} \rfloor$. We use the floor function to get how many complete packets can be transmitted successfully within D because the last packet will be interrupted in the middle of transmission and should be disregarded. Therefore, by traversing L (or L'), we can find the optimal L that can maximize the total data bits $Nbits$ which is $\sum_{D \in \mathbb{D}} L * \lfloor \frac{D}{(L+2)*I} \rfloor$ that can be transmitted in ten minutes. Then the raw throughput T will be $\frac{Nbits}{600}$ bps. Factoring in the transmission bit error, the theoretical effective throughput T' can be estimated as $T \times (1 - H(P_e))$.

The simulated results are shown in Table 4. The theoretical effective throughput T' under the interference of the ten apps ranges from 10.1 bps for Uber Eats to 22.8 bps for Google Play Books. These results are also consistent with Table 3 - more available time results in greater effective bandwidth. When there is less interference, L' will be longer and T' will be higher. Therefore, *Tx* may dynamically adjust L based on current interference instead of using a fixed L'. The 'Overall' category is evaluated on the combination of all the refresh rate data from the ten apps, rather than simply averaging all the individual L' and T' values. The overall T' is 14.24 bps. These results show that even in the presence of interference, an attacker can find sufficient time and achieve a practical throughput to transmit short sensitive data.

Table 4: Optimal packet length and theoretical throughput of the anti-interference scheme

Interfering App	Optimal L'	T' (bps)
Google Play Books	94	22.8
Google News	98	19.1
Reddit	43	15.9
Amazon Shopping	36	14.5
Play Store	44	13.4
Gmail	32	13.3
Twitter	37	12.9
Telegram	30	11.7
Messenger	30	10.5
Uber Eats	24	10.1
Overall	38	14.2

4 ATTACK SCENARIO 2: BUILDING COVERT CHANNELS BETWEEN APPS AND WEBPAGES

In Section 3, we built a covert channel between two apps by exploiting DRR to bypass system monitoring. In this section, we expand the scope of RefreshChannels to webpages. Launching attacks from webpages is generally more challenging than from apps, as attackers have fewer resources at their disposal and face additional restrictions imposed by browsers [14, 15, 19, 36]. Despite the challenge that webpages cannot access refresh rate directly, we build covert channels between an app and a webpage that may be directly (e.g., a phishing link) or indirectly (e.g., as an advertisement that runs in benign webpages or apps) accessed by a user, as shown in Fig. 1 (b). We describe two specific attacks and their threat models where our malicious webpage can act as either side of the covert communication channel, and evaluate them on five mobile devices and four browsers.

4.1 Webpage as *Rx*

4.1.1 Threat Model. In this scenario, an app running on the victim's mobile device works as the *Tx*. It can access sensitive data and attempts to transmit the data covertly to avoid legal complications arising from direct transmission. The app is either malicious or has unintentionally incorporated malicious third-party libraries, as discussed in Section 3.1. To receive the data covertly, the attacker may use a phishing link to trick the victim into accessing the malicious webpage, which functions as the *Rx*. Aside from phishing links, attackers can also embed malicious code in advertisements or analytics services that can run in benign webpages or apps, as has been done in sensor-related attacks [19, 36, 37]. For brevity, we will henceforth use 'webpage' to represent all these possibilities. This allows our code to be present on thousands of websites or apps, extending the range and impact of our attack far beyond the original covert channel.

4.1.2 Channel Design and Implementation. The webpages were implemented using HTML and JavaScript and run in an *iframe* widely used in related work [19, 36]. Since there are no APIs for webpages to read the refresh rate, we use frame rates to infer refresh rates. To get the frame rates, we utilize the function *requestAnimationFrame* that is present in all tested browsers' JavaScript APIs, as per the W3C recommendation [38]. The webpage calculates the elapsed time between the callbacks to derive the frame rate. The design and implementation of the *Tx* app is the same as Section 3.2.3.

4.1.3 Evaluation. We use the same metric, i.e., channel capacity, as in Section 3.3.1 to evaluate the channel by sending 1,000 bits of data. The received frame rates are very noisy to demodulate directly, as shown in Fig. 6. Based on the observation that most of the noise appears at the rising and falling edges with very large or small values, we choose the median filter to remove the noise. The processed traces are much cleaner and easier to demodulate, as shown in the figure.

Evaluations on different devices. We test our attack across all five devices, each installed with Google Chrome version 107.0.5304.105, which was the latest version when our experiments began. Fig. 5 shows the capacities of different devices when the webpage acts as *Rx*. Both Samsung phone and tablet can reach around 14.4 bps. In [39], memory access times are utilized to transmit information from a background process to a JavaScript program executing in a web browser, where the bit rate they achieve is around 11 bps on

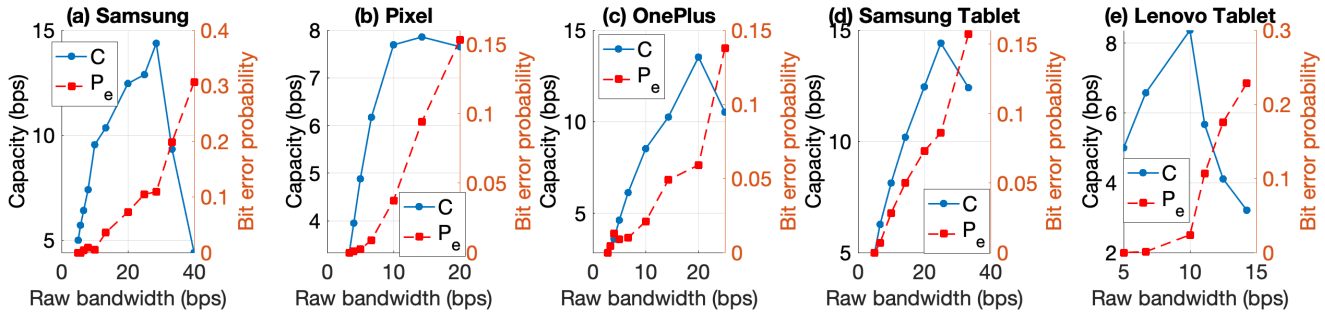

Figure 5: The capacities of different devices when the webpage is *Rx*.

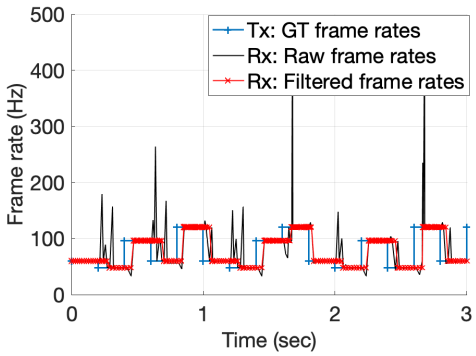

Figure 6: Ground-truth (GT) refresh rates of *Tx* app, raw and filtered frame rates of *Rx* webpage on Samsung S22 Ultra

an Intel Core i5 computer. `RefreshChannels` achieves comparable performance to it, but on a mobile device.

Evaluations on different browsers. We also extend our evaluations to different browsers as well. We chose four of the most popular browsers on mobile devices [40]: Google Chrome (107.0.5304.105), Firefox (107.1.0), Samsung Internet (19.0.1.2), and Opera (72.3.3767.68685). To illustrate the effects of different browsers, we conduct tests exclusively on the Samsung phone, and only the zero-BER bandwidths are reported in Table 5, though the maximum capacity will be higher. The result demonstrates that the choice of browsers affects zero-BER bandwidth. It is possible that different browsers add varying amounts of noise to counteract timer-based attacks by reducing timestamp resolution and adding jitter to increase uncertainty [41].

Table 5: Zero-BER bandwidths of different browsers on Samsung when the webpage runs as *Rx*.

Browser	Pulse width (ms)	Bandwidth (bps)
Chrome	350	2.86
Samsung Internet	400	2.50
Firefox	200	5.00
Opera	450	2.22

4.2 Webpage as *Tx*

4.2.1 Threat Model. In this scenario, a webpage has data that it needs to send to a background app without raising alerts. For example, although browser vendors have introduced private browsing modes to prevent user tracking, it is still possible for the webpage to circumvent such protections by exfiltrating user tracking identifiers to an app using covert channels [13–15]. We propose a method that enables a webpage to affect the refresh to transmit data to an app monitoring these rate changes.

4.2.2 Channel Design and Implementation. Based on the observation that dynamic content on the screen affects the refresh rates, the webpage changes graphical elements or text on the page to achieve this effect. We found that even a single-pixel difference can affect the refresh rate, which is undetectable to the victim. When there is a bit '1' to transmit, the webpage will change the content for a period to induce a higher refresh rate. Similarly, when there is a bit '0', it will remain static to allow the refresh rate to fall to the idle rate.

When the webpage acts as a transmitter, the app needs to run as a receiver by monitoring the refresh rates. However, the refresh rate callback functions previously used in Section 3.2.3 are often unresponsive to refresh rate changes induced by changing content. To solve this challenge, we use the frame rate to infer the refresh rate. We leverage the Android SDK *Choreographer.FrameCallback* interface to extract the timestamp of each frame and calculate the frame rate using the time elapsed between frames.

4.2.3 Evaluation. The frame rates are shown in Fig. 8 when the webpage is *Tx*. For Samsung and Pixel, the frame rates increase to 120 fps while the Samsung Tablet and Lenovo Tablet have small sharp fluctuations in frame rates. In both cases, the attacker is capable of encoding information in the frame rate. Additionally, we also demonstrate this attack across the four browsers from Section 4.1.3 on Samsung phone. To evaluate the capacity, the webpage transmitted 1,000 random bits to the receiver app. As shown in Figure 7, the maximum capacity is 2.2 bps when using Google Chrome. It is lower than the capacity of the "app-to-app" scenario in Section 3.3.1 because not only must webpages indirectly affect the refresh rate by changing pixels on the page, the Rx app also uses noisy frame rates to infer the refresh rate. Matyunin et al. use the CPU load to build the covert channel [14], achieving a bandwidth of 5-8 bps with a BER around 10% which translates to a capacity of 2.7-4.2 bps. Our

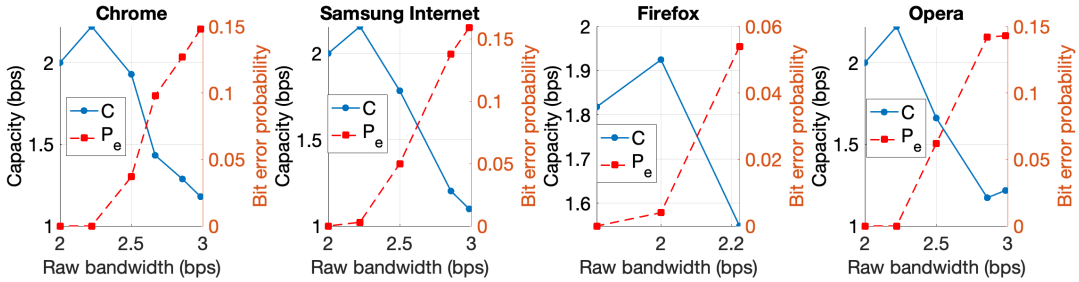

Figure 7: The capacities of different browsers on Samsung when the webpage is *Tx*.

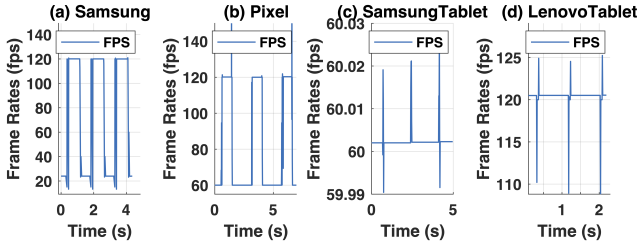

Figure 8: Frame rates of the devices when webpage is *Tx*.

attack achieves a similar performance. Therefore, the channel can be used to transmit short private data like a 35-bit user identifier under the average page dwell time of around a minute [15, 42].

5 COUNTERMEASURES

In this section, we discuss some potential countermeasures. While certain strategies effective against sensor-based attacks may also be applied to `RefreshChannels`, there are unique challenges that complicate full prevention. Consequently, mobile device vendors need to pay particular attention to these issues. We focus on analyzing the potential countermeasures qualitatively, and leave quantitative evaluation as future work.

Elevated usage privileges. A straightforward defense involves requiring elevated permissions to access refresh/frame rate APIs, similar to the approach taken for sensitive sensors [16–18]. However, many legitimate apps need to monitor and change the refresh/frame rate, which inconveniences users with additional permission requests. Not only is it inconvenient, but access control is also not fully effective, as attackers can still indirectly modulate the refresh/frame rate by changing content on the screen without accessing the APIs, as shown in Section 4.2.2. Another choice is to disallow background apps from accessing refresh or frame rates, which is effective at the expense of legitimate apps' flexibility and functionality.

Attack scope reductions. Instead of directly introducing access control and hurting user experience, it's possible to limit the effectiveness of `RefreshChannels` in several ways.

First, the frequency at which an app can access refresh or frame rate could be restricted. For instance, Android 12 has introduced the HIGH_SAMPLING_RATE_SENSORS permission [18], which elevated the difficulty of launching sensor-based attacks as most legitimate apps do not require high sensor sampling rates. Similarly, normal apps are unlikely to frequently change the refresh rate within a short period. However, directly applying the frequency limitation to refresh or frame rate APIs to choke the covert channel bandwidth encounters the aforementioned exploit of modulating the refresh rate by modifying screen content. Hence, it would be more effective to limit the refresh rate change frequency at the system or hardware level, though this introduces additional complexity. Choosing the best threshold is a tradeoff problem between DRR usability, available refresh rate numbers, available idle time, and private data length, which future work can explore.

Next, to address the exploit of changing screen content, a minimum content size threshold for influencing the refresh rate could be established to reduce the stealthiness of the attack. In both attack scenarios, a single-pixel surface or screen content change suffices to modulate the refresh or frame rate, and is impossible to detect by the user. The advantage of this approach is the minimal impact on user experience, as minor content changes involving a few pixels occur infrequently and are rarely significant enough to be perceived, though this can induce extra overhead.

Finally, one can introduce delays and randomization to reduce the capacity of the attack. Unfortunately, these delays will also harm user experience, app flexibility, and battery life.

Suspicious app behavior detection. The two covert channel attacks rely on setting the refresh/frame rates in an unnatural pattern to send *SYNC*/*END*/*DATA* signals. One possible approach involves detecting this abnormal behavior and triggering defenses to interfere with the covert communication, akin to the approaches taken for sensitive sensors [2, 43–46]. Power consumption-based detection methods can also be applied here, but it is challenging to achieve low false positive and negative rates, as malicious apps can compress private data into a few bits, encode wisely to minimize the time the refresh rate is modulated, adapt to produce benign-looking patterns that can be hidden among normal activities, or exfiltrate at a lower rate if all fails.

6 DISCUSSION AND FUTURE WORK

This work provides novel insights into using DRR switching to launch covert channel attacks on mobile devices. Despite some limitations, we believe these attacks open up some promising research directions.

Affected devices. Any mobile devices with DRR may be potentially affected by `RefreshChannels` because underlying DDR switching

principles are similar across devices. Mainstream mobile device vendors, including Apple, Samsung, Google, Oneplus, and Lenovo, have adopted DDR and list it as one of their devices' key features. Though we focus on attacks on five Android devices from four vendors, we also demonstrate the potential to similarly attack iOS (Apple) devices, which recently introduced ProMotion displays with dynamic refresh rates. Our preliminary tests on an iPhone 13 Pro show that the *CADisplayLink* [25] object can suggest the refresh rates, indicating the feasibility of building covert channels on iOS devices as its DRR switching strategy is similar to Android. Touch-induced frame rate changes can also be recorded through browsers like Safari and Chrome, as shown in Fig. 9. This suggests the potential for keystroke inference attacks on iOS devices, which can be explored in future work.

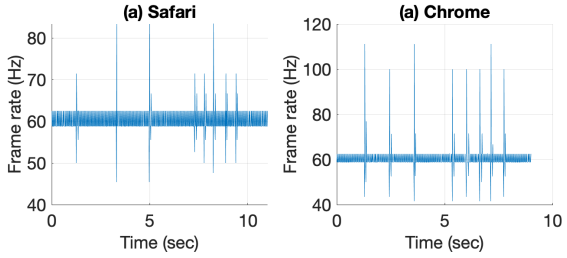

Figure 9: The frame rates recorded using Safari and Chrome on iPhone 13 Pro for three normal touches and five quick touches.

Stealthiness. Given that a change in a single pixel is sufficient to alter the refresh rates, it is challenging for the victim to notice such a pixel-level difference. Furthermore, the attack would typically be deployed when the screen content is idle/static, so it will not cause screen tearing, touch response slowdown, or other noticeable fluctuations in user experience. To evaluate its stealthiness, we conducted a user study[2] with ten volunteers using the Samsung smartphone. In this study, we designed two scenarios: the "static" scenario, where the user can only observe the screen without touching it, and the "interactive" scenario, where the user can use the phone normally. For each scenario, two sessions are shown to the participant: one in which `RefreshChannels` is active, while it remains idle in the other. We randomized the order of these two sessions and then asked the users if they noticed any user experience differences between the two sessions. If the answer was "yes," we asked the users to explain the differences they noticed and identify the session with the active attack. The procedure was repeated three times for each scenario. The results showed that no differences were identified in the thirty runs of the "static" scenario. In the "interactive" scenario, users believed that they perceived differences in only three out of the thirty runs: slowdowns when opening an app, changes in smoothness when playing videos, or reduced responsiveness when scrolling webpages quickly. However, two of these three answers were incorrect, i.e., the attack was not active in the session where they perceived the worse experience. We believe these differences were either imagined or caused by occasional system performance fluctuations, as similar differences were not observed in previous or subsequent runs. This user study demonstrates that `RefreshChannels` cannot be detected by users in both static and interactive scenarios, even when the user is aware that the attack was launched in one of the two sessions, showing its stealthiness.

[2]The study was approved by our institution's internal review board (IRB).

More advanced modulation schemes. Based on the number of available refresh rates on different devices, we designed several basic modulation schemes in Sections 3.2 for the proof-of-concept purpose. Future work can explore more advanced schemes that can fully exploit the channel resources, coupled with proper error correction codes like forward error corrections to correct errors.

Potential affected private data. As general covert communication channels, our proposed attacks can be used to send any short private user data, such as IMEI, GPS coordinates, or credit card numbers, as shown in previous works [7–9, 11, 12, 34]. A 35-bit string, sufficient to uniquely identify the approximately 7.9 billion people on the planet [15], can be transmitted within seconds using our channel. This transmission time is less than the average page dwell time, which is slightly under a minute [42].

Keystroke inference attacks. Timing side-channel attacks ultilize the time difference of different operations to infer cryptographic keys or private data [46–51]. Recent attacks show that the inter-key-interval (IKI) between two touches can leak the keystrokes, as a longer interval implies that two keystrokes are further apart on the keypad [52–55]. The adversary's goal is to infer the victim's secret PIN by eavesdropping on the timings of the keystrokes. Here, we develop an app with soft numeric keyboards with a classic layout as considered in prior works [52–54]. Fig. 10 shows the frame rates of three PIN touches on the five mobile devices we tested. When the user taps a PIN button, the frame rate will increase from idle rates to around 120 fps or generate peaks around 120 fps, which indicates the keystroke timings. After obtaining the timings, the possible PIN pairs can be inferred by modeling the entire PIN entering process with a Hidden Markov Model (HMM) [51, 54]. For the lock screen's PIN input, though the refresh rate remains fixed, we noticed some fluctuations in the frame rate around 120 fps on the Pixel 6 Pro, similar to Fig. 10 (b). These fluctuations, although challenging to detect, suggest that hardware behavior varies slightly in response to PIN input animations. Future works can further explore this vulnerability.

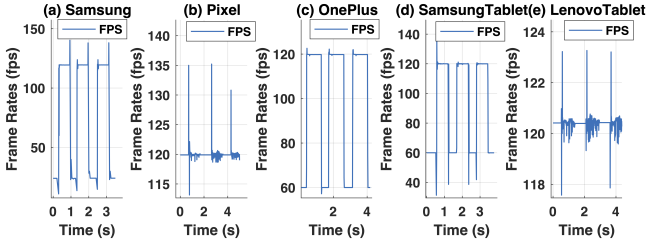

Figure 10: Frame rates when pressing PIN buttons of an app on different devices.

Fingerprinting/Tracking attacks. Different apps and websites may exhibit unique refresh/frame rate patterns depending on user

interaction and dynamic content, exposing them to fingerprinting attacks by recording refresh/frame rates. Apart from identifying apps or websites, users can also be identified by the unique properties of their browser, system environment, or hardware [14, 15]. Therefore, refresh/frame rates may also be used to track web browser users.

Background app execution. Though there exists no access control around refresh/frame rate, Android limits apps' background running time to minutes, which can shorten the attack time window. Note that this is a general challenge for mobile device attacks. First, short private data like IMEI is still possible to be shared within seconds or minutes. Besides, private user data, such as IMEI and credit card numbers, is usually fixed and will not change over months or years. Therefore, launching the attack successfully once is enough. Also, the malicious app can masquerade as a legitimate app to have a longer lifespan, since the Foreground Service permission is automatically granted and the notification can be obfuscated, as shown in recent attacks [8, 56]. To keep Surfaces alive when running in the background, the app can switch to overlay mode by disguising itself as a normal app, as more than 35% of the most popular apps use it to implement key features and it is even automatically granted in certain cases [21, 57, 58]. Besides, even without overlay mode, it can subtly alter the color or a single pixel of a silent notification icon to induce frame rate changes, which cannot be noticed by human eyes to keep its stealthiness. More methods can be explored in future work.

7 RELATED WORK

In this work, we demonstrate two attack scenarios on mobile devices, namely inter-app and app-web covert channels. Therefore, we first summarize previous works related to these attacks. Additionally, considering `RefreshChannels` as a potential attack vector that may enable more attacks, we draw comparisons with similar attack vectors, such as sensor-based and micro-architectural attacks, to show its distinctive features and potential implications within a wider spectrum of mobile security and privacy research. Again, given the extensive scope of research in this field, a comprehensive discussion within this paper is unfeasible. Therefore, we focus on examining studies that are especially significant, novel, and relevant.

Inter-app covert channels on mobile devices. Inter-app covert channels in mobile devices have been studied extensively over the last decade and have been shown to pose a tangible security and privacy threat to mobile users [8–12]. Some attacks use physical transmission media. For example, Novak et al. [34] built covert channels using light, while Block et al. [12] used ultrasonic frequencies, and Masti et al. [11] modulated processor core temperatures on multi-core platforms. Other attacks exploit side effects of the software interface. For instance, Soundcomber [7] utilized vibration/volume settings, and Shepherd et al. [8] modified the sampling rate of on-device sensors. Our first attack is based on affecting and accessing the refresh rate, so it belongs to the latter category.

App-Web covert channels on mobile devices. Webpages have been used as transmitters in covert channels. Matyunin et al. [14] used CPU load to secretly communicate data from a web browser to a background app. The website uses CPU-intensive operations to encode data, and the receiver app measures the execution time of a code fragment to gauge the CPU load and decode data. Webpages can also act as the receiver side of a covert channel. In [39], memory access times are utilized to transmit information from a background process in a computer to a JavaScript program executing in a web browser. They rely on the difference in access time between cache hits and misses to transmit information. ARM-based mobile devices have been found to be vulnerable to this type of cache-based covert channel [59], and are likely vulnerable to an attack similar to [39]. With `RefreshChannels`, a webpage can act as either side of the covert communication channel by affecting or monitoring the refresh/frame rate.

Sensor-based attacks on mobile devices. Sensors can be used to launch a series of attacks, such as covert channel attacks [7, 60], side channel attacks [23, 61], and fingerprinting attacks [62, 63]. Due to such threats, access to sensors has been strictly restricted on mobile devices. For example, the OS will require the user's explicit permission to access sensitive sensors like microphones. Besides, the W3C has also recommended disabling sensor access on cross-origin iframes, which limits sensor-based attacks on webpages [19]. However, as a new attack vector based on DRR, `RefreshChannels` may bypass these restrictions. Therefore, the insights provided by this work can help design more secure web standards and mobile devices that are resistant to `RefreshChannels`.

Microarchitectural attacks on mobile devices. Microarchitectural attacks are very powerful by exploiting vulnerabilities in the microarchitecture of modern computer processors [64–66]. However, they are also difficult to implement due to the complex design of modern microarchitectures, lack of public documentation, noise introduced by other system activities, etc [67–69]. What's more, such attacks are even harder on mobile devices [70, 71]. On the contrary, our attack only requires basic knowledge of app development and uses public APIs, making it easy to implement. Such attacks are often referred to as OS-level attacks [71]. Besides, future work can explore combining `RefreshChannels` with microarchitectural attacks to create new attacks or improve existing ones.

8 CONCLUSION

Modern mobile devices require strong security as they host an increasing amount of private data and services. This paper investigates a novel vulnerability, `RefreshChannels`, that facilitates different attack scenarios on mobile devices utilizing dynamic refresh rate switching. In the first attack, we showed that two apps are able to communicate using refresh rates, bypassing the OS sandboxing and isolation measures. Then, we demonstrated that even a malicious advertisement displayed on a webpage can covertly communicate with another app without any user awareness, which allows the proposed attacks to be more widespread. We implemented these attacks on five popular smartphones and tablets from four different vendors and also evaluated the second attack on four popular browsers, underlining the universality of our approach, which could potentially affect millions of users. Our findings indicate that the proposed attack vector is a threat to mobile devices, and we discussed several countermeasures to mitigate the proposed attacks. In the future, we will explore the possibilities of keystroke

inference and fingerprinting attacks, and design more secure mobile devices with the insights provided by this work.

ACKNOWLEDGMENTS

The authors would like to thank the shepherd and the reviewers for their comments that helped tremendously in improving this work. The research reported in this paper was sponsored in part by: the DEVCOM Army Research Laboratory under Cooperative Agreement #W911NF-17-2-0196; the National Science Foundation under Awards #1705135, #2211301, and #2312089; and, the NIH mDOT Center under Award #1P41EB028242. The views and conclusions contained in this document are those of the authors, and they should not be interpreted as representing the official policies, either expressed or implied, of the funding agencies.

REFERENCES

[1] Nikolina Cveticanin. Hacking statistics to give you nightmares, 2023. https://dataprot.net/statistics/hacking-statistics/.

[2] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In *NDSS*, volume 25, pages 50–52, 2012.

[3] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 426–436. IEEE, 2015.

[4] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 691–706, 2015.

[5] Swarup Chandra, Zhiqiang Lin, Ashish Kundu, and Latifur Khan. Towards a systematic study of the covert channel attacks in smartphones. In *International Conference on Security and Privacy in Communication Networks*, pages 427–435. Springer, 2014.

[6] Nikolay Matyunin, Jakub Szefer, Sebastian Biedermann, and Stefan Katzenbeisser. Covert channels using mobile device's magnetic field sensors. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 525–532. IEEE, 2016.

[7] Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.

[8] Carlton Shepherd, Jan Kalbantner, Benjamin Semal, and Konstantinos Markantonakis. A side-channel analysis of sensor multiplexing for covert channels and application fingerprinting on mobile devices. *arXiv preprint arXiv:2110.06363*, 2021.

[9] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In *28th USENIX security symposium (USENIX security 19)*, pages 603–620, 2019.

[10] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 51–60, 2012.

[11] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal covert channels on multi-core platforms. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 865–880, 2015.

[12] Kenneth Block, Sashank Narain, and Guevara Noubir. An autonomic and permissionless android covert channel. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 184–194, 2017.

[13] Nikolay Matyunin, Yujue Wang, Tolga Arul, Kristian Kullmann, Jakub Szefer, and Stefan Katzenbeisser. Magneticspy: Exploiting magnetometer in mobile devices for website and application fingerprinting. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*, pages 135–149, 2019.

[14] Nikolay Matyunin, Nikolaos A Anagnostopoulos, Spyros Boukoros, Markus Heinrich, André Schaller, Maksim Kolinichenko, and Stefan Katzenbeisser. Tracking private browsing sessions using cpu-based covert channels. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 63–74, 2018.

[15] Peter Snyder, Soroush Karami, Arthur Edelstein, Benjamin Livshits, and Hamed Haddadi. {Pool-Party}: Exploiting browser resource pools for web tracking. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7091–7105, 2023.

[16] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J Wang, and Crispin Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *2012 IEEE Symposium on Security and Privacy*, pages 224–238. IEEE, 2012.

[17] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the eighth symposium on usable privacy and security*, pages 1–14, 2012.

[18] Android. Sensor rate limiting, 2022. https://developer.android.com/guide/topics/sensors/sensors_overview#sensors-rate-limiting.

[19] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. The web's sixth sense: A study of scripts accessing smartphone sensors. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1515–1532, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243860. URL https://doi.org/10.1145/3243734.3243860.

[20] Yicheng Zhang, Carter Slocum, Jiasi Chen, and Nael Abu-Ghazaleh. It's all in your head (set): Side-channel attacks on ar/vr systems. In *USENIX Security*, 2023.

[21] Michalis Diamantaris, Serafeim Moustakas, Lichao Sun, Sotiris Ioannidis, and Jason Polakis. This sneaky piggy went to the android ad market: Misusing mobile sensors for stealthy data exfiltration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1065–1081, 2021.

[22] Fenghao Xu, Wenrui Diao, Zhou Li, Jiongyi Chen, and Kehuan Zhang. Badbluetooth: Breaking android security mechanisms via malicious bluetooth peripherals. In *NDSS*, 2019.

[23] Matthias Gazzari, Annemarie Mattmann, Max Maass, and Matthias Hollick. My (o) armband leaks passwords: An emg and imu based keylogging side-channel attack. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 5(4):1–24, 2021.

[24] Ady Abraham. High refresh rate rendering on android, Apr 2020. https://android-developers.googleblog.com/2020/04/high-refresh-rate-rendering-on-android.html.

[25] Apple. Optimizing promotion refresh rates for iphone 13 pro and ipad pro, 2021. https://developer.apple.com/documentation/quartzcore/optimizing_promotion_refresh_rates_for_iphone_13_pro_and_ipad_pro.

[26] StatCounter. Mobile operating system market share worldwide, 2023. https://gs.statcounter.com/os-market-share/mobile/worldwide.

[27] Android Open Source Project. Graphics, 2023. https://source.android.com/docs/core/graphics.

[28] Android. Frame rate, 2023. https://developer.android.com/guide/topics/media/frame-rate.

[29] Android. Refresh rate callback, 2023. https://developer.android.com/ndk/reference/group/choreographer#achoreographer_registerrefreshratecallback.

[30] Android. Frame rate callback, 2023. https://developer.android.com/reference/android/view/Choreographer.FrameCallback.

[31] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. {DRAMA}: Exploiting {DRAM} addressing for {Cross-CPU} attacks. In *25th USENIX security symposium (USENIX security 16)*, pages 565–581, 2016.

[32] Hamed Okhravi, Stanley Bak, and Samuel T King. Design, implementation and evaluation of covert channel attacks. In *2010 IEEE International Conference on Technologies for Homeland Security (HST)*, pages 481–487. IEEE, 2010.

[33] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. Lord of the ring (s): Side channel attacks on the {CPU}{On-Chip} ring interconnect are practical. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 645–662, 2021.

[34] Ed Novak, Yutao Tang, Zijiang Hao, Qun Li, and Yifan Zhang. Physical media covert channels on smart mobile devices. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 367–378, 2015.

[35] AndroidRank. Open android market data, 2023. https://www.androidrank.org/.

[36] Maryam Mehrnezhad, Ehsan Toreini, Siamak F. Shahandashti, and Feng Hao. Touchsignatures: Identification of user touch actions and pins based on mobile sensor data via javascript. *Journal of Information Security and Applications*, 26: 23–38, 2016. ISSN 2214-2126. doi: https://doi.org/10.1016/j.jisa.2015.11.007. URL https://www.sciencedirect.com/science/article/pii/S2214212615000678.

[37] Jiexin Zhang, Alastair R Beresford, and Ian Sheret. Sensorid: Sensor calibration fingerprinting for smartphones. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 638–655. IEEE, 2019.

[38] James Robinson and Cameron McCormack. Timing control for script-based animations, 2022. https://www.w3.org/TR/animation-timing/.

[39] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In Aggelos Kiayias, editor, *Financial Cryptography and Data Security*, pages 247–267, Cham, 2017. Springer International Publishing. ISBN 978-3-319-70972-7.

[40] StatCounter. Browser market share worldwide, 2023. https://gs.statcounter.com/browser-market-share.

[41] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. Sok: In search of lost time: A review of javascript timers in browsers. In *2021 IEEE European*

Symposium on Security and Privacy (EuroS&P), pages 472–486, 2021. doi: 10.1109/EuroSP51992.2021.00039.

[42] Chao Liu, Ryen W White, and Susan Dumais. Understanding web browsing behaviors through weibull analysis of dwell time. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 379–386, 2010.

[43] Amit Kumar Sikder, Hidayet Aksu, and A Selcuk Uluagac. 6thsense: A context-aware sensor-based attack detector for smart devices. In *USENIX Security Symposium*, pages 397–414, 2017.

[44] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. "andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

[45] Prakash Shrestha, Manar Mohamed, and Nitesh Saxena. Slogger: Smashing motion-based touchstroke logging with transparent system noise. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 67–77, 2016.

[46] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. Keydrown: Eliminating software-based keystroke timing side-channel attacks. In *Network and Distributed System Security Symposium*. Internet Society, 2018.

[47] Denis Foo Kune and Yongdae Kim. Timing attacks on pin input devices. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 678–680, 2010.

[48] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W Fletcher, and David Kohlbrenner. Hertzbleed: Turning power {Side-Channel} attacks into remote timing attacks on x86. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 679–697, 2022.

[49] Gaofeng Dong, Ping Wang, Ping Chen, Ruizhe Gu, and Honggang Hu. Floating-point multiplication timing attack on deep neural network. In *2019 IEEE International Conference on Smart Internet of Things (SmartIoT)*, pages 155–161. IEEE, 2019.

[50] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pages 104–113. Springer, 1996.

[51] Dawn Xiaodong Song, David A Wagner, Xuqing Tian, et al. Timing analysis of keystrokes and timing attacks on ssh. In *USENIX Security Symposium*, volume 2001, 2001.

[52] Mengyuan Li, Yan Meng, Junyi Liu, Haojin Zhu, Xiaohui Liang, Yao Liu, and Na Ruan. When csi meets public wifi: inferring your mobile phone password via wifi signals. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1068–1079, 2016.

[53] Jingchao Sun, Xiaocong Jin, Yimin Chen, Jinxue Zhang, Yanchao Zhang, and Rui Zhang. Visible: Video-assisted keystroke inference from tablet backside motion. In *NDSS*, 2016.

[54] Wenqiang Jin, Srinivasan Murali, Huadi Zhu, and Ming Li. Periscope: A keystroke inference attack using human coupled electromagnetic emanations. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 700–714, 2021.

[55] Ximing Liu, Yingjiu Li, Robert H Deng, Bing Chang, and Shujun Li. When human cognitive modeling meets pins: User-independent inter-keystroke timing attacks. *Computers & Security*, 80:90–107, 2019.

[56] Ke Sun, Chunyu Xia, Songlin Xu, and Xinyu Zhang. StealthyIMU: Extracting permission-protected private information from smartphone voice assistant using zero-permission sensors. In *NDSS*, 2023.

[57] Yuxuan Yan, Zhenhua Li, Qi Alfred Chen, Christo Wilson, Tianyin Xu, Ennan Zhai, Yong Li, and Yunhao Liu. Understanding and detecting overlay-based android malware at market scales. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 168–179, 2019.

[58] Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. Cloak and dagger: from two permissions to complete control of the ui feedback loop. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1041–1057. IEEE, 2017.

[59] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, Austin, TX, August 2016. USENIX Association. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp.

[60] Wen Qi, Wanfu Ding, Xinyu Wang, Yonghang Jiang, Yichen Xu, Jianping Wang, and Kejie Lu. Construction and mitigation of user-behavior-based covert channels on smartphones. *IEEE Transactions on Mobile Computing*, 17(1):44–57, 2017.

[61] Liang Cai and Hao Chen. Touchlogger: Inferring keystrokes on touch screen from smartphone motion. In *6th USENIX Workshop on Hot Topics in Security (HotSec 11)*, 2011.

[62] Anupam Das, Nikita Borisov, and Matthew Caesar. Tracking mobile web users through motion sensors: Attacks and defenses. In *NDSS*, 2016.

[63] Sanorita Dey, Nirupam Roy, Wenyuan Xu, Romit Roy Choudhury, and Srihari Nelakuditi. Accelprint: Imperfections of accelerometers make smartphones trackable. In *NDSS*, volume 14, pages 23–26. Citeseer, 2014.

[64] Yuval Yarom and Katrina Falkner. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *23rd USENIX security symposium (USENIX security 14)*, pages 719–732, 2014.

[65] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[66] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[67] Catherine Easdon, Michael Schwarz, Martin Schwarzl, and Daniel Gruss. Rapid prototyping for microarchitectural attacks. In *USENIX Security Symposium*, 2022.

[68] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *NDSS*, 2020.

[69] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit. https://cs.adelaide.edu.au/~yval/Mastik/, 2016.

[70] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. {AutoLock}: Why cache attacks on {ARM} are harder than you think. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1075–1091, 2017.

[71] Xiaokuan Zhang, Xueqiang Wang, Xiaolong Bai, Yinqian Zhang, and XiaoFeng Wang. Os-level side channels without procfs: Exploring cross-app information leakage on ios. In *Proceedings of the Symposium on Network and Distributed System Security*, 2018.