

PRISM: To Fortify Widget Based User-App Data Exchanges Using Android Virtualization Framework

Ying Tat Ng
Singapore Management University
Singapore
yingtat.ng.2021@phdcs.smu.edu.sg

Haiqing Qiu
Singapore Management University
Singapore
hqqiu@smu.edu.sg

Zhe Chen
Singapore Management University
Singapore
zhechen@smu.edu.sg

Xuhua Ding
Singapore Management University
Singapore
xhding@smu.edu.sg

Abstract

We present PRISM, an UI hardening technique for an Android app to safeguard its widgets against a corrupted kernel. PRISM ensures secure interface rendering and allows for visual authentication, which developers could use to enable user intent confidentiality protection. Our design leverages the recent Android Virtualization Framework with minimal changes to the existing UI framework and graphics subsystem. It is much easier to deploy and use PRISM on Android phones than TrustZone-based secure UI schemes, because the apps are not admitted to the Secure World and retain their full rights to manage and control their own interfaces. We have implemented the prototype of PRISM and a test app on Google Pixel 7 and assessed its security, usability and performance with extensive experiments. The results successfully validate the strength of its security and show unnoticeable latency in most interface operations.

ACM Reference Format:

Ying Tat Ng, Zhe Chen, Haiqing Qiu, and Xuhua Ding. 2025. PRISM: To Fortify Widget Based User-App Data Exchanges Using Android Virtualization Framework. In . ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 Introduction

The user interface (UI) of mobile applications is the communication channel for user-app interactions. For apps geared for conducting transactions in banking, e-commerce, and healthcare, their UIs are naturally abundant in private and valuable data, and thus become an attractive target for attacks. User space attacks typically abuse the rich features of the Android UI framework for dynamic human-app interactions. They range from clickjacking [20, 35, 38], accessibility manipulation [3, 7, 9, 19] and task hijacking [25] in the past, to the recent PHYjacking [32] using a malicious activity and an attack on the privacy indicator [6]. Notorious phishing attacks [5, 29]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Asia CCS'25, Hanoi, Vietnam

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

exploring the human inability to discern two similar-looking interfaces remain as a persistent threat with limited countermeasures [2, 4, 16, 28] available. More powerful and more stealthy UI attacks could come from kernel-privileged malware. Like other commodity operating systems, Android OS is not spared from kernel vulnerability exploits [21]. A kernel adversary can directly tamper with an app's UI and steal data without misusing Android UI features.

Arm TrustZone has been widely used to counter kernel-based UI attacks in VButton [17], TruZ-Droid [36], Rushmore [22], SecurePay [15], SEPI [6] and Android Protected Confirmation [11]. Although these schemes vary in capabilities and app contexts, they share the approach of leveraging the high privilege of the Secure World (SW) to block malicious accesses and override Android control of the touchscreen.

Besides performance-related issues, the TrustZone-based approach has two inherent drawbacks. Firstly, the approach faces the dilemma of handling app-specific UI code (e.g., an event handler) and data. On the one hand, the unprotected code cannot generate trustworthy UI data or securely handle user input data. On the other hand, to port it to SW violates the least-privilege principle and undermines TrustZone's security assurance, i.e., to provide a sanctuary to highly trustworthy code only. Existing schemes balance the two sides by placing a dedicated program implementing the predefined service in SW. By and large, Rushmore [22] and SecurePay [15] propose a decrypt-then-display service while Android Protected Confirmation [11], TruZ-Droid [36] and VButton [17] implement a display-then-send service. While benefiting from the security services, app developers lose the freedom to determine the secure UI's layout, style, contents (partially), and even the computation over the user inputs. The second drawback is related to practicality. There exist roadblocks against adopting TrustZone-based techniques, as evidenced by the fact that such research prototypes were mostly implemented on development boards, instead of off-the-shelf commodity phones. Phone manufacturers impose tight control in admitting software to run in SW and are generally reluctant to make changes, considering that the code and data therein are so critical to the mobile phone platform.

In this paper, we propose the *pKVM-based Protected Interface Scheme* (or PRISM for short) for mobile apps such as a healthcare app to selectively harden their privacy- or security-sensitive widgets against a kernel-privileged adversary. It is still at the app developer's own discretion to determine what to display in the protected

interface and how to respond to user actions. PRISM builds upon the recently introduced Android Virtualization Framework (AVF) with minimal changes to the Android UI framework and the underlying graphics subsystem. Note that PRISM is *not* geared for generic UI protection, as it is ill-suited for content-rich interfaces such as video streaming, file editing, and gaming. PRISM provides the following security assurance upon a hardened UI.

- *Secure interface rendering.* A PRISM-protected interface displays its widgets (buttons, radio buttons, checkboxes, switches, toggle buttons, and textboxes) with integrity and secrecy against a kernel-privileged adversary. The integrity assurance covers both the app's display data and, more importantly, its visual effect.
- *Visual authentication.* PRISM allows a security-vigilant user to visually authenticate the interface s/he sees. For all interfaces displayed through PRISM, a PRISM icon is securely displayed to differentiate protected interfaces from unprotected ones. A user can also securely bind an alias to a PRISM app so that only when *this* app is displaying a PRISM protected interface does PRISM show the corresponding alias. Visual authentication helps vigilant users to detect phishing-like attacks under the trust-on-first-use (TOFU) assumption.

To use PRISM, an app needs to create a *secure endpoint* inside its Microdroid *protected VM* (pVM) – Google's virtualization based TEE – to hold the app's secure UI related code and data. The secure endpoint is responsible for displaying sensitive interfaces as well as processing user inputs. Essentially, PRISM protects the data flow from the secure endpoint to the user, not the other way round. However, the secure app-to-user channel lends itself to the app to share a secret with the user so that a user's input via a widget can be "encrypted" by using a secret shuffling. In this manner, an app with a PRISM-protected interface can protect the secrecy of a user's option choice, numerical inputs, and even passwords.

As compared with TrustZone based approaches, using pVM as a TEE requires more considerations to reduce the (modified) hypervisor's impact on Android execution. A deeper challenge of our approach is to preserve the Android UI framework and keep the size of the isolated code as small as possible. While the app's sensitive UI code is securely isolated, we do not decouple its execution from the original workflow Android expects.

We implemented a PRISM prototype on Google Pixel 7 with 1124 SLOC added into the pKVM, and developed a mobile app using PRISM to interact with the user. According to our experiments, a PRISM protected interface does not show any performance drop as compared with an unprotected one. An anonymized demo video is available for viewing at <https://bit.ly/4jq71af>¹. The prototype demonstrates that the deployment of PRISM is more practical than TrustZone based UI schemes, as it only requires minute modification to the kernel, instead of the hardware vendor's authorization.

To summarize, we make the following contributions.

- We propose PRISM, an Android AVF-based secure UI mechanism that allows an app to securely display a sensitive interface and a user to visually authenticate it, both against a kernel privileged adversary.

- We design two shuffling mechanisms for an app to leverage PRISM-based secure interface rendering to protect the secrecy of a user's widget input such as a selection, a PIN or a password.
- We build a prototype of PRISM and testing applications on an off-the-shelf Google Pixel phone and evaluate its security and performance.

ORGANIZATION. The next section briefly explains the background of techniques used in PRISM. Section 3 formulates the problem and presents the overview of PRISM. We explain the secure interface rendering in Section 4 and how an app protects user inputs in Section 5. Section 6 reports our prototype implementation followed by performance and security evaluations in Section 7 and 8, respectively. A comparison between PRISM and related work is presented in Section 9. Section 10 concludes the paper.

2 Background

2.1 Android Virtualization Framework

The Android Virtualization Framework (AVF) [10], recently introduced in Android 13, is built on the *protected KVM* or pKVM, a hypervisor running in EL2 and managing *protected virtual machines* (pVMs). An attractive usage of AVF is for an app to create a pVM as a trusted execution environment (TEE) and to run its security-sensitive workloads therein. For this purpose, Android provides *Microdroid* [14] as a stripped-down version of the Android OS. Such pVMs are generally known as Microdroid pVMs.

An app prepares a binary image as the payload of its own Microdroid pVM instance which is launched via Android APIs. AVF achieves launch-time integrity and trustworthiness through Android Verified Boot [12], starting with pVM firmware (pvmfw), followed by the kernel and Microdroid. In the first launch of the pVM, the pvmfw starts a new pVM instance by creating a disk image *instance.img*. At each stage of the boot chain, the sealed measurement of the following stage is written into *instance.img*. This enables verification of the pVM instance as subsequent launches will entail the same process of measurement at each stage and checked against the values stored in *instance.img*.

At each stage of the boot chain, a unique identifier, known as the compound device identifier (CDI), is derived based on Device Identifier Composition Engine (DICE) [13]. The pKVM first derives the CDI for pvmfw using its own CDI and the pvmfw image's digest, which would be used to derive the CDI for the kernel using the pvmfw's CDI and the kernel image's digest. This is repeated to derive Microdroid's CDI. This process verifies the images along the boot chain as any change in a loaded image at any stage will change both its own CDI and the CDIs of subsequent stages. Sealing keys are also derived from the CDI of each boot stage. Although Microdroid pVMs use the same kernel and Microdroid image files, they are different virtual machine instances. In other words, an app's workload exclusively runs in its pVM instance. To distinguish between pVM instances, the pvmfw includes a salt into the DICE process. AVF enforces VM-level isolation among the host Android and each of the pVMs by setting the Stage-II translation tables, which prevents other pVMs or host Android from reading a pVM's CDIs and sealing keys. Note that the host Android kernel is *not* more privileged than pVM kernels, as they all run at EL1.

¹The source code of PRISM and the app will be released after the work is accepted to publish.

Starting from Android_14_r13, AVF supports remote attestation against pVMs in a similar way to SGX enclave attestation. During manufacturing, a device's DICE root public key is registered in a trusted Remote Key Provisioning (RKP) server. The device is pre-installed with a service pVM known as the *RKP VM* which plays the role of the local anchor for regular pVM remote attestation. The RKP server remotely verifies the continuous presence the RKP VM according to its DICE chain and certifies its public key (if it is legitimate). For a pVM to attest to its launching time integrity, it submits its DICE chain for the RKP VM to verify. Upon a successful verification, the RKP VM certifies the public key generated by the attesting pVM. The pVM signs its attestation data using the corresponding private key. The remote verifier verifies the signature using the certification chain rooted at the RPK server's credential.

2.2 Google Pixel Phone Display

Functioning at the bottom of the graphics stack, the hardware display mechanism handles how images composed by Android graphics are rendered on the screen. As Pixel phones use Samsung LCD panels, it is the Samsung Exynos SoC Display and Enhancement Controller (DECON) driver running in the kernel that completes this "last mile" in graphics output, i.e., to transport image data from framebuffer to the LCD. We briefly explain the mechanism below, as it is the prerequisite to understand how PRISM secures interface rendering. Note that other manufacturers such as MediaTek and Qualcomm use similar mechanisms in their display controllers [22].

DECON comprises six Display Pre-Process (DPP) and six overlay layers called *windows*. Largely, the DECON driver pairs one DPP with one window so that the eventual image on the screen is a result from the hardware's blending of all layers in use. Figure 1 illustrates how DECON works. Each window is conceptually a rectangular region starting from the top-left corner of the screen and is equipped with a suite of registers to store rendering related parameters such as window status (enabled/disabled), dimensions, the α -blending mode (transparent or opaque), the foreground and background alpha value. Each DPP is equipped with a set of registers specifying arguments related to the graphic memory data, including the virtual address (VA) of the framebuffer, the compression type (e.g., Arm Frame Buffer Compression or AFBC²), the color format of the framebuffer data and the DMA type. The framebuffer-to-LCD data transportation requires the assistance of the System Memory Management Unit (SMMU) which translates VAs to physical addresses.

During image composition, the six overlays are stacked by the hardware in a fixed order. From the bottom to the top, the layers are termed as Win-0, Win-1, ..., and Win-5, respectively, by the driver. If a layer's blending mode is set as opaque, it blocks the visibility of the same screen regions at all lower layers. Hence, when Win-5 is set as full-screen using opaque blending, all other five layers are not shown to the user. Figure 1 illustrates an example of the composition effect where only three layers (Win-5, Win-1 and Win-0) are used. Win-5 is in opaque mode, covering the upper half of the screen, while Win-1 and Win-0 are blended with only the lower-half visible.

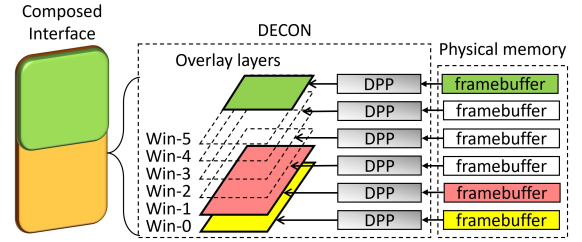


Figure 1: Overlay blending by Samsung DECON.

3 Design of PRISM

We first formulate the security problem we aim to solve and then present an overview of PRISM design.

3.1 Problem Formulation

3.1.1 Adversary Model. The adversary is malware which has compromised Android and the underlying kernel. It can run arbitrary code with kernel privilege. We do not consider malicious human users operating the phone. In addition to using the attacks mentioned in Section 1, the adversary can leverage its kernel privilege to read or modify relevant memory contents, e.g., injecting code and tampering with the data therein. It can also become the man-in-the-middle between the user and a victim app by showing a faked interface to the user and injecting data to the app. However, it is incapable of compromising pKVM or launching physical attacks [18, 31] that mimic human operations. We also consider a phishing adversary which attempts to impersonate a victim app by displaying an interface with the same appearance to the user. Since it remains an open problem to detect phishing apps during installation, we follow the Trust-On-First-Use (TOFU) model [33] as used in the literature. In other words, the genuine app is installed *prior to* a phishing app.

3.1.2 Scope of Protection. We consider security of widget interfaces only as they are the ones commonly used by apps to prompt users to provide input data (PINs and bank account numbers) under a context. Such interfaces are more attractive to the adversary than others because of the values of the data involved. It is *not* in our scope of study to provide generic interface protection such as gaming, video streaming and file viewing. A widget interface is essentially a duplex communication channel between two *endpoints*: a human user and an app's interface functions determining what to display and receiving user inputs. The channel allows for both the *user-bound* flow, i.e., the perceptible data rendered on the screen to the user, and the *app-bound* flow, i.e., the user's inputs through the touchscreen.

PRISM is designed primarily to safeguard the user-bound flow for two reasons. First, the user-bound data prompts the human user to act according to the displayed contextual information and is thus the prerequisite for secure human-app interactions. Second, human users are the weakest link in the security chain and face more attacks as evidenced by many incidents in the past. Specifically, PRISM addresses the following three security aspects:

- **Interface Visual Integrity** It means that what the user perceives on the screen is exactly what the app intends to show. The visual integrity not only concerns the binary integrity of the app's

²<https://www.arm.com/technologies/graphics-technologies/arm-frame-buffer-compression>

interface data, but also ensures no distortion of its visual effects by blocking, re-sizing, color-tinting etc.

- **Interface Data Secrecy** An interface usually displays a mixture of public data commonly displayed for all users and private data which is specific to the particular user. For instance, the string “Saving Account” is public whereas the account number and the balance is private. PRISM is used to protect secrecy of an app interface’s private data.
- **Interface Authenticity** Admittedly, PRISM cannot empower a user to visually detect a phishing interface [5, 29] in all circumstances, which is a longstanding open problem. Nonetheless, under the TOFU model, PRISM allows a vigilant user to visually authenticate apps s/he becomes “acquainted” with through labeling.

3.1.3 Incidental Benefit from Secure User-bound Flow. We observe that the app’s capability of securely displaying secret data to the user is analogous to secret key distribution. If humans were capable of encrypting data mentally, a secret key could be displayed to share with the user and to encrypt the app-bound flow without additional system support to escort the flow using a trusted entity. Therefore, we study how to embody this benefit from PRISM’s interface protection without burdening users to perform encryption.

3.2 Overview

We design PRISM as a secure interface service for Android apps to invoke and protect their interface rendering. Figure 2 illustrates our approach wherein PRISM interposes itself on the user-bound data flow and does not intervene in the opposite direction.

- **Endpoint.** Under the adversary model described above, it is necessary to safeguard secrecy and integrity of the app’s endpoint, i.e., its interface handling functions. If not, the adversary can directly tamper with the interface before its rendering. An app launches its *secure endpoint* in its own Microdroid pVM. AVF ensures that data and execution flows inside a pVM is secure against attacks from (corrupted) Android and other pVMs. The pKVM identifies the secure endpoint via static and runtime identifiers as described in Section 4.2. Moreover, the app’s server can remotely verify whether the secure endpoint being created as expected by using AVF’s remote attestation scheme.
- **User-bound Flow.** We design a pKVM-based secure display system whereby the pKVM securely controls the *topmost* layer of the screen (Win-5). With this capability, both the pKVM’s data and the interface data from a secure endpoint can be securely and dynamically rendered in Win-5 without obstruction or modification.

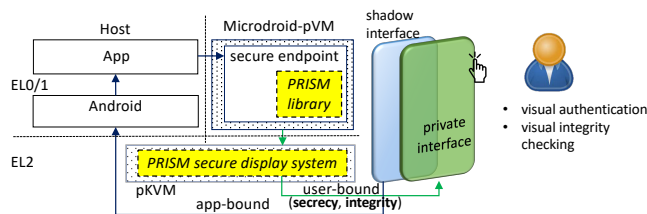


Figure 2: Overview of PRISM architecture.

The *private interface* in Figure 2 refers to the app’s security-sensitive interface created by its secure endpoint and rendered via PRISM. It is displayed to the user and is inaccessible to Android. The *shadow interface* is the public deputy of the private interface. Containing public data only, it is generated by the app using standard Android APIs and is rendered by Android, however covered by its private counterpart. The purpose of the shadow interface is to allow Android to properly handle user inputs despite having no access to the private interface.

PRISM supports visual endpoint authentication for vigilant users. The pKVM *always* controls the top-left corner of the screen to indicate whether the present interface on the screen is under PRISM protection or not. In addition, when rendering a private interface from a secure endpoint, the pKVM also displays the app’s *alias* (if any) which is previously chosen by the user at the app’s first launch.

REMARK. It is a distinctive feature and our weighed decision that PRISM is *not* involved in app-bound flow as in Figure 2. It keeps PRISM compact and lightweight, without replicating the Android UI framework and with only minute modifications. Although technically possible (as discussed in Section 9.2), protecting touchscreen inputs demands a significant modification to the existing I/O subsystem and is performance-disruptive to the entire system. A touch device driver in the Android kernel alone has 24,773 lines of code, not to mention the code in Android’s input framework. Our observation is that Android’s Protected Confirmation [11] is a ready solution for integrity checking. As we show in Section 5, the benefit from secure interface rendering can *partially* cope with the input secrecy problem.

4 Secure Interface Rendering

In this section, we first explain how, as part of system bootup, the pKVM controls the topmost layer of the phone screen (Win-5) based on the overlay mechanism in Samsung Exynos. We then elaborate how the user chooses an alias for an app and registers it to the pKVM upon the app’s first launch. Finally, we describe the layout and composition of the interface displayed to the user.

REMARK. According to Park et. al. [22], other display controllers including “Qualcomm MSM, TI’s OMAP, Nvidia’s Tegra and Mediatek” use the same or similar overlay mechanism. Our approach for secure interface rendering is also feasible for mobile phones using these chipsets.

4.1 Protection of Win-5 Rendering

As stated in Section 2, Win-5 is rendered by the hardware at the topmost layer of the screen. In PRISM, the pKVM controls the configuration and the framebuffer used to render Win-5 to securely display the contents visible to the user.

Configuration Protection. After the Android host initializes all peripheral devices during system bootup, the pKVM revokes the host’s permissions to manage Win-5 by blocking host access to any registers related to Win-5 rendering. Specifically, it isolates Win-5 control registers storing display parameters such as dimensions and opaqueness. It sets Win-5 dimensions as full screen size by default and blending mode always as opaque so that the window is displayed in the opaque mode covering the entire screen. The pKVM

also secures a DPP (e.g., DPP-4 in Figure 3) to exclusively render Win-5 by isolating and setting this DPP’s registers. The setting instructs the DPP to render Win-5 using the framebuffer located at a designated address. Since this dedicated framebuffer is used in PRISM until the phone reboots, we name it the *PRISM framebuffer* for ease of presentation. The isolation prevents the kernel adversary from maliciously altering the chosen DPP to render another overlay window or to use a different framebuffer address. The DPP isolation in PRISM does not disrupt overlay rendering and composition as the display driver never changes an existing pairing between a DPP and an overlay window.

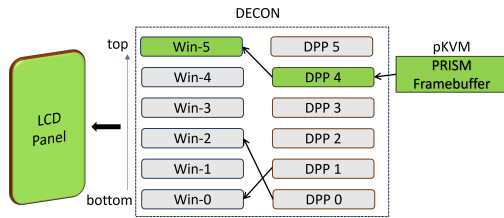


Figure 3: Secure Win-5 rendering.

Framebuffer Protection. Although DPP register isolation ensures that the VA of the PRISM framebuffer is not modifiable, additional measures are still needed to protect address translation and buffer accesses. The pKVM protects the buffer against illegal CPU accesses and DMA accesses. To block the host Android’s CPU access, it removes the mappings to all physical pages of the PRISM framebuffer from the Stage-II translation tables servicing the host Android. To block illicit DMA accesses through the SMMU, the pKVM takes over the control to the SMMU registers during device bootup. At runtime, it intercepts accesses to the Stage-II page tables used by the SMMU to prevent the PRISM framebuffer used for other devices’ DMA except the display.

PRISM’s protection and exclusive control of Win-5 rendering does not interfere with Android’s graphics framework in principle, because the latter has the built-in adaptability to adjust its image composition process according to the number of available overlay layers rendered by the hardware.

4.2 Secure Endpoint and Alias

To use PRISM, an app launches a Microdroid pVM with its secure endpoint as the payload. The endpoint consists of the app’s sensitive interface management code and the PRISM library which includes functions for the secure endpoint to engage with PRISM. Note that all the app’s functions that accesses sensitive UI data need to be enclosed inside its secure endpoint. We highlight that each endpoint resides in *its own pVM* whose memory pages are isolated from the host Android and other pVMs.

Microdroid pVM Attestation. When the app launches the secure endpoint, it initiates a remote attestation with its server. Built upon a successful verification of the attestation data, the app server and the secure endpoint use a TLS-like protocol to establish a shared key *k*. The shared key allows them to authenticate the exchanged data using the standard HMAC algorithm.

Endpoint Runtime Identification. Since PRISM’s service is open to all apps, there would be multiple secure endpoints. For the pKVM to identify a secure endpoint through its pVM at runtime, we modify Microdroid to provide a secure endpoint’s identifiers. The intuition is to treat the sealing key derived by Microdroid with its CDI as its *static identifier* since it lasts until the image is deleted. The VTTBR0_EL2 value for the pVM’s Stage-II mapping hierarchy, which is accessible to the pKVM, is treated as its *runtime identifier*. The pKVM uses Table 1 to manage and identify pVMs holding secure endpoints. Specifically, when launching a pVM, Microdroid passes its sealing key to the pKVM. In response, the latter updates the corresponding entry in Table 1 with the VTTBR0_EL2 value currently serving this pVM. When the pKVM receives a hypercall, it locates the current VTTBR0_EL2 and looks up Table 1 to identify the pVM instance holding the secure endpoint.

Alias Registration. Built on top of endpoint identification, we introduce an alias mechanism to mitigate phishing-like attacks on the basis of TOFU. An app’s alias, a string of letters chosen by the user, serves as the app’s unique “birthmark” for the user to visually recognize its private interface. When its secure endpoint displays its private interface for the very first time, an app obtains its alias from the user through a text input widget, such as a randomized keyboard, and registers it to PRISM through a hypercall. In response, the pKVM first checks whether the alias had been used in Table 1, to ensure each alias is unique. If not, it locates the entry in Table 1 by using the present VTTBR0_EL2, and then inserts the alias there. Once a secure endpoint is assigned with an alias, the pKVM does not allow the pairing to be changed.

Table 1: The pKVM’s Endpoint Table.

Sealing Key	VTTBR0	Alias
0x12345	0x1234568	BankX
0xabcdef	0x1234567	BankY
...	...	

Note that while the assigned alias should not be modified, secrecy protection is unnecessary. PRISM ensures that only the pKVM can display an app alias.

4.3 Win-5 Layout

The pKVM’s control over the contents and the layout of Win-5 starts *immediately* after phone bootup and lasts until a power-off or reset. The dimensions and contents of Win-5 depend on whether an app’s private interface is in rendering. Hence, a user may see two types of screen layout, as illustrated in Figure 4, depending on whether a private interface is active.

No Private Interface in Rendering. In this scenario, Win-5 only consists of the *state region* at the top-left corner with a fixed size as in Figure 4(a). A red open lock is displayed to declare that there is no private interface in rendering. Except the state region, the rest of the screen is rendered by Android and/or apps.

Private Interface in Rendering. In this scenario, Win-5 is divided into the state region in the top left corner, the *alias region* besides the former, and the *widget region* bordering the previous two as in Figure 4(b). The state region displays a closed green lock indicating that a private interface is in rendering. The alias region displays the

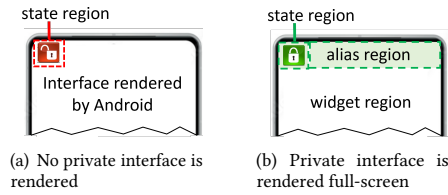


Figure 4: PRISM Screen Layouts. The state region for visual state authentication is always under the pKVM’s control.

app’s alias (if any). Since the contents of these two regions are solely decided by the pKVM, we call them collectively as the *hypervisor panel*. The app’s secure endpoint determines the contents and the height of the widget panel. The widget panel always occupies the full screen width while the height varies based on the app’s need, which gives the app the flexibility to display a public interface under its private one.

Note that the state region is controlled by the pKVM at all times, analogous to the LED light installed in some devices indicating activities inside SW. This is crucial for visual authentication as the information in the state region is always trustworthy. It prevents the adversary from staging a faked hypervisor panel and deceiving the user into trusting an unprotected interface. The user can always trust the lock’s state and safely determine whether the interface is under PRISM protection and the interface’s identity if an alias is shown.

4.4 Win-5 Composition For Private Interface

A secure endpoint’s requests to display a private interface is forwarded to the pKVM through a hypercall and is only granted when there is no other private interface rendered. This ensures that Win-5 is exclusively used by one app at a time. We explain below how the data in the PRISM framebuffer is securely provisioned by the pKVM for the hypervisor panel and the secure endpoint granted to use the widget region.

PRISM Framebuffer Split. To prevent any secure endpoint from reading or writing into the hypervisor panel, the PRISM framebuffer is split into two parts and page table based access control is imposed to prevent trespassing. Figure 5 illustrates how the pKVM and the secure endpoint access the PRISM framebuffer in different ways. While the pKVM accesses the hypervisor panel pages using its own Stage-I translation, the endpoint’s access undergoes Stage-II translation. Since data in the framebuffer describes screen pixels row by row, we choose the height of the hypervisor panel so that its framebuffer data is exactly page aligned. Among all physical pages in the PRISM framebuffer, those corresponding to the hypervisor panel are never mapped in a secure endpoint’s Stage-II translation tables.

Hypervisor Panel. For the state region, the pKVM copies the open green lock image to the designated pages in the PRISM framebuffer. For the alias region, the pKVM reads the VTTBR0_EL2 for the calling pVM, and then uses it to retrieve the corresponding alias from Table 1. If successful, it writes the alias into the alias region of the panel. Otherwise, the alias region displays the predefined background (i.e., a green rectangle) without any text. It is infeasible

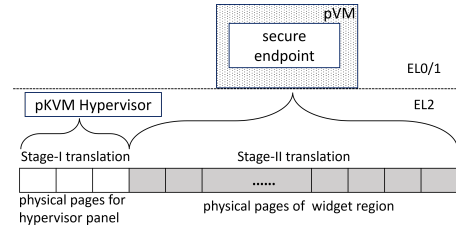


Figure 5: Split in the PRISM framebuffer where physical pages for the hypervisor panel are NOT mapped to any pVM.

for a malicious secure endpoint to make another endpoint’s alias to be displayed on its private interface. The user is thus given a firm ground to visually authenticate the app. A detailed analysis is in Section 8.

Widget Region. An app’s secure endpoint controls all bytes within the widget region, including the widget data and user-specific data. The former includes button images, text labels, frame dimensions etc. As the same data is used among all users of the app, it is essentially *public*. The user-specific data is about the user’s own status, such as the account balance, identity and subscribed services, and is thus kept *private* to the adversary. This data can be downloaded by the secure endpoint from its server or created and stored locally inside the pVM. Upon launch, the pVM has no mappings to the PRISM framebuffer. The secure endpoint chooses a range of Intermediate Physical Addresses (IPAs) and issues a display request to the pKVM. To prevent a rogue app from abusing Win-5 usage, PRISM grants the request on the condition that the app’s activity is the *topActivity* which, in Android’s definition, refers to the activity presently shown to the user. In other words, PRISM does not override Android’s decision upon which app’s activity to be displayed. Note that retaining Android’s management role does not put the private interface at any risk of integrity/secretcy breach, as it pertains to availability only. After the pKVM builds the mapping from the indicated IPA range to the physical pages for the widget region, the secure endpoint composes its private interface and writes the graphic data to the relevant pages. Note that the pKVM does not need to read/write the widget region, except zeroing all bytes there when an exception occurs during secure endpoint execution.

5 Selected App-Bound Data Protection

PRISM’s secure interface rendering allows the secure endpoint to visually share secrets with the user. In this section, we show how an app protects the secrecy of certain types of user inputs against a kernel adversary, even though the human user cannot compute any ciphers. An appealing feature of the algorithms is that they do not require any additional support from PRISM or modification of Android UI framework. We also show how PRISM supports attested app-server data exchange.

5.1 The Rationale

Consider the two toggle buttons in Figure 6 with identical appearance and screen locations, except their states. A click on the first would imply “no” while a click on the latter implies “yes”. In both scenarios, the adversary sees clicks at the same screen position and

knows the button's text label. However, without knowing the button state the user sees, the adversary cannot determine the user's answer. (The same also applies to no-click in both scenarios.) Note that the adversary always has a 50% chance³ to guess the intent correctly, even when the entire input I/O channel is isolated. Thus, when the button state is a secret to the adversary, the raw I/O data exposes *zero* bit information of the user intent, which is the strongest possible confidentiality protection in information theory.

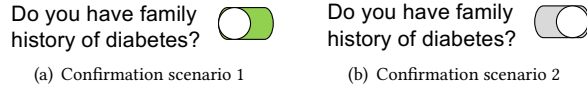


Figure 6: Two identical interfaces except the button states.

In general, there is on-screen data dictating how the user expresses her intents in the form of UI actions. One type of such data, e.g., a toggle button's state, where the user decides to act or not, is a 1-bit intent. Another type of such data where the user decides which location out of many to act upon, expresses a multi-bit intent. The on-screen keyboard layout is such an example. We coin the term *action-defining data* to refer to such data as it is different from the descriptive data (e.g., text labels) that prompts the user to generate intents, (e.g., a “yes” answer).

Clearly, to derive an intent from raw I/O data requires using action-defining data in order to make a proper interpretation. Our rationale is that, in *each* private interface rendering, an app turns its action-defining data into a *fresh and random secret* protected by its secure endpoint and PRISM's secure rendering, the adversary is unable to derive user's intents. On the other hand, the action-defining data is still visible to the user and serves its original purpose.

Next, we consider two types of action-defining data and show how our rationale is realized with simple algorithms to safeguard secrecy of inputs from popularly used widgets and credentials such as passwords and PINs.

5.2 State-Based Action Defining

Android's *checkboxes* and *toggle buttons* are popular widgets to receive a user's one-bit intent. The button's default state displayed to the user is the action-defining data. A user decides to act by clicking the button only when her intent is different from the displayed state. For such widgets, an app's secure endpoint sets a button's default state by selecting a random bit, as depicted in Figure 6. A user therefore has a 50% chance to click or not to click.

5.3 Position-Based Action Defining

Android's radio buttons, action buttons, and on-screen keyboard are widgets for a user to express a multi-bit intent by selecting one out of n candidate intents displayed at different screen positions. For these widgets, the on-screen arrangement of n candidates is the action-defining data that tells the user the click location corresponding to her intent.

For a widget of n radio buttons, the secure endpoint permutes all n labels using a freshly chosen random number, so that the position

³The probability is higher if the adversary has *a priori* knowledge of the user. Our security assertion remains valid.

of a radio button is randomly paired with a label in each interface displayed. For an on-screen keyboard, a user conveys her message intent through *multiple* clicks. It is imperative to prevent the sequence of clicks from leaking patterns of the message. Although the secure endpoint can re-shuffle the entire keyboard after each click, the cost is too high and is extremely unfriendly to users.

Suppose that the on-screen keyboard shows $K[0], \dots, K[n-1]$, a random permutation of n distinct characters and $K[i]$ is the character at position i , for $0 \leq i \leq n-1$. After the secure endpoint displays $K[0], \dots, n-1]$ on the screen, it runs the re-shuffle algorithm (Algorithm 1) whenever receiving a click and refreshes the interface. Figure 7 shows the effect.

Algorithm 1 Keyboard Re-Shuffling Algorithm (with $K[i]$ being clicked)

```

1:  $j \xleftarrow{R} [0, \dots, n-1]$  ▷ find a random index
2: swap  $K[i]$  with  $K[j]$  ▷ It is possible that  $i = j$ 

```



(a) Private keyboard composed by an app's secure endpoint (b) Refreshed keyboard with letter swapping after 'a' is clicked.

Figure 7: Illustration of a private on-screen keyboard with 38 letters and letter swapping after a click.

Intuitively, the secure endpoint resets the adversary's view of K so as to neutralize the advantage just gained from observing one click. Note that its strength and simplicity is established upon the premise that the adversary is blocked from accessing $K[0, \dots, n-1]$. The security of Algorithm 1 is proved in Section 8.

REMARK 1. The problem of on-screen keyboard is in the same vein as the access pattern privacy problem addressed by the Oblivious RAM (ORAM) [27] and Private Information Retrieval (PIR) [34]. However, Algorithm 1 cannot be applied for PIR or ORAM, because their adversary observes both the data items and indexes of retrieval while our adversary sees the indexes only, owing to PRISM's security assurance.

REMARK 2. We do not advocate the use of the random keyboard for text message input as the random layout hinders users from quickly finding each key. Nonetheless, it is feasible for short and secret messages such as PINs and passwords.

5.4 Data Processing and Submission

After deriving the user's sensitive inputs, the app's data processing functions inside the secure endpoint make necessary processing, e.g., to compose the inputted password. If needed, those functions can also submit processed data to its server. Recall that the secure endpoint and its server shares key k following the remote attestation. Hence, the app can encrypt and authenticate its data sent to

the server. Conversely, the server can securely transport data back to the endpoint.

6 Implementation

We implement a prototype of PRISM on Google Pixel 7 with 8GB RAM, ARM64 8-core processor (2x2.85 GHz Cortex-X1 & 2x2.35 GHz Cortex-A78 & 4x1.80 GHz Cortex-A55) and a 2400×1080 pixel touchscreen. It runs Android 13.0.0_r11 with Linux kernel 5.10.107. The pVM kernel is developed based on the GKI of Linux kernel 5.10.107. To test the PRISM prototype, we also implement a mobile app, App_X.

6.1 The pKVM and Win-5

In total, 30 lines of assembly code and 1094 lines of C code are added to the pKVM to support the functionality required by PRISM, primarily for memory isolation and exception handling. By setting up the Stage-II translation tables for the host, the pKVM isolates a physical page frame between 0x1C255000 to 0x1C256000 for Win-5 control registers, 6 frames between 0x1C0D0000 to 0x1C0D6000 for DPP control registers, and 6 frames between 0x1C0B0000 to 0x1C0B60000 for DPP DMA control registers. Similarly, the pKVM isolates the PRISM framebuffer which occupies 2590 pages in total. Among them, 70 pages are for the hypervisor panel and 2520 pages are for the widget region. The memory attribute for all these pages is set as normal memory with write-through. The widget region is 2336×1080 pixels with around 10.1 MB data while the hypervisor panel is 64×1080 pixels comprising one 64×64 pixel square for the state region and one 64×1016 pixel rectangle for the alias region.

The added exception handler handles EL2 exceptions triggered by the host's access to the isolated Win-5 and DPP registers, as well as the PRISM framebuffer. Read accesses to the registers are emulated by the handler, while writing accesses are aborted. The modified pKVM also includes PRISM related HVC handlers. One is to obtain the PRISM framebuffer address from the Samsung DECON driver and the other responds to the secure endpoint's request for PRISM operations such as alias registration and private interface rendering.

6.2 Host Android OS

The Android OS (including its kernel) is slightly modified. We added 30 lines of code to the Samsung DECON driver to pass the PRISM framebuffer address to the pKVM during trusted device bootup. With the developer option on, Win-5 is treated as the debug layer by Android for the developer tool's exclusive usage, such as displaying a touch cursor. PRISM's implementation is at the expense of the developer tool. We remove its drawing functions: `pointer_trace`, `draw_labels` and `dispatchPionterGestures` since their executions conflict with PRISM's protection on Win-5 rendering.

6.3 Microdroid pVM

We add three types of system calls with 90 lines of C code and 44 lines of assembly code to the Microdroid pVM kernel for PRISM operations. The first type is issued from the modified Microdroid manager during pVM launch to register the secure endpoint identifiers to the kernel. Once the secure endpoint is registered, it is immutable throughout the lifecycle of the pVM. The second type is issued from the secure endpoint to request for mappings to the

widget region of the PRISM framebuffer. The Microdroid kernel issues an HVC to the pKVM and the kernel returns the widget region's VA to the secure endpoint after building the corresponding mappings on the Stage-I translation table. The third type is issued from the secure endpoint to relinquish the app's access to the PRISM framebuffer. The kernel then issues an HVC to the pKVM to unmap the widget region of the framebuffer.

6.4 Mobile App

We implement a testing app (denoted as App_X) to demonstrate how an Android app uses PRISM to harden its interaction with a user through private widget interfaces. App_X consists of two parts: one running in the host handling most of the application functionality and a C program (i.e., its secure endpoint) as its Microdroid payload.

6.4.1 Host Portion. App_X's host portion implements an event handler for each widget appearing in its shadow interface. The handler passes the corresponding widget identifier to the secure endpoint. For text input, App_X does not invoke Android's native keyboard app. Instead, a separate fragment is used to capture the touch coordinates and pass them to the secure endpoint as the private keyboard input.

6.4.2 Secure Endpoint. The secure endpoint is implemented with 1112 lines of C code. It composes a full-screen input interface and a half-screen profile interface (Figure 10(b)), and leverages the PRISM library to protect secrecy of user inputs. At runtime, its Microdroid pVM consumes 49,100 pages excluding the PRISM framebuffer.

Sensitive Interface Rendering. The profile interface image is about 5.1 MB and contains sensitive information that could not be read by Android. The profile interface demonstrates that PRISM allows the app to have a hybrid interface as in Figure 10(b) with the public portion below the private interface. Note that it is the developer's responsibility to ensure that the untrusted data on the screen does not undermine the app's security.

The input interface comprises four widget groups: three text boxes, three radio buttons, two checkboxes and two buttons. As articulated in Section 5, radio buttons uses position-based action defining data which requires shuffling of the positions of each radio button and checkboxes uses state-based action defining data which randomizes the initial state. The resulting 1.67 MB image data and the background data (about 10 MB) are combined into the frame buffer. All shuffles used are stored in the event-to-intent mapping so that the endpoint can search for the intent according to the triggered event.

To respond to a user action, the endpoint updates the widget's appearance, e.g., a checked checkbox. In addition, it also highlights this update by changing the associated text to red and adding a red star symbol beside the widget (as shown in Figure 8). Each update highlight is retained for three seconds so that a vigilant user can detect altered inputs and faked events piggybacking with real ones. To facilitate interface drawing, we provide the raw images for all widget states as building blocks. The endpoint only needs to write the proper images in AFBC format to the corresponding locations in the PRISM framebuffer. About 200 KB are written to the framebuffer for an update.

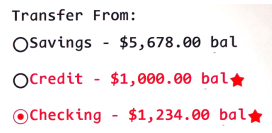


Figure 8: A photo showing widget change highlights.

Keyboard and Number Pad. The private keyboard and number pad both uses the bottom half of the private interface and 3.6 MB are written to the PRISM framebuffer to initialize the area. As shown in Figure 7, App_X has a private keyboard consisting of 38 keys: 10 digits, 26 letters, a space character and a backspace. In order to improve usability, keys for digits are colored blue, keys for vowels are colored orange and the other remaining keys are colored beige. This allows users to find the desired key quicker as it narrows the search based on color. The keyboard is displayed on the screen as a 725×1080 pixel image where each letter is a 96×96 pixel image and the 38 characters are organized in 5 rows of 8. The private number pad consists of 11 keys: 10 digits and a backspace, arranged in 4 rows of 3. Keys for odd digits are colored blue while keys for even digits are colored white. Each digit is a 128×256 pixel image. Both implement Algorithm 1 to obliterate traces of user touches. When a user taps on a key, the coordinates of `otionEvent` are passed by Android to App_X's host portion which forwards them to the secure endpoint. A key swap following a user tap results in 81 KB written to the framebuffer for the keyboard and 268 KB written for the number pad.

6.4.3 Development Effort. There are three steps to retrofit an existing app or develop a new app with PRISM. Firstly, developers follow Android APIs to incorporate the Microdroid pVM into the app and move all sensitive data and code into the secure endpoint within the pVM. Developers have to ensure sensitive data would not be leaked out of the pVM during app-endpoint interactions. Secondly, developers are expected to create private interfaces, ensuring the widget positions matches the corresponding widgets on the shadow interface. Developers could use third-party libraries to ensure the widget positions would be adjusted for varying screen sizes and resolutions. Lastly, input event handling code is to be implemented on the host portion of the app as well as in the secure endpoint. As Android cannot track states of widgets in the private interface, developers need to develop their own functions in the secure endpoint to maintain widget states and properly (re)draw the private interface images in the framebuffer. Note that the complexities of pVM identification are handled solely by PRISM and the pKVM, where no additional effort is required from the developer. While the PRISM prototype provides low-level system and security utilities such as framebuffer management and crypto functions for shuffling and randomization, it does not support image rendering. We remark that a developer can use third-party libraries to generate images of widgets and text.

7 Performance Evaluation

We evaluate the performance of PRISM by a range of experiments on App_X. We use two methods to measure the time spent by operations. For CPU operations, such as composing the private interface image, we directly instrument our code to obtain the incurred CPU time. In

contrast, touchscreen operations cannot be precisely measured by using software. For instance, the time interval between the moment when a user taps on the screen to the moment when the interface is updated on the screen cannot be measured by our code. Hence, we take a video recording of the screen updates and count the number of relevant frames. Since our recorder produces videos at 1000 fps, an UI operation shown in n frames is estimated to last around n ms.

7.1 PRISM Primitives

We instrument the code of App_X and measure the time taken for pVM launch, i.e., from the pVM initialization function call to the time when the *Payload Ready* event callback is returned. We compare it with the Google Microdroid Demo app [1]. While App_X's pVM payload (24.9 MB) is much larger than the Demo app (4.9 MB), their launch time, 3.48 seconds v.s. 3.33 seconds, only has about 1.5% difference. To further confirm that the pVM launch time is not dominated by the payload size, we customize Google's Demo app to include the same payload as App_X's secure endpoint. The modified demo app also takes about similar time (3.55 seconds) to launch. We argue that the performance penalty could not be avoided if pVM security is desired. To avoid degradation of the user experience, our prototype of App_X loads its pVM in the background during app launch.

pKVM CPU Operations. It takes about 10 μ s for the CPU exception level to switch from EL0 to EL2 through the combination of an SVC and an HVC. To isolate the PRISM framebuffer and DECON registers, the pKVM spends 207 μ s in modifying 2603 Stage-II translation table entries. It takes 21 μ s to prepare the Stage-II translation table to assign the PRISM framebuffer to a pVM. Switching the private interface on and off costs 3 μ s including modifying Win-5's dimensions by updating the control register and drawing the lock-/unlock icon on the state region. Retrieval and drawing the alias cost about 1.31 μ s. Android's read access on an isolated DECON register costs about 1.3 μ s more than reading an unprotected register.

Secure Endpoint CPU Operations. Software executions within the Microdroid pVM are mainly for computation and updating the PRISM framebuffer. We measure time spent for the initial preparation of the widget interface, keyboard and number pad, as well as updates following user operations. The time costs are reported in Table 2 showing the computation overhead is much smaller than memory operations on the framebuffer.

Table 2: CPU time for secure endpoint operations.

Operations	Computation	Framebuffer update
interface initialization	13.4 μ s	21.4 ms
widget update	4 μ s	1.3 ms
keyboard initialization	8 μ s	2.6 ms
keyboard update	7 μ s	46 μ s
number pad initialization	8 μ s	2.5 ms
number pad update	7 μ s	142 μ s

7.2 Interface Latency

To understand how the user’s app experience is affected by PRISM, we run several UI performance experiments against App_X and measure the latency. We develop a baseline of App_X using the default Android interface APIs. The baseline displays the same UI interface with *all* PRISM related functions stripped off. Our performance comparison also include Android’s Settings app using the same types of widgets. We measure three UI operations: private interface rendering; widget operations on the private UI; and private text inputs. Table 3 summarizes the performance differences.

Table 3: UI latency (in ms) of three apps.

UI Events	Settings	Baseline App	App _X
Interface Rendering	258	162	150
Keyboard Launch	210	217	122
Widget Operation	129	131	136
Text Input	105	120	71
Number Pad Input	102	118	57

The latency of interface rendering refers to the time interval from the moment when the user clicks the activation button to the moment when the new interface is fully displayed on the screen. Since the private interface rendering in PRISM deals with the raw image, the time cost slightly varies with the complexity of the interface and the image size. The keyboard launch latency refers to the time interval between the moment when the user taps and the moment when the keyboard is fully shown. The baseline app launches the system’s default Keyboard app whereas App_X renders a new interface for keyboard launch.

For widget operations, the response time is between the moment when the user clicking the widget and the moment when the updated interface is fully displayed on the screen. The latency of text input refers to the interval from the moment when the user taps a letter on the keyboard to the moment when the letter is displayed on the textbox. App_X’s latency is much smaller than the baseline and the System app, because the latter two use an additional app, i.e., the Keyboard app.

In short, the performance of PRISM protected widget UI is on par with a normal Android app’s. The key reason is that the hardware I/O cost dominates the UI latency. Note that we do not claim PRISM as a UI performance booster despite its smaller latency which is likely to be attributed to the secure endpoint’s direct operations on the framebuffer as compared with a longer execution path within Android graphics subsystem. According to Deber et al. [8], a human user can notice the difference in latency between two touch-based systems only if it is above 69 ms. Thus, the difference in latency between the baseline app and App_X is not perceptible, which can be corroborated by our video recording. Hence, PRISM does not incur perceptible latency to users, supposing that the pVM launch is in the background.

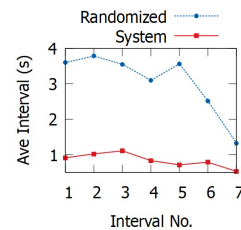
7.3 User Experience Study

To understand how user experience is impacted by keyboard and number pad shuffling, we run a user study with 20 volunteers. IRB

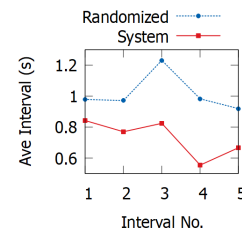
approval was obtained from our institution⁴. The study consists of two contrasting experiments using App_X followed by a user acceptance survey. Serving as the baseline, the first experiment measures the total time that a user takes to enter a string of six letters followed by two numbers (simulating password input) and to enter a personal identification number (PIN) of six digits, both via the system keyboard. The second experiment is the same as the first except that the interface is protected using PRISM’s randomized keyboard and PRISM’s randomized number pad. Finally, a user acceptance survey consisting of two identical sets of four questions was administered to solicit users’ views on the randomized keyboard and randomized number pad. The survey uses a five-point scale where a score of 1 refers to “Strongly Disagree” while a score of 5 refers to “Strongly Agree”. Results of the user acceptance survey could be found in the Appendix. Table 4 reports the total average time for a user to enter the given 8-character string and the given 6-digit PIN. The randomized keyboard took a high toll on user experience while the randomized number pad had a much lower impact. Figure 9(a) shows the average interval between keystrokes for the randomized keyboard and the system keyboard when entering the string. The time interval for the last two keystrokes are significantly lower as the last two characters of the strings are numbers and the keys have a distinct colour. On user acceptance, the randomized keyboard has an average score of 3.26 while the randomized number pad has an average score of 3.83.

Table 4: Average time for text input.

Tests		Average Time
8-character strings	Baseline	5.92s
	Randomized	21.41s
	Cost	15.49s
6-digit PIN	Baseline	3.66s
	Randomized	5.08s
	Cost	1.42s



(a) Randomized Keyboard vs System keyboard



(b) Randomized Number pad vs System keyboard

Figure 9: PRISM Summary of study results for usability.

⁴No personally identifiable information (PII) were collected to protect the identities of volunteers.

8 Security Evaluation

8.1 Secure Interface Rendering

Visual Effect Integrity. Visual effect integrity is attained by the fusion of two security measures. First, with PRISM's isolation upon the secure endpoint memory and the PRISM framebuffer, the adversary cannot modify an app's interface data. Second, the protection of the registers for Win-5 and DPP in use ensures that the adversary cannot tamper with the rendering of the protected PRISM framebuffer in Win-5 which is opaque and full-sized, and always rendered as the topmost layer by the touchscreen. Hence, the adversary can neither distort Win-5's visual effect by manipulating other overlay layers nor modify its interface data.

We validate the assurance with two experiments. Firstly, we test App_X against Twilight⁵ which uses Android's overlay APIs to attach a semitransparent colored filter to the app's interface. Twilight's overlay is submitted and composed by Android, together with the focused app. The screenshot (as in Figure 10(a)) made by Android indicates that App_X's shadow interface is indeed covered, while the photo (as in Figure 10(b)) shows that the phone screen displays App_X's private interface without being affected. We also test the attack used in PHYjacking [32] where a malicious covering activity is called and Android composes it in front of the victim app. As in the case of the Twilight attack, malicious activity covers App_X's shadow interface without affecting the private interface.

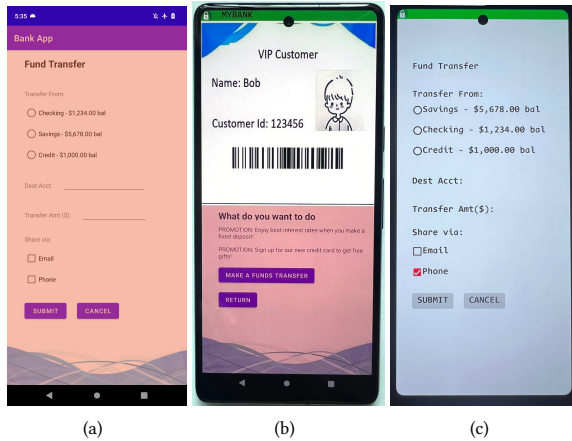


Figure 10: Two attack experiments. (a) & (b) Only App_X's shadow interface is covered by Twilight and PRISM half-screen profile interface is not covered (Android screen capture for (a) and photo for (b)); (c) App_X's phishing interface identified via the alias (photo).

Private Interface Data Secrecy. The isolation of interface data in the pVM memory and the framebuffer also protects data secrecy. We validate the assurance with two experiments. Firstly, we insert an attack function to the host kernel which attempts to read the private interface data in the Microdroid pVM and the PRISM framebuffer. Both accesses result in an EL2 exception. Secondly, we use Android's

⁵ <https://play.google.com/store/apps/details?id=com.urbandroid.lux>

built-in screen capture and XRecorder⁶ to grab the screen data. As expected, these apps only capture App_X's shadow interface.

8.2 Visual Authentication

Visual authentication is for the user to manually check the origin of the displayed data. A phishing app and/or the corrupted host kernel may impersonate a victim app by displaying its own interface with a similar appearance to the victim's. We analyze how PRISM counters such an interface forgery attack from two angles: whether the victim interface is private or public and whether the adversary uses PRISM in his attack. See Table 5 for the summary of the analysis.

A vigilant user can easily differentiate a fake public interface (even made by a malicious kernel) from the target's private one, because a public interface displays neither the closed lock icon nor an alias in the hypervisor panel at the top of the screen. Using a public interface to impersonate another public one is beyond PRISM's security assurance since PRISM is not involved in attack or defense.

Table 5: PRISM's defense against phishing attacks.

Phishing Interface	Target Interface	
	private	public
public	detectable	-
private	detectable	detectable

The attack using a private interface to impersonate the victim's private interface is essentially an attack on endpoint authenticity. To examine PRISM's security against such attacks, we cloned App_X into another app App_{X'}, so that App_{X'} has the same interface as App_X, mimicking a repackaging attack targeting App_X. Figure 10(c) shows that the phishing interface indeed appears the same as App_X. However, it does not have the same alias as App_X since PRISM treats it as a different secure endpoint as the generated identifiers would be different from the target secure endpoint. If App_{X'} attempts to perform an overlay attack by using a compromised Android host to set the topActivity to itself, the resulting overlay would show the alias of App_{X'}, instead of App_X, and be caught by a vigilant user. We stress that the authenticity is only obtained at the assumption of Trust-on-First-Use.

8.3 Indirect User Input Integrity Protection

It is PRISM's limitation that it does not empower an app's secure endpoint to validate authenticity or integrity of user input data. Namely, it can neither tell if there is indeed a user action generating the data nor determine if the received data is modified along the path. We examine below the kernel adversary's attacks on the input path in two scenarios and show the consequences.

In the first scenario, the adversary forcibly injects data to the app's code outside of its secure endpoint or directly communicates with the app server. Nonetheless, such data will *not* be accepted with trust by the app's server because its submission is not through the secure endpoint-server channel. In the second scenario, the adversary injects or alters data passing through the I/O flow to the secure endpoint. If successful, the data is eventually processed and sent by the secure endpoint and would be consumed by the server

⁶ <https://play.google.com/store/apps/details?id=videorecorder.videorecorder.screenrecorder>

with trust. For the second scenario, PRISM provides indirect input integrity protection. First, the illicit modification is reflected in the interface and can be caught by a vigilant user. The secure endpoint only receives user inputs when rendering its private interface. The kernel adversary cannot feed data to it at arbitrary time. Hence, a modified input or an injected input leads to unexpected UI changes. Since the user-bound flow is fully protected, a prudent user can detect highlighted interface updates not matching her intention. Second, the secure endpoint only receives screen coordinates as input data. Without knowing the layout of the private interface, the adversary cannot decisively confirm the consequence of its fake input. Similar to ciphertext modification, it is harder for the adversary to achieve goals with rich semantics, such as to set the payee account to a particular number, though it is easier to fake a widget operation.

8.4 Secrecy of User Intent

Section 5.2 and 5.3 propose that the app can take advantage of PRISM's interface protection to protect secrecy of certain types of user intents. Admittedly, the attained secrecy is strictly weaker than a fully-isolated app-bound channel provides, as the latter blocks the adversary from learning any raw I/O data. Nevertheless, we show that from the information theory perspective, a user intent protected by the proposed methods remains confidential. The exposed raw I/O data can at most be used as side-channel information to deduce user intent. According to information theory, the secrecy of user intent is defined as the difference δ between the adversary's success probability of guessing a user intent in two settings: (1) the a-priori probability i.e., without knowing the screen coordinates; (2) with the screen coordinates in our scheme. Our three theorems in Appendix A prove that δ is zero.

9 Related Work and Discussions

9.1 Related Work in Secure UI

ClickShield [23], WindowGuard [24], and Whattheapp [4] cope with user space adversaries launching interface tampering attacks such as clickjacking and are strictly weaker than PRISM. TrustZone based schemes introduce a Trusted Application for their respective workloads. Both VButton [17] and Android Protected Confirmation [11] tackle interface integrity only. Both obtain a user's confirmation upon the securely displayed data. A TA securely displays the received confirmation and the click event is delivered to the TA before being sent to the server with a signature. TruZ-Droid [36] displays a textbox for the user to submit passwords to the app's server.

Other schemes are concerned with interface secrecy, instead of integrity. SchrodinText [26] proposes a TrustZone based agent that receives an encrypted text string from an app's server and displays it on the interface from TrustZone by using glyphs prepared by Android in the framebuffer without compromising secrecy of the text. It neither addresses other UI security issues nor takes user inputs. SecurePay [15] proposes the same secure text displaying mechanism with the focus on two-factor authentication wherein the app server sends the encrypted OTP and a transaction summary to the TA. It also implements a software-based indicator whose security depends on a secret shared between the user and the device. The Rushmore TA [22] receives an encrypted image from an app

server and displays the plaintext image using a framebuffer not accessible to Android.

Table 6 shows a detailed comparison between PRISM and those schemes. Only PRISM provides visual authentication against apps while others authenticate the interface state. As noted in [17], when the Trusted OS (TOS) controls the screen, Android and the apps in the Normal World all become unresponsive since the TOS does not handle all interrupts. It thus leads to potential I/O data losses and unpleasant user experience. In PRISM, Android's operations are not affected when a private interface is in rendering. However, the PRISM-based shuffling mechanisms for user input secrecy protection entails inconvenience to users, especially for text input.

TruZ-View [37] reduces the code base of the TOS's graphics subsystem by copying the framebuffer data prepared by the rich OS. Virtualization techniques have been used to protect device I/O. A user-verifiable trusted path [39] builds a secure and authenticated channel between a device endpoint and a program endpoint. Built on top of XMHF [30], the wimpy kernel [40] establishes a secure I/O channel for sensitive applications to use peripherals managed by a commodity OS.

9.2 Discussions

TrustZone versus AVF. TrustZone based schemes rely on the predefined TA's interface. A mobile app can only provide data to be rendered in the predefined position and style and cannot securely process user inputs either. In contrast, an app using PRISM can display whatever contents it needs, and also processes user intentions as per normal. TrustZone-based schemes assume security of *all* software in SW, because any malware therein can easily produce the same interface as the designated TA. The security of PRISM depends on the pKVM. We stress that TAs cannot access the pKVM unless the TOS (or the trusted hypervisor if EL2 is enabled in SW) provides a TA the mapping to the pKVM memory. Moreover, both the TOS and the pKVM expose interfaces (SMC and HVC, respectively) to the Normal World kernel.

Enhanced User Input Integrity Protection. An approach for user input integrity is to enable interrupt virtualization. The pKVM intercepts and sends all the screen coordinates to the secure endpoint, while injecting others as virtual interrupts back to Android. However, this approach has several drawbacks. Firstly, the pKVM code base is significantly expanded to handle I/O operations. Recognizing and handling all the events is an onerous task. Secondly, interrupt interception takes a toll on the system-wide performance. Most importantly, by bypassing Android, PRISM has to properly translate the raw screen events into a UI event. It is a complex and cumbersome function as the phone's screen supports many user operations. For instance, a user may slide his/her finger across a button, which generates a sequence of interrupts. PRISM has to recognize them as a finger-sliding instead of a tapping on the button.

10 Conclusion

We have presented PRISM, a technique for apps to harden sensitive widgets against a kernel adversary. PRISM ensures secure interface rendering and facilitates visual authentication. Built on top of them, an app can use shuffling algorithms to protect user intent secrecy. We have implemented a prototype of PRISM and a testing app on

Table 6: A comprehensive comparison between PRISM with the state-of-the-art of UI security.

Schemes	Approach	User-Bound Flow		Sensitive App-Bound Flow			Visual Authentication
		Security	App controls	Secrecy	Integrity	Sent to	
VButton [17]	TrustZone	integrity	one image	No	No-tampering	Server	State only
TruZ-Droid [36]	TrustZone	integrity	text	Yes	No-tampering	Server	State only
APC [11]	TrustZone	integrity	text	No	No-tampering	Server	State only
SchrodinText [26]	TrustZone	Full	text from server	-	-	-	State only
Rushmore [22]	TrustZone	Full	image from server	-	-	-	State only
SecurePay [15]	TrustZone	Full	text from server	-	-	-	State only
PRISM	Virtualization	Full	whole widget region	Yes	Verification	App	State and apps

Google Pixel 7 with Android 13. The experiments show no performance dip and user experience is barely affected except text input. Moreover, the prototypes show that PRISM can be easily deployed on Android phones for applications to harden sensitive interfaces.

Acknowledgment

This research is supported by the National Research Foundation, Singapore, and the Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Proposal ID: NCR25-DeSSMU-0001). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore, and the Cyber Security Agency of Singapore.

References

- [1] 2022. Microdroid Demo App. <https://android.googlesource.com/platform/packages/modules/Virtualization/+refs/heads/android13-d1-release/demo/> <https://android.googlesource.com/platform/packages/modules/Virtualization/+refs/heads/android13-d1-release/demo/>.
- [2] Simone Aonzo, Alessio Merlo, Giulio Tavella, and Yanick Fratantonio. 2018. Phishing Attacks on Modern Android. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1788–1801.
- [3] Jeroen Beckers. 2021. *New mobile malware family now also targets Belgian financial apps*. <https://blog.nviso.eu/2021/05/11/new-malware-family-now-also-targets-belgian-financial-apps/> <https://blog.nviso.eu/2021/05/11/new-malware-family-now-also-targets-belgian-financial-apps/>.
- [4] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. What the App is That? Deception and Countermeasures in the Android User Interface. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 931–948.
- [5] Sen Chen, Lingling Fan, Chunyang Chen, Minhui Xue, Yang Liu, and Lihua Xu. 2021. GUI-Squatting Attack: Automated Generation of Android Phishing Apps. *IEEE Transactions on Dependable and Secure Computing* 18, 6 (2021), 2551–2568.
- [6] Yurak Choe, Hyungseok Yu, Taeho Kim, Shinjae Lee, Hojoon Lee, and Hyoungshick Kim. 2024. (In)visible Privacy Indicator: Security Analysis of Privacy Indicator on Android Devices. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. 1630–1643.
- [7] Lucian Constantin. 2019. Emergent Android Banking trojan shows app overlay attacks are still effective. <https://www.csoonline.com/article/3455136/emergent-android-banking-trojan-shows-app-overlay-attacks-are-still-effective.html>
- [8] Jonathan Deber, Ricardo Jota, Clifton Forlines, and Daniel Wigdor. 2015. How much faster is fast enough? User perception of latency & latency improvements in direct and indirect touch. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI)*. 1827–1836.
- [9] Yanick Fratantonio, Chenxiong Qian, Simon P. Chung, and Wenke Lee. 2017. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 1041–1057.
- [10] Google. 2024. *Android Open Source Project*. <https://source.android.com/docs/core/virtualization/architecture>
- [11] Google. 2024. *Android Protected Confirmation*. <https://developer.android.com/privacy-and-security/security-android-protected-confirmation>
- [12] Google. 2024. *Android Verified Boot*. <https://source.android.com/docs/security/features/verifiedboot/avb>
- [13] Google. 2024. *Device Identifier Composition Engine*. <https://source.android.com/docs/security/features/dice>
- [14] Google. 2024. *Microdroid : Android Open Source Project*. (2024). <https://source.android.com/docs/core/virtualization/microdroid>
- [15] Radhesh Krishnan Konoth, Björn Fischer, Wan Fokkink, Elias Athanasopoulos, Kaveh Razavi, and Herbert Bos. 2020. SecurePay: strengthening two-factor authentication for arbitrary transactions. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*. 569–586.
- [16] Jiawei Li, Jian Mao, Jun Zeng, Qixiao Lin, Shaowen Feng, and Zhenkai Liang. 2024. UIHash: Detecting Similar Android UIs through Grid-Based Visual Appearance Representation. In *Proceedings of the 33rd USENIX Security Symposium*. 665–682.
- [17] Wenhao Li, Shiyu Luo, Zhichuang Sun, Yubin Xia, Long Lu, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. VButton: Practical Attestation of User-Driven Operations in Mobile Apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 28–40.
- [18] Seita Maruyama, Satoshi Wakabayashi, and Tatsuya Mori. 2019. Tap 'n Ghost: A Compilation of Novel Attack Techniques against Smartphone Touchscreens. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 620–637.
- [19] Charlie Osborne. 2020. This new Android mobile malware targets banks, financial services across Europe. <https://www.zdnet.com/article/this-new-android-mobile-malware-is-striking-banks-financial-services-across-europe/> <https://www.zdnet.com/article/this-new-android-mobile-malware-is-striking-banks-financial-services-across-europe/>.
- [20] Amer Owaida. 2020. Critical Android flaw lets attackers hijack almost any app, steal data. <https://www.welivesecurity.com/2020/05/27/critical-android-flaw-lets-attackers-hijack-almost-any-app-steal-data/>
- [21] Danny Palmer. 2022. *Time to update: Google's Android updates fixes 41 flaws, five critical*. <https://www.zdnet.com/article/android-security-google-updates-fix-these-five-critical-vulnerabilities/> <https://www.zdnet.com/article/android-security-google-updates-fix-these-five-critical-vulnerabilities/>.
- [22] Chang Min Park, Donghwi Kim, Deepesh Veersen Sidhwani, Andrew Fuchs, Arnob Paul, Sung-Ju Lee, Karthik Dantu, and Steven Y Ko. 2021. Rushmore: securely displaying static and animated images using TrustZone. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 122–135.
- [23] Andrea Possemato, Andrea Lanzi, Simon Pak Ho Chung, Wenke Lee, and Yanick Fratantonio. 2018. ClickShield: Are You Hiding Something? Towards Eradicating Clickjacking on Android. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1120–1136.
- [24] Chuangang Ren, Peng Liu, and Sencun Zhu. 2017. WindowGuard: Systematic Protection of GUI Security in Android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 1–13.
- [25] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. 2015. Towards Discovering and Understanding Task Hijacking in Android. In *Proceedings of the 24th USENIX Security Symposium*. 945–959.
- [26] Ardalan Amiri Sani. 2017. SchrodinText: Strong Protection of Sensitive Textual Content of Mobile Applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 197–210.
- [27] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: an extremely simple oblivious RAM protocol. *Journal of the ACM (JACM)* 65, 4 (2018), 1–26.
- [28] Mingshen Sun, Mengmeng Li, and John C. S. Lui. 2015. DroidEagle: Seamless Detection of Visually Similar Android Apps. In *Proceedings of the 8th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. 1–12.
- [29] Guliz Seray Tuncay, Jingyu Qian, and Carl A Gunter. 2020. See no evil: Phishing for permissions with false transparency. In *Proceedings of the 29th USENIX Security Symposium*. 415–432.
- [30] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. 2013. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *2013 IEEE Symposium on Security and*

- Privacy*. 430–444.
- [31] Kai Wang, Richard Mitev, Chen Yan, Xiaoyu Ji, Ahmad-Reza Sadeghi, and Wenyuan Xu. 2022. GhostTouch: Targeted Attacks on Touchscreens without Physical Touch. In *Proceedings of the 31st USENIX Security Symposium*. 137–140.
- [32] Xianbo Wang, Shangcheng Shi, Yikang Chen, and Wing Cheong Lau. 2022. PHY-jacking: Physical Input Hijacking for Zero-Permission Authorization Attacks on Android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 1–17.
- [33] Dan Wendlandt and Adrian Perrig. 2008. Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. 321–334.
- [34] Peter Williams and Radu Sion. 2008. Usable PIR. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 139–152.
- [35] Yuxuan Yan, Zhenhua Li, Qi Alfred Chen, Christo Wilson, Tianyin Xu, Ennan Zhai, Yong Li, and Yunhao Liu. 2019. Understanding and Detecting Overlay-Based Android Malware at Market Scales. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 168–179.
- [36] Kailiang Ying, Amit Ahlawat, Bilal Alsharifi, Yuexin Jiang, Priyank Thavai, and Wenliang Du. 2018. TruZ-Droid: Integrating TrustZone with Mobile Operating System. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 14–27.
- [37] Kailiang Ying, Priyank Thavai, and Wenliang Du. 2019. TruZ-View: Developing TrustZone User Interface for Mobile OS using Delegation Integration Model. In *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy*. 1–12.
- [38] Lingyun Ying, Yao Cheng, Yemian Lu, Yaoguo Gu, Purui Su, and Dengguo Feng. 2016. Attacks and Defence on Android Free Floating Windows. In *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. 759–770.
- [39] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. 2012. Building Verifiable Trusted Path on Commodity x86 Computers. In *2012 IEEE Symposium on Security and Privacy*. 616–630.
- [40] Zongwei Zhou, Miao Yu, and Virgil D. Gligor. 2014. Dancing with Giants: Wimpy Kernels for On-Demand Isolated I/O. In *2014 IEEE Symposium on Security and Privacy*. 308–323.

Appendix

A. User Input Secrecy

In the following, we formally treat the user input secrecy with an information theoretic analysis. Let random variable I represent a user intent and A represent the access pattern of a user operation for I . We prove that knowledge of A yields no advantage in guessing intent I .

A1. Widget State Randomization. The first theorem is for checkbox and toggle button widgets whose states are randomized. The domains for I and A are $\{0, 1\}$. The bit for I means that user intent is “No” or “Yes” while the bit for A is represents a click or absence of a click.

THEOREM 1 (STATE RANDOMIZATION). *For all $a, \iota \in \{0, 1\}$, $\mathbb{P}(I = \iota) = \mathbb{P}(I = \iota|A = a)$.*

Proof. We first prove the theorem under $a = 0$. Note that

$$\begin{aligned} \mathbb{P}(I = \iota|A = 0) &= \frac{\mathbb{P}(I = \iota, A = 0)}{\mathbb{P}(A = 0)} \\ &= \frac{\mathbb{P}(A = 0|I = \iota)\mathbb{P}(I = \iota)}{\mathbb{P}(A = 0|I = 0)\mathbb{P}(I = 0) + \mathbb{P}(A = 0|I = 1)\mathbb{P}(I = 1)} \end{aligned}$$

Note that $\mathbb{P}(A = 0|I = 0) = \mathbb{P}(A = 0|I = 1) = 1/2$, since the state is randomized by the endpoint evenly. Hence, for all ι , $\mathbb{P}(I = \iota|A = 0) = \mathbb{P}(I = \iota)$. The same applies to $A = 1$, which then completes the proof. \square

A2. Widget Position Shuffling. Theorem 2 summarizes intent secrecy of a radio button widget comprising a group of n elements located at n screen positions. We suppose that the adversary knows all positions and all text labels, however, without the knowledge of

the real-time label-position associations. The domain of I and A is $[0, n - 1]$ which represent the indexes in the shadow interface where there is no shuffling.

THEOREM 2 (POSITION SHUFFLING). *For all $a, \iota \in [0, n - 1]$, $\mathbb{P}(I = \iota) = \mathbb{P}(I = \iota|A = a)$.*

Proof. We first fix a as a number in $[0, n - 1]$.

$$\mathbb{P}(I = \iota|A = a) = \frac{\mathbb{P}(I = \iota, A = a)}{\mathbb{P}(A = a)} = \frac{\mathbb{P}(A = a|I = \iota)\mathbb{P}(I = \iota)}{\sum_{i=0}^{n-1} \mathbb{P}(A = a|I = i)\mathbb{P}(I = i)}$$

Since the endpoint randomly shuffle the n text labels, for all user intent i , the probability that it is corresponding to position a is always $1/n$, i.e., $\mathbb{P}(A = a|I = i) = 1/n$ for all $0 \leq i \leq n - 1$. Hence, we have $\mathbb{P}(I = \iota|A = a) = \mathbb{P}(I = \iota)$. Furthermore, the deduction above is independent of a , which completes the proof. \square

A3. Private Keyboard. The third theorem below formulates the security of inputs from a randomly permuted on-screen keyboard refreshing using Algorithm 1. Suppose that the keyboard displays n characters in total. (In our experiments, n is 11 and 38 for the number pad and password keyboard, respectively.) Let $C = [c_1, \dots, c_n]$ be the sequence of characters displayed at n positions on the keyboard on the shadow interface. I is an array of t characters while A is an array of t keyboard indexes. We first prove the following lemma which claims that no matter what the access pattern is generated from the user’s keyboard inputs, the adversary always finds the resulting keyboard layout to be uniformly random.

LEMMA 1. *Let A_t denote a sequence of t indexes exposed to the adversary. For all t and A_t , it always holds that $\forall 0 \leq i \leq n - 1, \forall c \in C$*

$$\mathbb{P}(K_t[i] = c|A_t) = 1/n$$

where $K_t[i]$ is i -th position in the layout after A_t being observed, and the probability is taken over the randomness used by the initial keyboard setup and t clicks.

Proof. We prove the theorem using induction on t which represents the length of the access pattern, i.e., the length of the user’s text input. We use $K_0[\dots]$ to denote the initial layout of the keyboard before any keystroke.

Step 1: Base Case. When $t = 1$, without loss of generality, suppose that $A_1 = \langle \bar{i} \rangle$, i.e., $K_0[\bar{i}]$ is clicked where $\bar{i} \in [0, n - 1]$. Note that the adversary gets \bar{i} only, instead of the corresponding character. We calculate the conditional probability of $\mathbb{P}(K_1[i] = c|A_1)$ for each position in the shuffled keyboard, i.e., K_1 . We evaluate the probability for $i \neq \bar{i}$ and $i = \bar{i}$, for any $c \in C$.

a) $i \neq \bar{i}$: For c to appear at $K_1[i]$, the probability is the sum of the probability of the event that c is at $K_0[i]$ and is not swapped and the probability of the event that c is at $K_0[\bar{i}]$ and is relocated to $K_0[i]$. Hence, we have

$$\mathbb{P}(K_1[i] = c|A_1) = \frac{1}{n} \cdot \frac{n-1}{n} + \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n}$$

Note that Algorithm 1 relocates the clicked letter across the entire keyboard.

b) $i = \bar{i}$: For c to appear at $K_1[\bar{i}]$, the probability is the sum of the probability of the event that c is at $K_0[\bar{i}]$ and is not relocated and the probability of the event that c is not in $K_0[\bar{i}]$ and is relocated

there. Hence, we have

$$\mathbb{P}(K_1[\bar{i}] = c|A_1) = \frac{1}{n} \cdot \frac{1}{n} + \frac{n-1}{n} \cdot \frac{1}{n} = \frac{1}{n}$$

Hence, this proves the equation of theorem in the base case.

Step 2: Induction. Suppose that the equation of the theorem holds for t for $t \geq 1$, i.e., $\mathbb{P}(K_t[i] = c|A_t) = 1/n$ for all i and c . In the following, we prove that it also holds for $t+1$. Let $A_{t+1} = \langle A_t, \bar{i} \rangle$.

We evaluate $\mathbb{P}(K_{t+1}[i] = c|A_{t+1})$, the probability of the event that, after t clicks with access pattern A_t , the $t+1$ -th click is at $K_t[\bar{i}]$ and $K_{t+1}[i]$ holds c after executing Algorithm 1. Similar to the base case, we consider two scenarios according to i .

- a) $i \neq \bar{i}$: For c to appear at $K_{t+1}[i]$ after A_{t+1} , the probability is the sum of the probability of the event that c is at $K_t[i]$ and is not swapped and the probability of the event that c is at $K_t[\bar{i}]$ and is relocated to $K_t[i]$. According to the induction hypothesis, the probability that c appears at $K_t[i]$ after A_t is $1/n$ and thus the probability that $K_t[\bar{i}]$ will be chosen for the input letter is also $1/n$. Hence, we have $\mathbb{P}(K_{t+1}[i] = c|A_{t+1}) = \frac{1}{n} \cdot \frac{n-1}{n} + \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n}$.
- b) $i = \bar{i}$: For c to appear at $K_{t+1}[\bar{i}]$ after A_{t+1} , the probability is the sum of the probability of the event that c is at $K_t[\bar{i}]$ and is not relocated and the probability of the event that c is not in $K_t[\bar{i}]$ and is relocated there. Hence, we have $\mathbb{P}(K_{t+1}[\bar{i}] = c|A_{t+1}) = \frac{1}{n} \cdot \frac{1}{n} + \frac{n-1}{n} \cdot \frac{1}{n} = \frac{1}{n}$. Thus, the lemma also holds for $t+1$, which completes the proof. \square

THEOREM 3. For all length t , for all possible A , $\forall j \in [0, t-1], \forall c \in \mathcal{C}$, we have $\mathbb{P}(I_j = c) = \mathbb{P}(I_j = c|A)$, where I_j is the j -th character in user intent I .

Proof. For all $A_t = [a_1, a_2, \dots, a_t]$, we have

$$\mathbb{P}(I_j = i|A_t) = \frac{\mathbb{P}(I = i, A_t)}{\mathbb{P}(A_t)} = \frac{\mathbb{P}(A_t|I_j = i)\mathbb{P}(I_j = i)}{\sum_{i=0}^{n-1}\mathbb{P}(A_t|I_j = i)\mathbb{P}(I_j = i)}$$

Note that I_j has is independent of a_0, a_1, \dots, a_{j-1} . According to Lemma 1, I_j still results in a keyboard with a random layout and therefore $\mathbb{P}(a_j, a_{j+1}, \dots, a_{t-1}|I_j = i) = 1/n^{t-j}$. Hence, we have $\mathbb{P}(A_t|I_j = i) = 1/n$ and thus $\mathbb{P}(I_j = i|A_t) = \mathbb{P}(I_j = i)$ which completes the proof. \square

B. User Acceptance Survey Results

Participants were given two identical sets of four questions to gather their views on the randomized keyboard and the randomized number pad. The questions are listed below.

- (1) I believe that using a randomized keyboard/number pad layout would enhance my online security.
- (2) I am willing to adopt a randomized keyboard/number pad layout if it means better protection against key logging malware.
- (3) The potential security benefits of a randomized keyboard/number pad layout outweigh the inconvenience of learning it.
- (4) I am concerned about the learning curve associated with using a randomized keyboard/number pad layout.

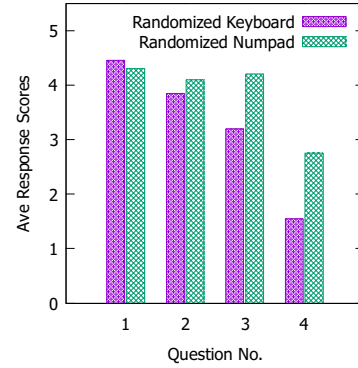


Figure 11: User Acceptance Scores for randomized keyboard and number pad.

Figure 11 summarizes the user acceptance scores and shows that while the users generally accepts that the randomized layouts will provide better security, users are concerned with the learning curve required as seen in the lower scores for Question 4.