SoK: The Evolution of Trusted UI on Mobile

Davide Bove
IT Security Infrastructures Lab
FAU Erlangen-Nürnberg
Germany

ABSTRACT

In mobile security research, one of the many challenges is the lack of trusted user interfaces (TUI). Currently, users are not able to reliably identify which screen content is genuine and which is potentially manipulated. The concepts of trusted displays and user interfaces, which offer a high confidence in the trustworthiness of screen contents, are not universally implemented on current consumer devices. In this paper, we systematically analyze the developments in the field of TUIs on mobile devices over seven years. We present a new taxonomy to define and categorize challenges and issues in current UI designs, with a focus on issues that negatively affect the security of the whole OS. In addition, we suggest directions where contributions in research could solve these issues.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; *Hardware security implementation*; Trusted computing; • **Human-centered computing** → Graphical user interfaces.

KEYWORDS

trusted ui; mobile security; android; trusted execution

ACM Reference Format:
Davide Bove. 2022. SoK: The Evolution of Trusted UI on Mobile. In *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security (ASIA CCS '22), May 30–June 3, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3488932.3517417

1 INTRODUCTION

User interfaces are the primary method of human-computer interactions. As humans make mistakes when using technology, a lot of effort is put into usability such that the user of a system can efficiently operate it. Research has shown that usability and security correlate, meaning that "decreasing the usability can lead to less security" [51]. The security of mobile devices is especially important, as more and more parts of everyday life is transferred to these devices. consumer mobile devices have become central to people's identities, such that a single security incident can cause a lot of damage.

On personal computers, there are a number of indicators that help identify a specific application, and the well-informed user can more easily investigate the actual application causing problems. Mobile devices unify the UI look-and-feel of all applications and hide the details, even for technical users. In consequence, users need to trust the system to show them exactly what they expect to see on screen. In this work, we show that this trust can not be guaranteed with current designs. From the 500 most popular apps on the Google Play Store, 27.2% use overlays [32], while research shows that overlays can be used in malware to deceive the user [14, 29, 42, 59]. For developers that use the Accessibility API, which assists users with disabilities navigating or using apps, only 0.37% of popular apps use them, and it is most often used for privileged access to notifications or to kill processes [25]. The API is also abused by malware to control the user device [15, 37, 54]. It is the balance of features versus security that sometimes leads to severe problems in the mobile ecosystem.

Our main contribution in this work is to survey and analyze the current state of trusted user interfaces (TUI) with a focus on the Android OS. We decided on a systematization of knowledge, which we argue is mandatory to assess the current state of research in the field. There is a need for a systematic review of the research field as the arms race of new features, abuses of these features and countermeasures has been rapidly going forward for the past years. For this reason, the research field needs to be systematically assessed to highlight the main research movements and to discover unexplored or unanswered issues. Specifically, we want to contribute a categorization of current challenges and proposed solutions in order to minimize redundancy in approaches. We also see a shift from system-level defenses to specialized hardware-based solutions on the market, which may be the consequence of research contributions to the field. Therefore, our research for this systematization will focus on the following research questions:

- **RQ 1: What issues are identified?**
  We survey the actual problems targeted by security research in the field of UI security on mobile devices.
- **RQ 2: Which approaches are taken against specific weaknesses?**
  We look at the proposed solutions of current research and especially focus on the *depth of access* of countermeasures, the *feasibility* in real-world environments and at *trends* of research.
- **RQ 3: What challenges remain to be addressed?**
  With this question, we concentrate on issues that have not been considered yet for future research efforts.

We included publications from conferences and journals within the last seven years, from 2014 to 2020. Older papers often focus on outdated and no more supported OS versions and features. We included research that presented new attacks or security vulnerabilities that affect UI security, as well as presented defense strategies against such attacks. In order to get a full picture of the solution space, we also looked at UI security publications outside the mobile domain (see Appendix C). Often, papers get overlooked when they are not published in top conferences and journals. Therefore,

This work is licensed under a Creative Commons Attribution International 4.0 License.

ASIA CCS '22, May 30–June 3, 2022, Nagasaki, Japan
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9140-5/22/05.
https://doi.org/10.1145/3488932.3517417

we extracted additional bibliography if papers heavily reference another publication. As a result, we also cover relevant literature that might be overlooked by a search engine or our own search methodology.

Furthermore, **our contributions** are:

(1) We provide an overview of UI-related issues on Android that are presented in literature. We analyze the main attack vectors and design issues in section 3.
(2) We evaluate current research on defense mechanisms against UI attacks as well as existing countermeasures in current devices (section 4) and beyond the mobile security field (see Appendix C).
(3) We identify and analyze systemic weaknesses discovered through our systematization and propose directions and areas for future research on the topic in section 5.

2 BACKGROUND

This section provides the background to understand UI security in the context of mobile devices and describes the fundamental software and hardware design of current devices.

2.1 Android UI Security

The system design of Android is based on functionality of the Linux kernel, and a number of security features rely on it. A core concept is the Android Application Sandbox, which strongly separates apps from each other. Any interaction or exchange between apps is done using some form of inter-process communication (IPC): either through Intents, which allow sending messages between apps or start services, or using the `Binder` kernel driver which allows to directly invoke remote function calls. The sandboxing extends both to an app's code and its data. In the context of UIs, this means that no application can access the UI or the memory of another app. But the OS offers a permission-based model to grant access to specific resources. These permissions are enforced and verified by the system and make sure that no app involuntarily exposes its data to other apps. For the users, these security measures are not visible, as they can interact with every app equally and can "share" most data with other apps.

2.2 ARM TrustZone

A Trusted Execution Environment is a CPU feature that offers an isolated environment for processing data. The goal is to have a secure execution of specific applications even in the presence of a compromised OS. Such applications, called trusted apps or trustlets, are responsible for processing security-critical operations, such as encryption or authentication. TEEs are hardware-supported technologies and ARM TrustZone implements such a model, which is widely used in current mobile smart devices. TrustZone introduces the concept of two security domains called "Normal World" and "Secure World" (more in Appendix A). The former is where the regular operating system is running, and the latter runs an additional TEE OS with a separate user and kernel space layer. The regular OS can not access the TEE and the communication is done through a secure monitor software running in a privileged layer. This monitor can be interacted with using special Secure Monitor Call (SMC) instructions. As an example, SMCs trigger the context switch between the two worlds. The TEE also defines shared memory regions that can be used by both normal and secure apps to exchange data.

Even though Google offers an own TEE implementation called Trusty TEE [7], most device vendors use customized implementations, therefore the number and type of trusted apps differ from one device to another. For iOS devices, Apple does not use TrustZone, but implements a similar technology in their processors called Secure Enclave [11].

2.3 Threat Model

We provide a categorization of threat models based on our systematization findings. Since attacks and defenses evolved over the years as devices and operating systems evolved, we chose a broad categorization based on attacker capabilities and the depth of access required to implement a defense. Every defense in section 4 has their own threat model, may work only against specific attackers or mitigates only specific vulnerabilities. But we found a consensus about a few major assumptions:

- There are different privilege levels in a system. If an attacker can execute arbitrary code inside any of these levels, that level and all the levels below it are deemed untrusted.
- Once a level is untrusted, only a defense at a higher privilege level may be used.

While there might be minor nuances in the presented papers, there are three threat models that are found in the majority of academic publications. Since mostly working on ARM processors, we use ARM's vocabulary of Privilege Levels (PL) to describe the different stages. This categorization is not extensive and only includes the models and assumptions we found in the literature:

- **User Mode**: An attacker can execute code in user space (PL0). This may be a regular app on the device.
- **OS Mode**: An attacker can execute code in OS and kernel space (PL1). This may be an app that has root access or uses an exploit to gain system-wide access.
- **TEE mode**: An attacker can execute code in the context of a Trusted Execution Environment (TEE). This may be through a vulnerability in a trusted app, in the trusted OS, or by deploying a custom app.

Denial-of-Service attacks where attackers disable functionality or stop the user from using specific features are **not considered** in the threat model, since attackers with OS mode access can just shut down or reboot the device indefinitely.

3 DESIGN ISSUES

This section presents the main design issues of the Android operating system regarding UI security. The issues represent the main focus of current research and look at several aspects that contribute to weaknesses of the UI security. For an overview of changes to Android throughout different versions, see Table B.1.

3.1 Window Management

Android's window management consists of system-controlled and user-controlled elements. The system manages the Activity stack,

which controls the navigation between different windows, but also takes care of drawing the windows. Users, in most cases the developers of apps, take care of the design of an app, from the general color scheme to every single view element that is shown during the execution of an app. Due to Android's design decisions, users have great freedom in styling, but the potential for abuse is just as high.

`I01`. Missing indicators: Current Android devices do not make it easy for a user to assess with confidence whether the current screen content is controlled by a legitimate app or not. Not only do apps have control over their own windows, but they can also use full-screen UIs to take over most of the screen space. In practice, this means that any application on a device can impersonate another legitimate app. This design flaw leads to phishing attacks [9, 29], clickjacking [17, 31, 35] and denial-of-service attacks [48, 49]. Security measures that mitigate some of these vulnerabilities have been implemented in newer versions of Android, but the root issue remains, as the operating system offers no identification of the screen contents by design and is therefore inherently vulnerable against this class of attacks.

One major design flaw in modern mobile operating systems is the lack of identifying information on what is being displayed on the screen. A regular user has no concept of windows and an app's user interface, called Activity, is mostly defined by the developer of the application. On Android, users have only two reliable options for exiting apps: pressing the Home button or using the app switcher. The back button, which is often used to exit an application, can be deactivated by the application.

For identifying which app is currently running on the screen, users need to enter an app's information screen. If the selected app uses a regular Activity, users can enter the app switcher, touch the app's icon shown on top of the view, and select "App info". In general, the App Info screen shows basic information such as the app's name and its icon. More tech-savvy users may trigger different app settings here, such as notifications or permissions. But still, this screen does not reveal any identifiable information that helps distinguish a legitimate app from a malicious or misdirecting app. This is therefore not an issue of technical proficiency of the user, but a design flaw in the original Settings application of Android.

If you compare this to other concepts, such as HTTPS indicators in browsers, it becomes pretty obvious that any application can pretend to be any other app, because there is no indicator that hints at a problem with the current screen contents. There are no systems in place to detect such impersonating apps on the device. This leads to the aforementioned problems, such as phishing, clickjacking, or other deception attacks. Most of the attacks that are attributed to lacking indicators can be mitigated with an additional indicator that shows the origin of an app, similar to how Extended Validation certificates work in the context of browsers [14].

`I02`. Unprivileged access to overlays: In recent research, there has been a focus on UI features that can be abused to trick and deceive users. In general, multitasking and having multiple windows on Android was not possible before Android introduced Multi-Window Support in Android 7.0 [5]. Overlays are windows that overlap other windows on the screen, and they have existed since the first version of Android. They allow an application to create custom "floating" windows on top of other Activities. In the case of semi-transparent popups and overlay windows, the problem of lacking indicators is even worse. Overlays can be placed on the screen from any active app and do not appear in the app switcher. They have no title bar, no mandatory layout and are not treated like regular windows.

A major problem that has affected UI security is the broken permission model around overlays prior to Android 7.0. Apps on newer OS versions not only have to declare the *SYSTEM_ALERT_WINDOW* permission to be able to use overlays, they also have to ask the user explicitly to enable the feature in the app's settings screen. If the same app is installed on an older device, Google grants this permission automatically when installed from their Play Store, thus without requiring the user's consent.

Even without the permission, developers can abuse Toast message to show overlays containing arbitrary content, such as UI controls, text or colored backgrounds. In general, Toast messages are supposed to be short text messages shown by an app for the user. Developers can decide between two duration settings, both defined inside the operating system. Regardless of the actual value of these settings, apps can repeatedly destroy and recreate Toast messages without any delay, leading to a permanent display of such a message. In addition to this, since the Android API allows defining custom views for Toast messages, they can have any appearance and span the full screen. Therefore, even though Android implements actual security measures to ensure that apps do not cover the screen without user consent, it still allows Toast messages to do this. As far as we know, this behavior has been deprecated in Android 11 and will most probably be gone for new devices.

`I03`. Overlays covering information: No matter if the user willfully enables overlays, is tricked into doing it or Toast messages are used, overlays have several security implications. In stacking window managers like Android's Window Manager, windows have a z-order, which is a numerical value that determines the order in which elements are drawn on the screen. When two UI elements overlap each other, the z-order determines which element is drawn on top of the other.

Prior to Android 8, overlay windows could overlap most of the screen, including the status bar, the integrated keyboard and any system settings. This allowed to fully control the screen and manipulate the user into performing any action, such as enabling Accessibility Services [31]. After exposing this design flaw, Google improved upon overlay windows and changed the z-order of overlay windows such that the menu and status bar can not be covered by an overlay anymore. In addition, Android introduced an API to detect and hide all overlays, which is used for some specific dialogs inside the system settings, but is not accessible to third-party apps [17].

Unfortunately, the design flaws presented by [31] are still present today. Especially, the problem of Context Hiding is still not solved universally. Context Hiding means that an overlay is used to hide the screen contents around UI controls such as buttons, therefore removing any indication about which window the control belongs to. In a confirmation dialog, hiding the dialog text and replacing it with a different text might trick the user into accepting whatever an attacker wants to have confirmed. Another scenario could be that a malware wants to hide its actions by manipulating the screen into

showing different values, such as a manipulated banking account balance.

As mentioned above, the protections introduced by Google are only implemented in a number of places in the system settings source code. Especially a method to detect and hide every overlay, which might completely solve the problem of Context Hiding, is currently private and can only be used with system apps. Even worse, this method is not used all over the OS, but only in selected parts of the Settings app. Therefore, apps can still use overlays to cover several parts of the system settings UI and trick the user into enabling settings. Several critical settings can still be activated this way, like the Developer options, the Android Device Bridge, the device encryption and even the security PIN used to unlock the device. Several tests on recent devices showed that the virtual keyboard is also not protected, allowing attacks that log the user input such as the "Invisible Grid Attack" [31].

`I04`. Apps can hijack the window stack: Android manages the life cycle of every app, from creation and destruction of windows to the management of tasks and processes. Multitasking in Android means that multiple processes run in the background, but only one application at the time is considered active. Tasks and windows are managed in stack structures and mostly controlled by the OS, but apps can manipulate some aspects of task management through the developer API.

In detail, Activities in third-party apps can define an attribute `taskAffinity`, which indicates to the OS which task an Activity prefers to belong to. In general, Android associates an Activity to a specific task, such that a window is not removed from memory until the task is done. A second attribute, `allowTaskReparenting`, breaks this limitation. As both are developer-controlled settings, the methods can be combined and abused in order to allow a malicious app M to be associated with a legitimate app A. In practice, this means that the malicious app can put itself on top of app A or redirect the user to a malicious activity to execute various attacks, such as stealing credentials, locking the device or monitoring user actions [49].

Similar to other flaws in Android, the purpose of these features is rather limited. Some apps use the Task Affinity attribute as a means to implement plugins for existing apps, such as additional controls for the phone app. In the work of [49] about 3.96% of 6.8 million apps were found to use the feature. It is not clear if these features were implemented for this purpose, but their presence and their current implementation undermines the security of legitimate apps and poses a great threat to the user.

3.2 Accessibility

Accessibility services have privileged access to UI elements and system resources and are able to read screen contents as well as simulate user input. This allows for great possibilities to enhance the user experience of users with disabilities. Over the years, many developers have implemented such services in their apps, but without the added benefits. Instead, the apps were enhanced with functionalities that are not accessible through the regular Android developer API.

`I05`. Lack of alternatives: In a study, it was discovered that most apps that implement accessibility services use them to access system notifications, kill background processes, execute automated actions and autofill text [25]. For some of these use cases, Google already provides safe alternatives, such as the Autofill framework [2] or the `NotificationListenerService` API to read notifications. In other cases, such as with automation, there is no alternative, therefore "task manager" apps request accessibility access to force-stop other apps. Since most new features are not ported to older Android versions or do not get deployed on unsupported devices, apps still use accessibility services, which have existed for years, to bridge the gap and support these devices.

`I06`. Highly privileged system access: The Application Sandbox is a main feature of Android's security model. This includes sandboxing such that apps have no access to other apps' data or code. Accessibility services are designed to have access to privileged actions that affect the whole operating system, such that part of this isolation is circumvented. Apps with an active accessibility service can monitor user actions, retrieve the window contents of any other apps and the system, and take actions for the user. In order to use accessibility services, apps need to request the `BIND_ACCESSIBILITY_SERVICE` permission. When installed through the official Play Store, a user receives no warning that an app includes an accessibility service.

In most cases, accessibility services need to be explicitly enabled by users in the System Settings app. In stock Android, the user is warned about the action with a modal dialog that contains a list of actions the services "needs to" do. The description of the specific actions is controlled by the system and constructed from the capabilities that developers declare for the service. Different from app permissions, accessibility services do not provide a method to selectively disable or revoke single permissions. Therefore, once such a service is activated, the requested information is provided to it without limitations. While developers can specify and filter the amount of data their service receives, this limitation is only self-imposed and not reflected by the system's warning dialog.

The power of accessibility services can be observed when trying to simulate user input. Once a service is activated, it can execute input actions and react to UI changes. Active accessibility services can enable other services, give additional permissions to apps and sniff any data entered into apps [35]. We reevaluated the techniques for current versions of Android 10 and found that devices are still "vulnerable" to malicious accessibility services. This has major consequences for the security of the system:

- Since an app can give itself all the permissions it needs, it is possible that such a service uses overlays to cover up its actions, providing some stealth to a malware.
- If a malicious service is enabled, there is no way to disable or limit its functionality.
- Using simulated input, a service can install additional apps to gain more foothold in a device. Also, a service can increase persistence by monitoring the system to detect and prevent malware removal attempts.

In summary, once a malicious accessibility service is active, the user loses any control of the device. The lack of isolation and fine-grained permissions is a serious design flaw and makes accessibility services the most dangerous attack vector for Android.

RQ 1: The main problems involve specific Android features or implementation bugs that undermine Android's security and sandbox model. Most security-relevant issues are caused by design flaws in window management, including overlays and the window stack, and accessibility services with their privileged access to UI.

4 DEFENSES OF THE UI

In this section, we present our categorization of defenses in order to identify and get an overview of solutions to the design issues discussed before. By putting the focus on different fields where countermeasures are applied, we identified several improvements that solve or may solve specific issues. With every defense method, we identify how to use it and if it applies to current real-world applications. An overview of all the defenses with regard to their implementation is given at the end of the section (see Table 1).

4.1 Countermeasures in the Android OS

Over the years, Google has received a number of reports regarding the issues identified by scientific findings. In fact, some new UI security measures have been implemented and the concepts behind them are described in the following sections.

D01. Touch Filtering: Originally, the term *Touch Filtering* is derived from a setting in the Android SDK that is called `filterTouchesWhenObscured`. When this flag is enabled for any UI element of an Activity, the operating system will filter out input events (e.g., touch, drag) for the selected element if another window overlaps it. This means that if there is a dialog, a Toast window or any of the overlay windows above the selected element, no touches will be detected by the app. A related security measure is the *Obscured Flag*, a special flag that signals the app that a window is overlapping the app. The flag is accessible by an app through the `MotionEvent.FLAG_WINDOW_IS_OBSCURED` API and is effectively an indicator that an overlay is above an app. The affected app can react to this event, for example by showing a warning or asking the user to disable the overlay. This method has existed since Android 2.3 and has been covered in relevant research [31, 47]. A flaw in the implementation of the Obscured Flag is that a touch event has to go directly through an overlay in order to be detected. Therefore, if a window is only partially covering an element and the user touches the part that is not covered, no flag is set, and no overlay window is detected. By using opaque overlays around protected UI elements (see **I03**) this security countermeasure is circumvented easily.

In previous Android versions, Android introduced a *Partially Obscured Flag*, which was reserved to system apps only [17]. With Android 10.0, it was made available to developers and is accessible through `FLAG_WINDOW_IS_PARTIALLY_OBSCURED` of the `MotionEvent` API. It works in the same way as the first flag, the difference being that the flag is also set when only parts of the elements are covered [4]. As a result, the flaws of the Obscured Flag were finally resolved. Unfortunately, this updated countermeasure does not necessarily help with the attacks that it is supposed to prevent. As an example, if a button is protected by the Partially Obscured Flag, a malicious application might still cover everything around this button. As long as no overlay overlaps the button, the app is not notified of the presence of overlays. Therefore, Touch Filtering as a defense mechanism for apps does require additional security measures in order to be effective, such as Window Punching [17, 34].

D02. Limiting overlay priority: Overlays are a predominant cause for issues in UI security research for Android. Therefore, Google decided to overhaul the implementation of overlays, without limiting too much the freedom to use them in apps. There are Toast windows, system windows, error windows and much more. The different types are used to prioritize them when the OS composes the screen contents. The decision of which window to draw on top of other windows is based on this categorization. Since Android 6.0, third-party apps can no longer decide the category of their windows. Developers can only assign the attribute `TYPE_APPLICATION_OVERLAY` to their custom windows. This newly introduced attribute and the deprecation of previously allowed types is part of a major revision of how the screen is composed inside the OS. While apps can still create overlay windows that cover the whole screen, they can no longer obscure elements with a higher priority, such as the status bar or the navigation bar. This feature alone does not fully prevent any overlay attacks, only the combination with the next two defenses can help the user mitigate a possible overlay attack.

D03. Additional indicators: With the introduction of prioritized overlay types, Android included additional measures to signal the user that an overlay is being displayed. When a new overlay is created and active, Android displays a permanent notification in its notification drawer. The notification message typically reads: "*AppName* is displaying over other apps" (see Figure 1). With an additional touch on the navigation, the user is lead to the app's setting page, where the overlay permission can be revoked. Depending on the default settings, an icon is shown on the status bar, showing that a new notification was created. Therefore, if the user notices that an app is creating overlays and covering the screen, they may use this notification to actively disable them for the specific application. The only drawback to this defense mechanism is that there are three ways to prevent the notification from showing.

(1) **Disabled notifications**: The user can easily disable the notification by changing the configuration of the Android System app, which is responsible for showing the indicator. This is expected behavior and some users might do it on purpose for apps they trust, in order to keep the notification from appearing. For an attacker, disabling the setting is only possible if they can persuade the user into doing it, for example by social engineering or using overlays to trick the user into clicking the corresponding checkbox.
(2) **Customized notifications**: Android allows apps that show overlays to use their own permanent notification that replaces the one shown by the operating system. An app that wants to prevent the user from accessing the overlay settings screen can choose to have a dummy notification that shows no text at all and that does nothing if activated. Android will then display a small icon in the notification that needs to be activated by the user to display an overlay warning (see Figure 1).
(3) **Timed deactivation of overlays**: A third option to prevent the user from noticing the overlay warning is making any

overlays disappear before the user actually sees the notification. For this, an app has to detect when the user is activating the status bar or looking at the notification. A practical way for an app to do this is to put an own Activity in the foreground (e.g., by repeatedly opening the activity) and using the `onWindowFocusChanged` callback to detect when the focus is taken from the Activity. If the screen is obscured by an overlay, the only remaining action for the user is to pull down the status bar. In that case, the callback is triggered, and the malicious app can temporarily hide its overlays to make the OS notification disappear.
More elegant ways to detect and hide notifications from the user might use side-channels of the OS. We argue that while timing attacks are complex and hard to execute in practice, they are a viable way and can be very effective against unaware users.

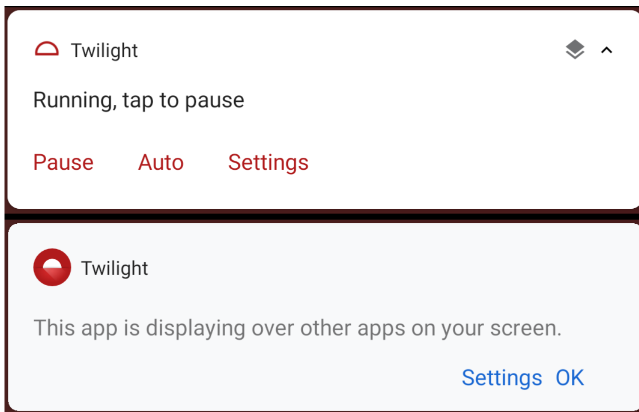

Figure 1: Above: Custom app notification with a clickable "overlay" icon in the top-right corner. Below: notification after touching the icon.

All the above methods that Google implemented for detecting overlays and identifying apps that use them are useless if there would be no way to disable them. This is where more visible defense mechanisms are introduced.

`D04`. Hiding overlays for critical dialogs: Since overlays have the capability not only to cover the screen, but also to catch any input, they are effectively capable to lock up the screen. As discussed in section 3, overlays can also be used to hide context information and trick the user into enabling critical system settings. Therefore, Android has implemented a method to hide all visible and invisible overlays during certain interactions. After the interactions, the overlays are restored to their original position. This security measure is pretty much unknown, but it can be found in the source code of the Settings app [17].

By looking through the code and inspecting different OS versions (from Android 7.1.2 to 10.0) on a Pixel 3 device, we identified several situations where this defense is triggered:

- During manual app installation (side-loading)
- Confirmation dialog when enabling accessibility services
- Confirmation dialog when enabling device admin apps
- App permissions screen
- Overlay Settings screen (for Android < 10). Android 10 removed this feature, which most likely is an oversight, as it was introduced again in Android R.

Most of these situations include a confirmation dialog that enables a critical system setting, but there are more situations where we would expect this measure to work:

(1) In the Notification settings of an app
(2) The "Install Unknown Apps" setting of an app that enables an app to install other apps
(3) When confirming a factory reset
(4) In the "Device Security" dialog when enabling or disabling a screen lock
(5) For the "OEM unlocking" setting in the Developer Settings that unlocks the bootloader
(6) For the "USB debugging" option in the Developer Settings
(7) In the Play Store when installing an app

For Android 11, we found that the hiding overlay defense was fully implemented for the whole Settings app and the App settings, therefore covering item 1–6 of the list above. There was no protection regarding the Play Store (item 7).

Since only system apps are allowed to trigger the overlay hiding feature, it can not be used by developers to protect their apps. While some authors argue that this defense technique is "too powerful to be made available to third-party apps" [46], it would make sense to open up the API and make developers decide for themselves. Overlays are overpowered windows, and allowing apps to limit this power would counterbalance the feature and stop many of the issues caused by overlay attacks. The defense could be implemented as every other regular API with respect to Android's security model, using the permission model to "allow an app to temporarily hide overlays" and limiting its use to when an app is active. In this case, backward compatibility is a non-issue, as any case of abuse may lead to the user removing the offending app, especially if it breaks other overlay apps.

In summary, the new overlay implementation limits some third-party usage of overlays, prioritizes system overlays and is able to hide third-party apps. A new permanent indicator is used to signal the user that an app may be covering the screen. We observed an increased use of the "hiding overlays" technique among system apps and might be seeing more system apps use them. A last feature was identified in recent Android versions that may improve on some issues with UI security. This feature is backed by the TEE solution that is used in most modern Android devices and thus offers an increased defense even in the situation of a full takeover of the operating system.

`D05`. Secure system dialogs: Android 9 introduces the Android Protected Confirmation functionality, which allows apps to prompt the user to confirm a statement. While the API is available through the SDK, the functionality requires special hardware support, which initially was only offered by Google through their Pixel 3 and Pixel 4 devices. From the user perspective, Protected Confirmation is a screen that is shown in full-screen mode and which displays a small message to confirm. A confirmation of the message is done either

through pressing one of two hardware buttons (usually the power and the volume-up button) or by using software buttons on the screen. The app that wants to use Protected Confirmation can define the text to show inside the prompt. Prompt lengths are limited and no other customization of the UI is possible. The prompt respects some accessibility settings, such as color inversion or increased font size, but the placement of labels and icons is predetermined and can not be changed or manipulated. A 1-second delay in the interaction with the dialog prevents users from accidentally confirming a message that pops up unexpectedly. We also observed that if any accessibility services are enabled, Protected Confirmation can not be used.

As the source code is publicly available through the Android Open-Source Project (AOSP), we looked into the internal implementation and found **libteeui**, the library responsible for drawing and rendering text and UI elements from the TEE context [22]. The `libteeui` library is implemented in C++ and includes several routines to render UI elements such as button or text labels, but also fonts and images. Layouts are defined using a Domain-specific language (DSL) similar to C++. The localization of the predefined texts are hard-coded into the library for all languages supported by Android. A secure input method is implemented to communicate with the physical hardware buttons of devices. According to the official documentation, communication with a touch controller is possible in order to support input through "on-screen software buttons" [6]. The resulting data produced by this app is encoded into a Concise Binary Object Representation (CBOR), a proposed standard for binary encoding of data [16]. This data is returned to the user app through the corresponding API call and needs to be signed with a cryptographic key in order to form a signed statement.

Both server and client benefit from integrity checks of the prompt data. If the message is generated on the server, the server can later assess with more confidence if the same message was displayed to the user. When an app creates a signing key to be used with Protected Confirmation, it needs to call the method `setUserConfirmationRequired`. This adds a specific byte sequence to the associated certificate, marking it as a Protected Confirmation key. This method has a second *undocumented* effect: during the signing process, the prompt text passed to the UI and the text returned by `ConfirmationUI` are compared in order to find manipulations, and an exception is thrown if the comparison fails. Assuming a strong **OS** type attacker, a modified or manipulated app may show the user a specific message but make the app sign an attacker-controlled one. In this case, it is even more important for the receiving server to correctly implement certificate checks. A check for the tag ID `trustedConfirmationRequired` in the certificate may reveal potential issues with the key creation or the confirmation process on a user device [8].

In summary, Android Protected Confirmation offers a sound method to obtain a valid decision from the user, verifiable through the use of established cryptographic methods. The UI offers only little attack surface, as it is implemented as a trusted TEE app and does not offer customization options. As the method is designed to be used in combination with a remote party, implementation errors by app and server back-end developers may undermine the security guarantees of Protected Confirmation.

4.2 Kernel- and OS-based methods

The majority of countermeasures we found in our systematization are based on system modifications. These affect the OS, but also the underlying kernel or user-facing APIs of the Android framework. Regarding the current issues of UI on Android, our findings reveal a focus on additional measures to detect potentially dangerous actions, such as clickjacking or app hijacking attempts.

`D06`. Overlay detection: As described in section 3, there are several design decisions that influence the potential threats from overlay attacks. Still, a number of legitimate apps make use of overlays to enhance the user experience or provide services that would otherwise be not possible. The most cited examples of legitimate overlay usage are the Facebook Messenger [27] app, which uses profile pictures on the screen as a shortcut to a conversation. On the other side, Twilight [52] uses full-screen transparent overlays to dim the screen and reduce the amount of eye strain for the user. There is a discussion going on about what defines "legitimate" uses of overlays. This is crucial for finding the right balance between freedom and security, as removing overlays from the OS may exterminate clickjacking attacks on Android, but also break many existing applications. As overlays have a global impact on a device, being able to cover both third-party and system apps, most solutions require modifications of the OS. Overlay windows have specific attributes that can be used to detect them. These include, but are not limited to:

- window position, size and z-value
- color and transparency (alpha) level
- creation flags (which control window behavior)
- host application (identified by package name)

While one can argue about the correct combination of these attributes in order to detect malicious usage, a survey over several apps from the Play Store determined how real-world apps use overlays [46]. From 454 apps from the Play Store, 15 from the F-Droid repository and 10 "screen filter" apps that request the draw-on-top permission, a total of 60 samples are manually analyzed by the authors in order to find characteristics of overlay usage. Even though the analysis in the work can not be considered representative for every possible legitimate use, the authors conclude that most overlays are created at the screen margins, are opaque and can be interacted with by the user (they are *touchable*). Furthermore, overlays are only used when the respective app is active. When an important message is to be shown, the overlays are created in the center of the screen. For the category of screen filters, all apps are using transparent full-screen overlays marked as *not touchable*. Based on these insights, a more effective detection can be implemented. Since the set of real apps is rather small, a more elaborate study might reveal more characteristics of benign behavior. In addition, malware samples using overlays may be analyzed in the same way in order to extract characteristics of malicious behavior.

`D07`. Clickjacking detection: When looking specifically for clickjacking attempts, the majority of touch events is produced by the user. For their Clickjacking Detection System (CDS), the authors define four conditions for detecting a potential clickjacking attempt [55]:

- the *receiver window* of a touch event and the currently active app are different
- the z-value of the receiver is higher (indicated by the *layer value* which determines the window type)
- the window is marked *not touchable*, which means it does not receive touch events
- the alpha value of the receiver window is greater than 0.95, meaning that the overlay is mostly opaque

An abstraction of the above clickjacking detection techniques is presented with ClickShield, where image analysis is used to detect overlays [46]. With a method called *Deblending* an algorithm takes in the expected screen content consisting of the currently active app *DST* and the resulting screen content *OUT*. Using the raw pixel data, the *SRC* image is computed, which results in the sum of all overlay window on top of DST and is analyzed as a whole, as opposed to other approaches that inspect every window separately. Afterwards, the SRC data is used to evaluate whether the user is being deceived by a potential attack.

Proposed solutions often differ in the reaction to detected events. Some solutions show a dialog that allows the user to remove an offending application [55], while other solutions only show a warning and let the user decide on how to proceed [46, 48]. In another case, the decision is given to the developers, who are given an interface to detect and react to overlays floating on top of their app [34]. Detection algorithm may differ depending on the machine learning model or data they are based on, which makes some detection more reliable than others.

D08. App hijacking detection: A more general approach to UI security is presented with the Android Window Integrity (AWI) model [48]. AWI describes a user session as a "chain of activities" where the current state of the UI is described by a tree of UI transitions. Combined with the concept of a "display owner", which is the app of the currently focused window on the screen, the model can detect suspicious actions including app hijacking and overlay windows. The authors also created *WindowGuard*, a Xposed module that implements AWI and alerts the user of the aforementioned actions by third-party apps. The analysis of the effectiveness of WindowGuard is limited, as it only includes own apps and known malware samples, but no real-world apps from the Play Store. Also, specific actions that are considered expected behavior might cause false positives with WindowGuard [56].

The principle of defining rules to differentiate normal and suspicious behavior is not a novelty in security fields such as intrusion detection. But the focus on Android UI transitions might be interesting to explore further, as the OS has predetermined ways to handle UI. By building a model that describes these processes it could make it easier to detect actions that might have unexpected or unwanted behavior.

D09. UI Sandboxing: Android makes use of various established protections to isolate processes, memory, and resources. In general, an app can not directly access other apps' resources. With regard to UI, apps can use intents to call other apps, but there is no way to control the called window. We already discussed accessibility services, which are the exception to this UI sandboxing, but malicious apps can also use vulnerabilities in apps and OS to get access to sensitive information. There are lots of approaches to minimize the threat of vulnerabilities in OS. Virtualization on Android is not a new concept, but is continuously being explored for different use cases [13]. Some issues with UI security are rooted in the possibility to access memory as a privileged user. This means that an attacker that gains elevated access to a device can effectively manipulate screen contents, read and write to internal memory and potentially control most of the hardware devices. There are already security considerations for the instance of a full OS or kernel takeover, which is where Trusted Environments come into play. But we found only few papers that consider sandboxing of UI components.

The work of [28] addresses part of this problem by isolating parts of the kernel and the OS into unprivileged containers. The mechanism called *Anception* uses virtual containers for code execution, while the actual read-only code is stored on a trusted host. This host also contains the UI and input stack to communicate with the application. The main idea is that an attacker that is able to compromise a container still has no possibility to access or manipulate the UI. All user inputs are considered sensitive, and therefore both the I/O and the virtual memory are handled in the trusted host.

Anception uses a Linux kernel module to implement its virtualization and logic. While the approach is valid on a conceptual level, the current implementation suffers from similar problems when an attacker manages to take over the kernel, which Anception can not protect against. The following section presents solutions that involve TEEs, which promise to protect against such strong attackers.

4.3 Trusted Execution Environment

Usually, device vendors implement their own security measures on top of Android. This is why vendors also develop their custom TEE implementations instead of using Google's Trusty TEE [20]. These are often packaged as binary blobs and offer different interfaces to interact with the Android OS. As the TEE OS usually runs in parallel to the *normal* user OS, a compromise of the latter does not affect the former. In order to protect UI interactions in the normal world, one can implement protections in the TEE as applications called *trustlets*. One major security property of TEEs is the size of the Trusted Computing Base (TCB), which represents the number of lines of code that needs to run in the trusted environment. The smaller the TCB, the fewer lines of code need to be executed in the TEE. From a statistical point of view, this means that fewer bugs are in the code. A bug or vulnerability in a TEE application may lead to full device takeover. A number of vulnerabilities have been found in the past that indicate that TEEs are not as secure as assumed [20, 30]. Therefore, secure implementations should be limited in size and features in order to reduce the TCB and the available attack surface. The following defenses move different procedures of UI management, from composition to rendering, to the TEE, in order to achieve different security goals.

D10. UI as a trusted app: When designing UIs for the TEE, it is necessary to identify a robust architecture for the implementation. As mentioned before, the bigger the TCB, the higher the risk of vulnerabilities. Therefore, moving the UI creation, management and rendering into the UI requires careful planning and well-defined interfaces. In addition, user input is also involved in the process, as UI without interaction is very limited, both in usefulness and

usability. Most solutions implement two modules that communicate over well-defined interfaces to bridge the gap between the normal world and the trusted world. The normal world app can be a kernel driver that simply proxies the data to the trusted world [19, 57], or the normal world handles all requests from apps and uses the driver in the trusted world to safely execute these requests [41]. For interacting with the user, either the touch driver is moved to the TEE [19] or the on-screen keyboard is implemented as a trusted app [57]. In order to draw to the screen, trusted apps require access to the underlying hardware devices. This is accomplished through device drivers that allow the OS to access the memory locations that are used for the screen, such as the frame buffer of the display device. Therefore, to move UI logic to the TEE, it is also required to move device drivers to the TEE, further increasing the TCB. Standard drivers of the Linux kernel and Android-specific drivers are often found to contain vulnerabilities that can be exploited, an example being the *Dirty COW* exploit [44].

Moving the whole UI stack to the TEE is a defense mechanism that requires lots of modifications to the TEE implementation and the underlying Android OS. From a security perspective, increasing the TCB increases the attack surface, which goes against the intentions of TEEs to provide a small and secure root of trust. Especially kernel drivers, which need to work with a multitude of different hardware configurations of various Android devices on the market, are critical in such a secure environment and might introduce unnecessary complexity in real-world implementations. Apart from practical difficulties, such solutions often cause a performance overhead to the system, as the context switch from normal world to trusted world requires time-consuming CPU operations. While for most solutions, a performance analysis is performed to calculate the average memory and execution overhead, there are no extensive evaluations of realistic usage patterns. For these evaluations, we suggest an analysis of performance and usability in the context of regular Android usage, such as operating different apps and using overlay windows. We expect that solutions that only trigger the secure code for specific views (such as TruZ-Droid [57]) outperform solutions that are implemented for general screen usage (e.g., SuiT [19], TrustUI [41]).

`D11.` **Single trusted UI components:** In order to keep down the TCB size, research has also explored solutions that only cover specific UI elements and limit themselves to selected components of the whole UI composition process. In most solutions, a secure UI is implemented as a trusted application in the Secure World, while a proxy driver in the normal user space is used to enable the interaction between the two worlds. The proxy driver forwards requests of the system to the secure UI component, where the logic to layout and draw UI elements is located. Examples for single trusted components are: a secure WebView element [39], a text renderer [50] or even parts of the original UI stack [58]. Such measures reduce the TCB considerably, but in some cases incur some performance penalties, especially if the switch between normal and secure world needs to be done several times per time frame. Also, when considering the attack surface, a compromised user-space or kernel-space component does not endanger the security of the whole TEE, even though even trusted apps with reduced code size can introduce severe vulnerabilities. A similar solution is presented by *VButton* [38], which includes a mode where the device itself does not generate any UI elements. Instead, the remote server composes the UI, sends a signed image, which is then directly displayed by a TEE-based graphics driver. The ARM TrustZone implements a secure mode for peripherals, which allows access to predefined peripherals (e.g., touchscreens, physical buttons, speakers) only to Secure World apps. *VButton* makes use of this to temporarily declare the touchscreen as "secure" and detect if the user touches the secure UI button. The paper focuses on the interaction by the user, so the scope of the solution is limited to single button UIs, confirmation dialogs and similar use cases.

`D12.` **Dedicated LED indicator:** One issue that many researchers point out in the field of UI security is that the user does not know if what is currently shown on a screen is genuine or not. Therefore, some solutions introduce the concept of "secure LED" indicators, as some devices on the market already include the necessary hardware. While the access to peripherals and hardware components is usually handled by the normal operating system, the ARM TrustZone Protection Controller (TZPC) allows memory regions or hardware peripherals to be marked as "secure peripherals" which are not accessible for the normal world as long as they are tagged [12]. LED indicators can be integrated for various use cases. If the color is changeable, you can use the color to give the user a hint on which background color should be visible (in order to detect overlay attacks) [39]. This is more effective with multiple LEDs, which are not widely available on modern devices. A more practical approach is informing the user that the current screen is secure when the LED is active and only allowing access to it from the trusted world [19, 58]. From a security perspective, this might be a valid method to indicate a secure context for the user, but in practice breaks the main functionality of the current Android system, namely having apps access the LED for showing notifications or as a battery indicator.

4.4 Additional hardware

In order to freely experiment with ARM functionality, research also experiment with external hardware peripherals to test different security configurations. Most solutions in this field are conceptional and make use of FPGAs for demonstrations.

`D13.` **Physical separation:** One goal of UI security is providing the user with a screen that can not be manipulated. This is often called a secure display and refers to the ability to gain and retain control of a screen in a way that is not easily manipulated. On the other side, manufacturers and developers also want to have flexibility in creating their own screen contents. One main pillar of trusted displays is the separation of hardware to have security-sensitive operations separated from regular interactions with a system. The separation can be logically, e.g., having a separate GPU kernel running on the same physical graphic card [60] or physically by providing two processor units in one device that do not share any memory regions and compute data independently of each other. For this, an additional coprocessor can be used that processes and manipulates the screen contents before the physical display receives the data [18]. This trusted CPU is located between the screen output of Android and an LCD screen and can overlay the output with its own content. The untrusted CPU on the Android device can send a number of requests to the trusted CPU in order

to display trusted content, such as a password confirmation screen. If the Android device is controlled by an attacker, the attacker can block all these requests or simply force a secure screen. This is why the secure CPU unit still needs to have an indicator to signal the user whenever a screen is "secured" or not.

Coprocessors for secure operations such as file encryption and cryptography are already used in some of today's smart devices, for example Google's Titan M [45] or Apple's T2 Security Chip [10]. As discussed in subsection 4.1, the attack surface with this defense method is composed of the interfaces between insecure OS and coprocessor. It is therefore not a standalone solution to the UI trust problem, but only one aspect that needs to integrate other defenses in order to be effective and practical.

		Defense Level			
	Threat Model	**USR**	**OS**	**TEE**	**Feasible**
D01	USR	●	●	○	●
D02	USR	○	●	○	●[1]
D03	USR	○	●	○	●[1]
D04	USR	○	●	○	●[1]
D05	OS	●	○	●	●[1]
D06	USR	●	●	○	○
D07	USR	○	●	○	○
D08	USR	○	●	○	●
D09	OS	○	●	○	○
D10	OS	○	●	●	●[1]
D11	OS	○	●	●	◐
D12	OS	○	○	●	○
D13	TEE	○	○	●[2]	●[1]

Table 1: Comparison of defenses in terms of threat model, defense (access) level and feasibility. Some defenses are (1) partly or completely implemented in stock Android, or (2) supported through a separate hardware component. The feasibility defines the potential for adoption of a specific implementation through Android.

RQ 2: Defenses against UI security flaws are mostly implemented on the OS level and in kernel space, with few exceptions making use of hardware-based TEE solutions. While some defenses are easily implemented in the OS, most current solutions can not be deployed on real-world devices without the support of vendors. Some solutions do not consider real-world use cases and break existing functionality or introduce severe performance penalties that decrease the chance of adoption by the vendors. Regarding trends, research tends to shift the implementation details to isolated containers in kernel space in order to stop attackers from taking over the whole system after a compromise of single components. While vendors introduce physically separated co-processors, core issues in the interfaces between secure and insecure mode remain.

5 CHALLENGES

In our systematization, we found several developments that resolve existing problems. Still, we see a number of problems with the presented solutions and hope to spark a debate on how to improve research in this niche field.

5.1 The Root of Trust

The most invasive solutions in published works are trying to fix the problem of a missing Root of Trust (RoT) in mobile devices. In current mobile devices, the RoT is included during manufacturing and tied to a specific device state. For example, Android devices ship with a signed version of the Android OS, signed by a private key known only to the vendor. A corresponding public key is stored inside a protected read-only part of the device [3]. In contrast, Apple uses their T2 security chip inside their iPhone and Mac products to establish a RoT [10]. The same chip is used for Secure Boot, cryptography features and more [33]. Trusted hardware components such as the T2 chip still rely on software interfaces to deliver a service to the consumer, which may always introduce bugs and in some cases lead to exploitable vulnerabilities, like the *checkra1n* exploit for Apple's T2 chips [43]. For UI security, the presence of a RoT is unfortunately not enough to ensure secure UIs or any form of secure input. Even assuming that these components were safe, this alone does not lead to a trusted path, as there are a number of untrusted components in the path between device and remote server, such as the networking stack, peripherals and more. Apart from remote and third-party components, most modern consumer devices are composed of some software and hardware pieces created by different manufacturers, which adds complexity to the whole security architecture of a single device. Therefore, we argue that RoTs and trusted paths in mobile devices are a lacking field of research which require more attention and innovative solutions.

5.2 Balance of features and security

Security is never the main purpose of any system, and this is especially true for modern mobile devices. Not every security feature is implemented to improve device security. An example is *Widevine*, a TEE app which implements digital rights management (DRM) for use in third-party apps. Widevine may protect the intellectual property of some companies, but does not actively contribute to the security of the device. On the contrary: in the past Widevine introduced a critical vulnerability that allowed attackers to escalate privileges [23]. On Apple devices, the T2 chip is both used to secure the boot process and to enable features like biometric authentication (TouchID, FaceID), so a vulnerability in the biometric authentication code may lead to a compromise of device security. These are not isolated cases, and TEE researchers have already assessed that the Trusted Computing Base consisting of trusted firmware, OS and apps, is excessively large on mobile devices [20].

The other way around, security-related defenses like Android's Protected Confirmation are not flexible enough and allow only limited integration into the design freedom that Android offers, which may seem like a limitation to developers and users. But more flexible defenses come with increased TCBs which again introduce more bugs and more potential vulnerabilities. Therefore, we argue that TEEs should be focusing on mitigating attacks and

vulnerabilities of the device. While there is a trend to use secure coprocessors inside devices, the actual problem of using these to implement marketable user features is still present.

5.3 Lack of concepts for user training and interaction

The analyzed literature focuses strongly on technical implementations of security. In doing so, authors make lots of assumptions on how users may interact with the system. Whenever a system requires the user to double-check, or when a new indicator (e.g., LEDs or warning messages) is introduced, it is assumed that users understand what they are supposed to do. In this context, there are very few considerations and solutions against spoofed indicators. Examples are unprivileged third-party apps that can easily create spoofed "warning" dialogs that may be used to confuse the user or might lead to user habituation [36]. The same can be done for security indicators like images, texts and any regular UI-based security feature.

A related problem is the lack of separation of security features and "regular" device usage, and the resulting assumptions about user interaction with the system. On Android phones, an LED built into the phone is often used for notifications, for showing the charging status of the battery and by different apps to get the user's attention. Often LEDs can also have different colors which convey different meanings and contexts. Adding a security-critical meaning on top of all these existing use-cases (as described by **D12**) may increase the confusion for the user and lead to ambiguity about the security of the current interaction.

As a positive example, Apple introduced a special home button with the iPhone 5S which includes Touch ID, a fingerprint reader underneath the home button. The new feature was marketed and presented primarily as a security feature, and is only used for user authentication. When an app tries to use Touch ID, a system message controlled by the OS asks the user to present a finger to scan. There is not much user confusion on how the feature works and user studies identified that key factors of TouchID usage are "its usability and perceived security" [21].

RQ 3: Trusted path concepts with a focus on UI are lacking in research.

- A focus should be put on separation of concerns.
- The lack of consideration for end users in the security design needs to be addressed.
- The shift to dedicated secure hardware needs to be further explored, and open source designs may be helpful for adoption by the established vendors.

6 CONCLUSION

This paper presents a study of current UI security research. A summary of issues and defenses in regard to the attacker model and the required depth of implementation for the countermeasures are presented in Table 1.

We found two main directions in research: *Overlay / Context Hiding* and *UI Control* (see Table 2). The first one describes clickjacking attacks, deception methods and some form of Denial-of-Service that allows attackers to cover up the whole screen. Defenses for this focus on overlay detection, filtering methods and including user decision when suspicious behavior is found. The second topic describes issues that allow attacks that take over (parts of) the screen, read the screen contents and even act on behalf of the user. Proposed solutions try to isolate the UI from the system, either through existing methods or by using hardware-backed features, such as TEEs. A lot of research papers focus on single design issues in the Android OS and propose solutions that could be adopted by the software vendors. There are considerations to deploy hardware measures to support single solutions, such as using dedicated hardware or using existing solutions such as TrustZone.

There are a few shortcomings that we want to highlight, which concludes our contribution and puts into focus what is really important. While technical solutions are presented, there is no consideration for the end-user perspective, as many solutions rely on the user for decisions on how to proceed, while regular users of mobile devices may have no concept of secure and insecure modes of operation (compare to **D12**). In our systematization, we also noticed a shift from pure OS-level solutions to HW-supported and TEE-supported solutions. This also moves the trust base from the phone to dedicated hardware modules, therefore transitioning to a concept similar to Trusted Platform Modules on regular personal computers. With regard to TEEs, special attention is needed as growing TCBs increase the chance to introduce software vulnerabilities affecting millions of devices, and TEE implementations can no longer be considered as secure as advertised. The shift to spatial separation (as described by **D13**) does not improve security by itself, as most issues are design decisions done on an OS and user level. We expect to see more research in the direction of dedicated security hardware, as we believe that open security architectures for mobile devices are lacking and may contribute significantly to the security of consumer devices.

	Overlays / Context Hiding	UI control
Description	clickjacking, DoS, deception	(full) takeover, privacy leak
Issue(s)	I01 – I04	I05 – I06
Defenses	D01 – D04, D06 – D08	D05, D09 – D11
Threat model	USR	OS

Table 2: Overview of issues in research, suggested defenses and assumed threat model (see subsection 2.3)

ACKNOWLEDGMENTS

We thank our anonymous reviewers for significantly improving this paper through their insights. This research was supported by the German Federal Ministry of Education and Research (BMBF) as part of the Software Campus project (Förderkennzeichen: 01IS17045).

REFERENCES

[1] Android. 2021. Android Releases. https://developer.android.com/about/versions. Last access: 2021-08-09.
[2] Android. 2021. Autofill framework. https://developer.android.com/guide/topics/text/autofill. Last access: 2020-08-13.
[3] Android. 2021. Device State. https://source.android.com/security/verifiedboot/device-state. Accessed: 2020-08-13.
[4] Android. 2021. MotionEvent - Android Developers. https://developer.android.com/reference/android/view/MotionEvent. Accessed: 2020-08-13.

[5] Android. 2021. Multi-Window Support. https://developer.android.com/guide/topics/ui/multi-window. Last access: 2020-08-13.
[6] Android. 2021. Protected Confirmation Implementation. https://source.android.com/security/protected-confirmation/implementation. Accessed: 2020-08-13.
[7] Android. 2021. Trusty TEE. https://source.android.com/security/trusty. Accessed: 2020-08-13.
[8] Android. 2021. Verifying hardware-backed key pairs with Key Attestation. https://developer.android.com/training/articles/security-key-attestation. Accessed: 2020-08-13.
[9] Simone Aonzo, Alessio Merlo, Giulio Tavella, and Yanick Fratantonio. 2018. Phishing Attacks on Modern Android. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 1788–1801. https://doi.org/10.1145/3243734.3243778
[10] Apple Inc. 2018. *Apple T2 Security Chip - Security Overview*. Technical Report. Apple Inc.
[11] Apple Inc. 2019. *iOS Security - iOS 12.3 - May 2019*. Technical Report. Apple Inc.
[12] ARM. 2021. About the TrustZone Protection Controller - ARM Developer. https://developer.arm.com/documentation/dto0015/a/about-the-trustzone-protection-controller. Accessed: 2020-08-13.
[13] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. 2015. Boxify: Full-fledged App Sandboxing for Stock Android. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 691–706. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/backes
[14] 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 *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 931–948. https://doi.org/10.1109/SP.2015.62
[15] David Bisson. 2020. *New Android Malware Channels Malicious Activity Through Accessibility Services*. https://securityintelligence.com/news/new-android-malware-channels-malicious-activity-through-accessibility-services/ Last access: 2021-11-01.
[16] C. Bormann and P. Hoffman. 2021. RFC 7049. https://tools.ietf.org/html/rfc7049. Last access: 2020-08-13.
[17] Davide Bove and Anatoli Kalysch. 2019. In pursuit of a secure UI: The cycle of breaking and fixing Android's UI. *it Inf. Technol.* 61, 2-3 (2019), 147–156. https://doi.org/10.1515/itit-2018-0023
[18] Anthony Brandon and Michael Trimarchi. 2017. Trusted display and input using screen overlays. In *International Conference on ReConFigurable Computing and FPGAs, ReConFig 2017, Cancun, Mexico, December 4-6, 2017*. 1–6. https://doi.org/10.1109/RECONFIG.2017.8279826
[19] Yang Cai, Yuewu Wang, Lingguang Lei, Quan Zhou, and Jun Li. 2019. SuiT: Secure User Interface Based on TrustZone. In *2019 IEEE International Conference on Communications, ICC 2019, Shanghai, China, May 20-24, 2019*. 1–7. https://doi.org/10.1109/ICC.2019.8761616
[20] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1416–1432. https://doi.org/10.1109/SP40000.2020.00061
[21] Ivan Cherapau, Ildar Muslukhov, Nalin Asanka, and Konstantin Beznosov. 2015. On the Impact of Touch ID on iPhone Passcodes. In *Eleventh Symposium On Usable Privacy and Security, SOUPS 2015, Ottawa, Canada, July 22-24, 2015*, Lorrie Faith Cranor, Robert Biddle, and Sunny Consolvo (Eds.). USENIX Association, 257–276. https://www.usenix.org/conference/soups2015/proceedings/presentation/cherapau
[22] Janis Danisevskis. 2021. Teeui layout and rendering. https://android.googlesource.com/platform/system/teeui/+/5821a43f983a216b49e8abb0f2b5c9998ea9fcd. Accessed: 2020-08-13.
[23] NIST National Vulnerability Database. 2016. CVE-2015-6639. https://nvd.nist.gov/vuln/detail/CVE-2015-6639. Last access: 2021-02-03.
[24] Aritra Dhar, Enis Ulqinaku, Kari Kostiainen, and Srdjan Capkun. 2020. ProtectIOn: Root-of-Trust for IO in Compromised Platforms. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/protection-root-of-trust-for-io-in-compromised-platforms/
[25] Wenrui Diao, Yue Zhang, Li Zhang, Zhou Li, Fenghao Xu, Xiaorui Pan, Xiangyu Liu, Jian Weng, Kehuan Zhang, and XiaoFeng Wang. 2019. Kindness is a Risky Business: On the Usage of the Accessibility APIs in Android. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*. USENIX Association, 261–275. https://www.usenix.org/conference/raid2019/presentation/diao
[26] Saba Eskandarian, Jonathan Cogan, Sawyer Birnbaum, Peh Chang Wei Brandon, Dillon Franke, Forest Fraser, Gaspar Garcia Jr., Eric Gong, Hung T. Nguyen, Taresh K. Sethi, Vishal Subbiah, Michael Backes, Giancarlo Pellegrino, and Dan Boneh. 2019. Fidelius: Protecting User Secrets from Compromised Browsers. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 264–280. https://doi.org/10.1109/SP.2019.00036
[27] Facebook. 2021. Facebook Messenger. https://www.messenger.com/. Last access: 2020-08-13.
[28] Earlence Fernandes, Ajit Aluri, Alexander Crowell, and Atul Prakash. 2015. Decomposable Trust for Android Applications. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015, Rio de Janeiro, Brazil, June 22-25, 2015*. IEEE Computer Society, 343–354. https://doi.org/10.1109/DSN.2015.15
[29] Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J. Alex Halderman, Zhuoqing Morley Mao, and Atul Prakash. 2016. Android UI Deception Revisited: Attacks and Defenses. In *Financial Cryptography and Data Security - 20th International Conference, FC 2016, Christ Church, Barbados, February 22-26, 2016, Revised Selected Papers*. 41–59. https://doi.org/10.1007/978-3-662-54970-4_3
[30] Fabian Fleischer, Marcel Busch, and Phillip Kuhrt. 2020. Memory corruption attacks within Android TEEs: a case study based on OP-TEE. In *ARES 2020: The 15th International Conference on Availability, Reliability and Security, Virtual Event, Ireland, August 25-28, 2020*, Melanie Volkamer and Christian Wressnegger (Eds.). ACM, 53:1–53:9. https://doi.org/10.1145/3407023.3407072
[31] 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 *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. 1041–1057. https://doi.org/10.1109/SP.2017.39
[32] Liangyi Gong, Zhenhua Li, Hongyi Wang, Hao Lin, Xiaobo Ma, and Yunhao Liu. 2021. Overlay-based Android Malware Detection at Market Scales: Systematically Adapting to the New Technological Landscape. *IEEE Transactions on Mobile Computing* (2021), 1–1. https://doi.org/10.1109/TMC.2021.3079433
[33] Ivan Krstić. 2019. Behind the scenes of iOS and Mac Security. https://www.blackhat.com/us-19/briefings/schedule/index.html#behind-the-scenes-of-ios-and-mac-security-17220.
[34] Abeer Al Jarrah and Mohamed Shehab. 2016. Maintaining User Interface Integrity on Android. In *40th IEEE Annual Computer Software and Applications Conference, COMPSAC 2016, Atlanta, GA, USA, June 10-14, 2016*. IEEE Computer Society, 449–458. https://doi.org/10.1109/COMPSAC.2016.150
[35] Anatoli Kalysch, Davide Bove, and Tilo Müller. 2018. How Android's UI Security is Undermined by Accessibility. In *Proceedings of the 2nd Reversing and Offensive-oriented Trends Symposium*. 1–10.
[36] Brock Kirwan, Bonnie Anderson, David Eargle, Jeffrey Jenkins, and Anthony Vance. 2020. Using fMRI to Measure Stimulus Generalization of Software Notification to Security Warnings. In *Information Systems and Neuroscience*, Fred D. Davis, René Riedl, Jan vom Brocke, Pierre-Majorique Léger, Adriane Randolph, and Thomas Fischer (Eds.). Springer International Publishing, Cham, 93–99.
[37] Joshua Kraunelis, Yinjie Chen, Zhen Ling, Xinwen Fu, and Wei Zhao. 2015. On Malware Leveraging the Android Accessibility Framework. *EAI Endorsed Trans. Ubiquitous Environ.* 1, 4 (2015), e1. https://doi.org/10.4108/ue.1.4.e1
[38] 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 2018, Munich, Germany, June 10-15, 2018*, Jörg Ott, Falko Dressler, Stefan Saroiu, and Prabal Dutta (Eds.). ACM, 28–40. https://doi.org/10.1145/3210240.3210330
[39] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. 2014. Building trusted path on untrusted device drivers for mobile devices. In *Asia-Pacific Workshop on Systems, APSys'14, Beijing, China, June 25-26, 2014*. 8:1–8:7. https://doi.org/10.1145/2637166.2637225
[40] Hongliang Liang, Mingyu Li, Yixiu Chen, Lin Jiang, Zhuosi Xie, and Tianqi Yang. 2020. Establishing Trusted I/O Paths for SGX Client Systems With Aurora. *IEEE Trans. Inf. Forensics Secur.* 15 (2020), 1589–1600. https://doi.org/10.1109/TIFS.2019.2945621
[41] Dongtao Liu and Landon P. Cox. 2014. VeriUI: attested login for mobile devices. In *15th Workshop on Mobile Computing Systems and Applications, HotMobile '14, Santa Barbara, CA, USA, February 26-27, 2014*. 7:1–7:6. https://doi.org/10.1145/2565585.2565591
[42] Tongbo Luo, Xing Jin, Ajai Ananthanarayanan, and Wenliang Du. 2012. Touchjacking Attacks on Web in Android, iOS, and Windows Phone. In *Foundations and Practice of Security - 5th International Symposium, FPS 2012, Montreal, QC, Canada, October 25-26, 2012, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7743)*, Joaquín García-Alfaro, Frédéric Cuppens, Nora Cuppens-Boulahia, Ali Miri, and Nadia Tawbi (Eds.). Springer, 227–243. https://doi.org/10.1007/978-3-642-37119-6_15
[43] Rick. Mark, Aun-Ali Zaidi, h0m3us3r, and mrarm. 2020. Jailbreaking the T2 with checkra1n. https://blog.t8012.dev/t2-checkra1n-guide/. Last access: 2021-02-03.
[44] Huasong Meng, Vrizlynn L. L. Thing, Yao Cheng, Zhongmin Dai, and Li Zhang. 2018. A survey of Android exploits in the wild. *Comput. Secur.* 76 (2018), 71–91. https://doi.org/10.1016/j.cose.2018.02.019
[45] Nagendra Modadugu and Bill Richardson. 2021. Building a Titan: Better security through a tiny chip. https://security.googleblog.com/2018/10/building-titan-better-security-through.html. Accessed: 2020-08-13.

[46] 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 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018.* 1120–1136. https://doi.org/10.1145/3243734.3243785

[47] Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. 2015. An investigation of the Android/Badaccents malware which exploits a new Android tapjacking attack. *Technical report, Technische Universitt Darmstadt* (2015).

[48] Chuangang Ren, Peng Liu, and Sencun Zhu. 2017. WindowGuard: Systematic Protection of GUI Security in Android. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017.* https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/windowguard-systematic-protection-gui-security-android/

[49] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. 2015. Towards Discovering and Understanding Task Hijacking in Android. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.* 945–959. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ren-chuangang

[50] 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'17, Niagara Falls, NY, USA, June 19-23, 2017*, Tanzeem Choudhury, Steven Y. Ko, Andrew Campbell, and Deepak Ganesan (Eds.). ACM, 197–210. https://doi.org/10.1145/3081333.3081346

[51] M. Angela Sasse, Matthew Smith, Cormac Herley, Heather Lipford, and Kami Vaniea. 2016. Debunking Security-Usability Tradeoff Myths. *IEEE Secur. Priv.* 14, 5 (2016), 33–39. https://doi.org/10.1109/MSP.2016.110

[52] UrbanDroid. 2021. Twilight. https://twilight.urbandroid.org/. Last access: 2020-08-13.

[53] Samuel Weiser and Mario Werner. 2017. SGXIO: Generic Trusted I/O Path for Intel SGX. In *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*, Gail-Joon Ahn, Alexander Pretschner, and Gabriel Ghinita (Eds.). ACM, 261–268. https://doi.org/10.1145/3029806.3029822

[54] Israel Wernik and Bohdan Melnykov. 2021. *PixStealer: a new wave of Android banking Trojans abusing Accessibility Services.* https://research.checkpoint.com/2021/pixstealer-a-new-wave-of-android-banking-trojans-abusing-accessibility-services/ Last access: 2021-11-01.

[55] Longfei Wu, Benjamin Brandt, Xiaojiang Du, and Bo Ji. 2016. Analysis of clickjacking attacks and an effective defense scheme for Android devices. In *2016 IEEE Conference on Communications and Network Security, CNS 2016, Philadelphia, PA, USA, October 17-19, 2016.* IEEE, 55–63. https://doi.org/10.1109/CNS.2016.7860470

[56] Fei Yan, Yijia Li, and Liqiang Zhang. 2018. ActivityShielder: An Activity Hijacking Defense Scheme for Android Devices. In *27th International Conference on Computer Communication and Networks, ICCCN 2018, Hangzhou, China, July 30 - August 2, 2018.* IEEE, 1–9. https://doi.org/10.1109/ICCCN.2018.8487367

[57] 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 2018, Munich, Germany, June 10-15, 2018*, Jörg Ott, Falko Dressler, Stefan Saroiu, and Prabal Dutta (Eds.). ACM, 14–27. https://doi.org/10.1145/3210240.3210338

[58] 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 Ninth ACM Conference on Data and Application Security and Privacy, CODASPY 2019, Richardson, TX, USA, March 25-27, 2019*, Gail-Joon Ahn, Bhavani M. Thuraisingham, Murat Kantarcioglu, and Ram Krishnan (Eds.). ACM, 1–12. https://doi.org/10.1145/3292006.3300035

[59] Lingyun Ying, Yao Cheng, Yemian Lu, Yacong Gu, Purui Su, and Dengguo Feng. 2016. Attacks and Defence on Android Free Floating Windows. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang (Eds.). ACM, 759–770. https://doi.org/10.1145/2897845.2897897

[60] Miao Yu, Virgil D. Gligor, and Zongwei Zhou. 2015. Trusted Display on Untrusted Commodity Platforms. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015.* 989–1003. https://doi.org/10.1145/2810103.2813719

A EXTENDED BACKGROUND

Modern processors implement different privilege levels, such as the protection ring model found in x86 architecture. The ARM architecture has a similar model with comparable levels: user (PL0), operating system (PL1), hypervisor (PL2) and secure monitor (PL3), with PL0 being the least-privileged and PL3 being the highest-privileged level. The ARM TrustZone is orthogonal to these levels, which means that the Secure World is also subdivided into these levels (see Figure A.1).

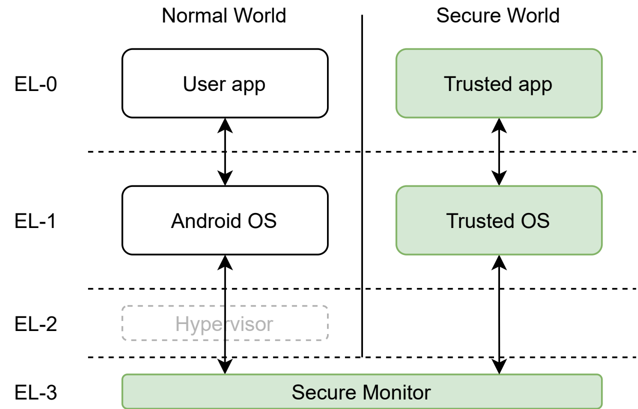

Figure A.1: Software architecture of a TrustZone-assisted Android device.

B ANDROID VERSION HISTORY

This work can only provide a snapshot of the current changes in Android. The most recent and planned changes are summarized in Table B.1. The data is retrieved from the official documentation [1].

Version	Description
1.0	Introduced Toast messages
1.6	Introduced Accessibility Services
2.3	Introduced Touch Filtering; Introduced *Obscured* flag
4.3	Added TEE support for KeyStore
6.0	Added special permission for overlays (granted through Play Store or by user)
7.0	Added permission dialog request for overlays
8.0	Restricted overlay usage and changed screen priority
9	Added Protected Confirmation; Added support for secure coprocessors (StrongBox)
10	Restricted access to screen contents; Enabled *Partially Obscured* flag for developers
11	Deprecated custom Toasts; Granted overlay permission when recording screen
12	Added blocking and detection of *Obscured* touch events

Table B.1: Evolution of Android UI security with relevant versions.

C BEYOND THE MOBILE DOMAIN

In order to validate the research gaps found in our research, we looked at other domains of UI security. Are the identified problems and issues from our systematization unique to the mobile world,

or have other branches of security research already figured out solutions for them? To answer this question, it makes sense to take in consideration how other platforms and environments solve this problem. Therefore, we also looked at work which solved similar UI issues in related domains.

A basic component of UI and the related IO security field is the existence of trusted elements and trusted paths. These are built upon one or several Root of Trusts (RoT). A root of trust is a source of security in a system, which is assumed to be safe against any serious attacker. In most cases, the RoT is built around a secret owned by a vendor. In recent devices the TEE is an inherent part of the RoT and many security features (e.g., full-disk encryption, cryptography services, secure execution) are built on the premise of the TEE being secure.

In this context, we found a number of exceptional papers that enable trusted paths using TEE, but which do not translate well into the mobile domain. SGXIO [53] implements generic trusted paths for Intel SGX enclaves, but relies on a hypervisor and a Trusted Platform Module (TPM) which are common on desktops and servers but not on mobile devices. With similar methods Aurora [40] makes use of proprietary Intel processor features for achieving the same goal.

In regard to additional hardware, we found several papers that employ external systems to improve IO security. ProtectIOn [24] does not use existing TEE solutions for x86, but adds a device between host and peripherals. A keyboard and a screen, an input and an output device respectively, are connected to the proxy device that captures the interaction, such that the keyboard and mouse are not directly connected to the host computer. Keyboard input is encrypted and sent to a remote server by using the untrusted host's network. In the other direction, output generated by the server is only decrypted inside the proxy device and then sent to the screen, displayed as an overlay over regular screen contents. The solution fits into the previous category of subsection 4.4, but in general is difficult to translate to the majority of mobile devices, since keyboard and screen are part of the same peripheral (a touchscreen). It may be a potential solution for users with convertible tablets that make use of an external keyboard and an external screen, but we argue this scenario is of limited practicality to the majority of use cases. A similar solution is presented by Fidelius [26], which uses a combination of one proxy device per peripheral and an Intel SGX enclave on the host.