# SchrodinText: Strong Protection of Sensitive Textual Content of Mobile Applications

Ardalan Amiri Sani
Computer Science Department
University of California, Irvine
ardalan@uci.edu

## Abstract

Many mobile applications deliver and show *sensitive and private textual content* to users including messages, social network posts, account information, and verification codes. All such textual content must only be displayed to the user but must be strongly protected from unauthorized access in the device. Unfortunately, this is not the case in mobile devices today: malware that can compromise the operating system, e.g., gain root or kernel privileges, can easily access textual content of other applications.

In this paper, we present SchrodinText, a system solution for strongly protecting the confidentiality of application's selected UI textual content from a fully compromised operating system. SchrodinText leverages a novel security monitor based on two hardware features on modern ARM processors: virtualization hardware and TrustZone. Our key contribution is a set of novel techniques that allow the operating system to perform the text rendering without needing access to the text itself, hence minimizing the Trusted Computing Base (TCB). These techniques, collectively called *oblivious rendering*, enable the operating system to rasterize and lay out all the characters without access to the text; the monitor only *resolves* the right character glyphs onto the framebuffer *observed* by the user and protects them from the operating system, e.g., against DMA attacks. We present our prototype using an ARM Juno development board and Android operating system. We show that SchrodinText incurs noticeable overhead but that its performance is usable.

> *"If one has left this entire system to itself for an hour, one would say that the cat still lives if meanwhile no atom has decayed. The first atomic decay would have poisoned it. The psi-function of the entire system would express this by having in it the* ***living and dead cat*** *(pardon the expression) mixed or smeared out in equal parts. It is typical of these cases that an indeterminacy originally restricted to the atomic domain becomes transformed into macroscopic indeterminacy, which can then be* ***resolved*** *by direct* ***observation****."*
>
> *- Erwin Schrödinger*[1]

[1]https://en.wikipedia.org/wiki/Schrodingers_cat



*MobiSys'17, June 19-23, 2017, Niagara Falls, NY, USA*



ISBN 978-1-4503-4928-4/17/06…$15.00

DOI: http://dx.doi.org/10.1145/3081333.3081346

## Keywords

Text protection; Mobile devices; UI safety; TrustZone; Virtualization

## 1. INTRODUCTION

There are millions of mobile applications today. The textual content that many of these applications show to the user in their UI can contain extremely sensitive and private information, such as social security number, bank account information, private messages, passwords (in a password vault), and verification codes used for two-factor authentication. *Such content must only be displayed to the user but must be otherwise protected against unauthorized access by malware.* Indeed, Zhou et al. found 644 malware samples (in 27 families) that harvest "user's information, including user accounts and short messages stored on the phones" [54]. Application developers attempt to defeat this by encrypting the raw text in their backend server and send the ciphertext to the mobile application. However, the application needs to decrypt the ciphertext before passing it to the operating system for rendering and showing to the user. Therefore, malware that compromises the operating system (or even just its graphics stack) can access the content.

This is a growing concern as mobile operating systems are increasingly large and unreliable. Malware hiding inside ordinary unprivileged applications can easily compromise the operating system and gain root or kernel privileges [50, 53]. Such malware can then attempt to extract sensitive content from other applications using various methods. For example on Android, these methods include but are not limited to the following. Malware with root privilege can (*i*) take a screenshot of the device UI, which might contain the aforementioned sensitive text or (*ii*) replace Android's UI rendering libraries; the compromised library can then leak the textual content of victim applications. Or malware with kernel privileges can (*i*) access the application's memory in order to read the plaintext after decryption or (*ii*) read the texture buffers allocated by the graphics stack to hold the character glyphs used in the text.

Unfortunately, as noted by Checkoway et al. [23], it is extremely difficult to fully protect an application from a compromised operating system. Yet, we ask ourselves: *can we at least protect the confidentiality of selected sensitive textual content of the application's UI in such an environment?* We care about protecting selected textual content since most, if not all, sensitive information delivered to the user via apps is in textual format.

In this paper, we present a system solution, called *Schrod-*

*inText*, designed to meet this goal. Our solution leverages novel hardware features in ARM processors to create a security monitor for showing and protecting the text. The monitor leverages both virtualization and TrustZone hardware available on modern ARM processor. The monitor is more privileged than the operating system and is used to protect the selected textual content shown on the display.

In SchrodinText, the application's backend server encrypts the text with a key only available to the monitor, hence protecting it from the operating system. However, this raises an important challenge: the text needs to be rendered by the graphics stack in the operating system. The final pixels displayed to the user depend on the font type, size, and color chosen by the app developer, all of which are available and understandable by the operating system, and not the monitor. The graphics stack includes several libraries and frameworks as well as kernel device drivers and is one of the largest and most complicated components of the operating system. Therefore, how can the monitor display the textual content to the user given that the graphics stack is implemented by the operating system?

At first glance, it seems like a feasible solution to this problem is to move the graphics stack to the monitor. However, such an approach would significantly bloat the Trusted Computing Base (TCB) of the monitor. A large TCB would make the monitor vulnerable to attacks itself hence defeating the original purpose. An alternative approach might then be to support a minimal and simplified version of the graphics stack in the monitor, in order to keep the TCB small. Unfortunately, with this approach, the content rendered by the monitor would not be visually well-integrated with the rest of the content, which are rendered by the complete graphics stack in the operating system. Moreover, such a solution would not be able to easily benefit from updates to the operating system graphics stack, which, for example, might introduce new fonts and effects from untrusted sources.

In SchrodinText, we introduce an alternative solution that addresses all of these shortcomings: *oblivious rendering*. With this solution, SchrodinText implements the text rendering fully in the operating system without access to the text itself, hence resulting in visually well-integrated content while minimizing the TCB of the monitor. The key idea is to have the operating system rasterize all the possible character glyphs given the attributes of the text selected by the developer, e.g., font type, size, and color. The resultant glyphs are then shared with the monitor using a simple and low-level memory API (referred to as glyph-book API hereafter). The monitor then decrypts the ciphertext and uses that to *resolve* the right glyphs onto the framebuffer *observed* by the user[2]. With SchrodinText, *the protected text will be visible to the user on the display but not available to the operating system, e.g., in a screenshot.*

We address two important challenges in SchrodinText. First, the operating system needs to perform the layout of text (i.e., determining the location on the UI to place each character), which not only depends on the relative size of the font to the size of text UI widget, it also depends on the text itself as different characters have varying widths. To address this challenge, we choose to limit SchrodinText to *monospaced fonts*, in which all characters have a fixed width. In addition, we introduce a technique for determining the line-breaks without access to the text, called *oblivious line-breaking*. These two techniques allow the operating system to perform the layout in full, only needing access to the *number of characters* in the text, and not the text itself.

Second, once character glyphs are rasterized and their locations are determined, they must be composited on top of other layers in the framebuffer and displayed to the user. Android on modern mobile devices uses the GPU for compositing. Exposing the resolved glyphs to the GPU makes them vulnerable to attacks by the operating system since the GPU is programmed by the operating system. We use two techniques, namely *multi-view pages* and *two-stage compositing*, to securely perform the compositing while protecting the resolved glyphs. Collectively, these two techniques allow us to use the operating system and GPU for compositing of non-protected textures and use the monitor (but not the GPU) for compositing the protected text glyphs, all the while protecting them from the operating system and other DMA-capable devices.

It is important to note that SchrodinText protects the *output text* of the applications but not the *input text*. That is, it protects the textual content that is shown to the user, such as received messages, bank account information, health records, and verification codes in two-factor authentication. It, however, does not protect the text input by the user, including outgoing messages and typed passwords. Protecting the input text also requires protecting the input stack in the operating system, which is out of the scope of this work.

We have designed SchrodinText with ease of use for developers in mind. More specifically, we provide a simple UI widget, called `SchrodinTextView`, which is a modified version of `TextView` used in Android to embed text in application's UI. The developer can simply embed this widget in the UI, very similar to existing widgets. The cloud backend sends the ciphertext to the application, which then binds it to the widget by a call to a `setCiphertext()` function, also passing a handle to the key needed to decrypt the text (the key is known to the monitor but not the operating system). SchrodinText takes care of rendering the text, showing it to the user, and protecting it from the operating system. To demonstrate the practicality of this API, we incorporate it in a messaging application to show and strongly protect the received messages.

SchrodinText does not require any changes to the mobile device hardware; it however requires its monitor to be deployed in the hypervisor and TrustZone secure world, and requires modifications to the operating system kernel and user space rendering libraries. Hence, we envision SchrodinText to be deployed by mobile device vendors, e.g., Google, Samsung, HTC, and Apple.

We implement SchrodinText for Android running on an ARM Juno development board, the only board that allows non-vendors to program the hypervisor and TrustZone secure world, to the best of our knowledge. We show that SchrodinText incurs modest memory and performance overheads. More specifically, depending on the font size, it can use an additional 150 kB to 4.8 MB for storing the glyph-books. Also, depending on the size of the framebuffer and the number and size of the glyphs, it can use an additional 104 kB to a maximum of 15.8 MB for creating protected

[2]This also explains our choice of the system name. Similar to Schrödinger's cat (which, put in simple words, can be dead or alive until it is observed), the actual characters in a text protected by SchrodinText are not determined until they are resolved (by the monitor) and displayed to the user

views of the framebuffer. Moreover, it increases the text rendering latency by about 30 ms to 270 ms for UI pages having 20 to 1000 characters (of font size 14 sp) protected by SchrodinText. Only 4 ms to 170 ms of this added latency is for the monitor-based compositing, which allows for acceptable scrolling performance especially for UI surfaces with smaller number of characters in them. Finally, SchrodinText increases the overall CPU usage in this rendering period by about 1 to 3 times, but does not use more than 19% of overall CPU resources in the system. While SchrodinText's overhead is noticeably higher than that of normal text, we believe that its performance is usable and the overhead is justified given the strong protection guarantees that it provides, especially for small number of protected characters.

# 2. BACKGROUND & ATTACKS

## 2.1 Android's Text Rendering

Here, we provide a basic overview of text rendering in Android. For more details, we refer the interested reader to [2].

There are three main operations in text rendering: glyph rasterization, layout, and compositing. Rasterization generates the glyphs of characters given the selected font type and size. The glyphs contain the pixel information of the characters at the target size on the screen. They, however, do not contain the RGB color information; they only contain the alpha channel information (hence one byte per pixel). The rasterized glyphs are stored in a texture buffer. This buffer is later used in compositing.

Layout determines the location of each character on the UI surface, and hence on the framebuffer. The location depends on the size and location of the widget enclosing the text, the surface size (which itself depends on the display size), and glyph sizes, which depend on the text itself as different character glyphs can have varying widths.

The last step is compositing. Once the character glyphs are ready in the texture buffer and their locations on the framebuffer are known, GPU is used (through the OpenGL framework) to read the glyphs and composite them on the right location in the framebuffer. The color of the text is applied at this stage. Compositing is achieved with alpha-blending, where the text glyphs are blended onto the surface below them, e.g., a background image. The framebuffer is then shared with the display controller for showing to the user.

## 2.2 Attacks

In this work, we are concerned with malware that attempts to steal textual content of other applications' UIs. Here, we explain *why* and *how* malware might perform such an attack.

**Why?** There are several categories of mobile applications with important text that malware might attempt to extract. One category is apps with user's private and sensitive information, e.g., messages in a messaging app, posts in a social networking app, bank account information in a banking app, or the social security number (SSN) in a finance app. Indeed, an important category of mobile malware is *spyware*, which attempts to monitor user's sensitive information, e.g., SMS messages. Some examples of such spyware are TheTruthSpy [15] and LetMeSpy [10]. Moreover, Zhou et al. found 644 malware samples (in 27 families) that harvest "user's information, including user accounts and short messages stored on the phones" [54].

Another category is apps that reveal passwords or one-time tokens to the user. For example, a password vault app might show user's passwords to her on the display. As another example, a two-factor authentication application, such as Microsoft and Google Authenticators [9, 12], or a messaging application might show a verification code to the user.

**How?** Here, we mention a few possible methods that malware can use to extract textual content of victim applications. For most of the attacks, malware, originally packaged as a normal app, should exploit a vulnerability in the operating system in order to escalate its privileges for root access or to execute code with kernel privileges. Malware can use various methods to gain root privilege [53], e.g., rowhammer attack [50]. To gain kernel privileges, it can try code injection [38] or Return-Oriented Programming [22, 48].

With root privileges, malware will be able to use two methods to access other applications' textual content. First, malware can simply capture a screenshot of the screen, which, if taken at the right time, will contain sensitive information displayed to the user. Second, it can replace text rendering libraries in Android with a malicious one. The replacement library can then be used to extract the text of other applications, which will load this library for text rendering. Whenever a victim application calls *setText()* on a text widget and passes a string, this string will be available to the library, which can then communicate it with malware, e.g., by writing it to a file.

With kernel privileges, in addition to the aforementioned attacks, malware can either access the application's memory in order to read the plaintext after decryption or read the texture buffers allocated by the graphics stack to hold the character glyphs used in the text.

# 3. OVERVIEW

## 3.1 Key Idea and Design

Our goal is to provide strong protection for sensitive and private textual content in mobile applications' UIs, even in the presence of a compromised operating system. Our key insight is that such content must be displayed to the user only; operating system should not have access to it. Towards this goal, we leverage novel hardware features on modern ARM processors, including virtualization and TrustZone hardware, to create a security monitor. Our key contribution is *oblivious rendering*, a set of techniques that allows the operating system to perform all the stages of text rendering without having access to the text itself. The monitor then resolves the right text on the framebuffer.

## 3.2 SchrodinText UI Widget

In SchrodinText, we do not protect all the textual content of an app. Only selected text, as determined by the application developer, will be protected. We achieve this by providing a new UI widget for protected text, called `SchrodinTextView`, which is mostly similar to the existing `TextView` widget in Android's application programming framework. The developer can position the widget on any desired location in one of application's UI surfaces, i.e., Android activities. The main difference is that, instead of setting a string

```
1 void drawText(byte[] ciphertext,
2               int keyHandle)
3 {
4     String text =
5         decrypt(ciphertext, keyHandle);
6     TextView view = (TextView)
7         findViewById(R.id.textWidget);
8     view.setText(text);
9 }
```

```
1 void drawText(byte[] ciphertext,
2               int textLen,
3               int keyHandle)
4 {
5     SchrodinTextView view = (SchrodinTextView)
6         findViewById(R.id.textWidget);
7     view.setCiphertext(ciphertext, textLen,
8                        keyHandle);
9 }
```

**Figure 1: Rendering protected text using Android's `TextView` (Left) vs. `SchrodinTextView` (Right).**

as the content of the widget (using `setText(String text)`), the encrypted version of the text, i.e., ciphertext, will be set as the content (using `setCiphertext(byte[] ciphertext, ...)`).

Figure 1 shows a sample code snippet for both `SchrodinTextView` and Android's `TextView`. In both cases, we assume that the developer attempts to use encryption for protection of the text. As can be seen, in the existing approach, the application first decrypts the ciphertext (e.g., using Android's Keystore system [1]), and passes it to the operating system for rendering by calling the `setText()` method with the plaintext. While encryption of the text can protect the text from eavesdropping in the network, it cannot protect it from the operating system since the plaintext is available in the application's memory and even passed to the operating system for rendering.

However, with `SchrodinTextView`, the ciphertext never gets decrypted by the application; it is simply passed to the operating system and eventually decrypted in the monitor. However, with `SchrodinTextView`, the rendering API (`setCiphertext()` needs two additional argument: `textLen`, which is the number of characters in the text, needed for layout by the operating system (see §4), and `keyHandle`, which informs the monitor of the key to use to decrypt the ciphertext. As these examples show, the amount of effort needed for using `SchrodinTextView` is comparable with that for using Android's `TextView`.

## 3.3 Workflow

The typical usage of SchrodinText is to allow the application to show a text to the user where the text is sent to the application from its backend server in the cloud. Figure 2 shows a basic overview of SchrodinText including the sequence of events taking place in it. The sequence is as follows. (1) The application's backend server exchanges a key with the monitor (§3.5). (2) The backend then uses the key to encrypt the text and send the ciphertext to the application. The server also sends the length (i.e., number of characters) of the encrypted text and the key handle. (3) The application on the mobile device uses `SchrodinTextView` to show this text. As arguments of this widget, it passes the ciphertext, the length of the text, and the key handle. Moreover, the application configures the widget with the desired font properties (e.g., type, size, and color). (4) Based on this metadata, the operating system rasterizes all the character glyphs and shares them with the monitor through a novel *glyph-book* memory API (§4.1.1) along with the ciphertext and the location of text characters on the framebuffer. (5) The monitor decrypts the ciphertext and uses that to resolve the right glyphs onto the right location on framebuffer shown

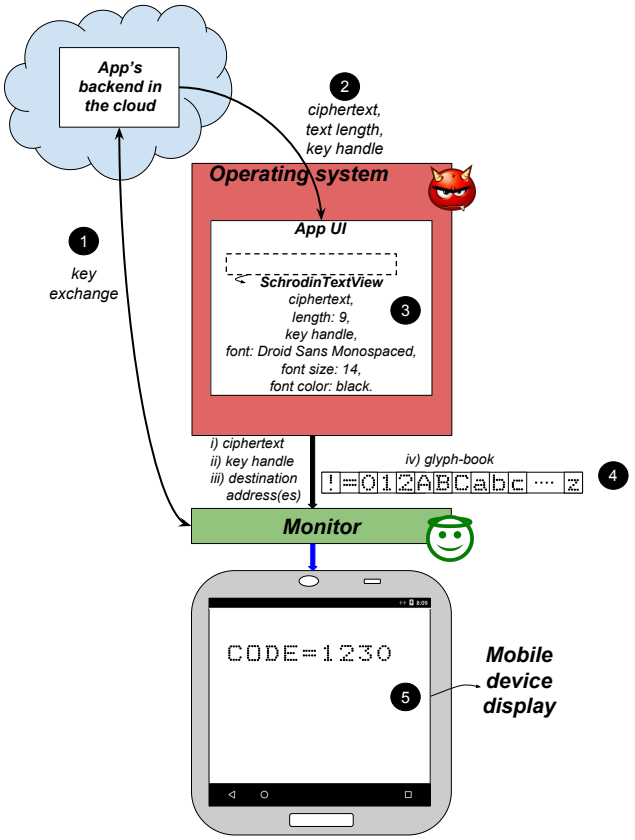



**Figure 2: Overview of SchrodinText's workflow.**

on the display. When resolving the glyph, the monitor also protects the corresponding framebuffer pages from the operating system (to avoid read-back) by leveraging nested page tables and IOMMUs provided by the virtualization hardware. To do this, the monitor uses the *multi-view* pages technique, in which the operating system and DMA-capable devices see one view of a corresponding framebuffer page without the text on it, whereas the display sees a different version with the text on it.

To better understand these steps, consider a messaging application. At install/login time, the application's cloud backend performs a handshake with the monitor to establish a shared secret key. From then on, the backend will encrypt all the messages with this key before delivering them to the application. The application then uses a `SchrodinTextView` to pass the ciphertext for display. The ciphertext is eventually passed to the monitor, which decrypts it and show it

on the display (using the glyph-book and layout information provided by the operating system).

Note that while in most use cases of SchrodinText, the ciphertext is sent from the application's cloud backend, in some use cases, the ciphertext might be retrieved from the local storage of the mobile device. For example, a messaging application might be used to show previously communicated messages. Or a password vault might store the ciphertext of a password in storage and show it to the user when needed.

The operating system performs the layout without access to the text itself. To make this possible, in addition to other techniques (§4.2), we limit the available fonts for `SchrodinTextView` to monospaced fonts, in which all character glyphs have similar widths. This allows the operating system to compute the relative location of each character without knowing the actual text. While monospaced fonts are not as common as normal fonts, they are well supported in mobile devices. For example, Roboto, the new typeface used in Android, provides 10 different styles of monospaced fonts [14], while it provides 12 different styles of normal fonts [13]. Similarly, the older typeface in Android, i.e., Droid Sans, supported one monospaced style [8] and two normal styles [7].

## 3.4 SchrodinText's Security Monitor

The monitor is built on top of two novel hardware security features in modern ARM processors. The first one is the ARM TrustZone [5, 19, 51] that divides the execution into two worlds: normal world and secure world. The normal world hosts the main operating system, such as Android. The secure world hosts security critical components. Moreover, the secure world has access to a device-unique key [18], which is not accessible to the normal world. This makes the secure world ideal for cryptographic operations. Indeed, on Android-based smartphones, Keystore can be implemented in the secure world [1].

The second hardware feature, recently added to ARM processors, is virtualization hardware [27]. It adds a new privilege mode in the normal world (in addition to the user and kernel modes). This new mode, i.e., the *hyp* mode, can be used to host a hypervisor. Among others, virtualization hardware comprises of (*i*) nested page tables, which map the operating system's physical pages to different machine pages (hence virtualizing the memory for the operating system), and (*ii*) I/O Memory Management Unit (IOMMU) components (referred to as System MMU or SMMU in ARM's terminology [4]), which map the physical pages seen in Direct Memory Access (DMA) operations by I/O devices, such as the GPU and display, to different machine pages (hence virtualizing the memory for these I/O devices).

Our monitor uses these hardware components for different purposes. It uses the virtualization hardware to host a hypervisor. The hypervisor is more privileged than the operating system and hence can display the protected text on the framebuffer, while preventing the operating system from accessing it. It does so by creating different views of the framebuffer pages using the nested page tables and IOMMU (§5.1). The monitor uses the secure world for cryptographic key management and operations.

## 3.5 Key Management

As mentioned earlier, the ciphertext is shipped from application's cloud backend to the mobile device. The backend encrypts the text with a key shared with the monitor. Therefore, before the server can use the key, it must establish and share it with the monitor. Different methods can be used for sharing a secret key. For example, the monitor can first generate a public and private key pair and share the public key with the cloud so that it can generate and share a secret key with the monitor. Alternatively, other key sharing mechanisms, such as Diffie-Hellman key exchange [29], might be used to safely share a secret key. In the rest of the paper, we simply assume that the application's cloud backend has already shared a key with the TrustZone's secure world on the device.

Note that it is possible to use public-key encryption for the protected text. That is, the cloud backend can encrypt the text with a public key, for which the private key is only known to the monitor. In SchrodinText, we use symmetric-key encryption since it incurs less computational overhead.

## 3.6 Threat Model

We assume a powerful adversary. Specifically, we assume that malware, originally disguised in an application, can compromise the operating system and gain root or kernel privileges (§2.2).

We assume that our monitor is protected from malware. First, the monitor is protected by the secure boot. Used on many modern commodity devices, secure boot performs integrity measurements of software loaded in memory at boot time, and compares it with the expected value (available as a signed measurement provided by the vendor). If the measurements do not match, a notice is shown to the user informing her that the loaded image is tampered with. We assume that the secure boot protects the TrustZone secure world runtime and the hypervisor. Therefore, no other entity, other than the device manufacturer, can deploy or modify the hypervisor and secure world runtime. Second, we assume that the hypervisor and the secure world are safe against runtime attacks due to their small sizes and narrow attack surfaces. While runtime attacks on these components have been demonstrated in the past [21, 28], they are much less frequent than runtime attacks on the operating system.

As mentioned in §3.5, we assume that the application's cloud backend can safely share a secret key with the monitor and protect the key from unauthorized access (note that the monitor protects the key on the device). If the key is compromised, SchrodinText will be ineffective as the adversary can simply decrypt the ciphertext.

In SchrodinText, we protect the text from some known side-channel attacks. More specifically, the monitor flushes the cache after writing to the protected view of framebuffer pages to defeat cache side-channel attacks. Moreover, the monitor is not easily vulnerable to timing channels as alpha-blending in the monitor uses the same number of loop iterations for all glyphs.

In SchrodinText, we protect the text in `SchrodinTextView` from access by the adversary, i.e., *confidentiality* guarantee. We do not, however, provide any *integrity* or *availability* guarantees.

**Integrity.** There are two ways to mount integrity attacks on the sensitive text. First, although an attacker does not possess the encryption keys, he could still modify the ciphertext arbitrarily. SchrodinText would still decrypt the modified ciphertext, but the plaintext now will be "garbage". Second, the attacker could replace one ciphertext with an-

other (perhaps provided to another secure application). The monitor cannot prevent such an attacks because it cannot verify if the ciphertext comes from the foreground application or not.

One might wonder how SchrodinText can provide additional defenses against these attacks. For the first attack, SchrodinText could use a cryptographically secure hash function to detect arbitrary ciphertext modifications. For the second attack, however, the monitor would need to be able to detect the foreground application. This would unfortunately come with significant additional bloat in the TCB.

**Availability.** An attacker with root or kernel privileges can prevent the text in a `SchrodinTextView` to be shown at all. It can achieve so by modifying the graphics stack so that the rendering of text in `SchrodinTextView` is not fully performed, e.g., by blocking hypercalls to the monitor.

## 3.7 Trusted Computing Base

We assume that the monitor is trusted. This means that both the TrustZone secure world runtime and the hypervisor are trusted. If malware can compromise the secure world, it can extract the protected keys and decrypt the content. If it can compromise the hypervisor, it can revert the protection of memory pages and access the final rendered pixels on the framebuffer or it can simply extract the decrypted text. We also assume that the security hardware is trusted including the virtualization and TrustZone hardware. We do not, however, trust the I/O devices, such as GPU and display; we protect against DMA attacks by them using IOMMUs. The SchrodinText's monitor uses different IOMMUs for different I/O devices. It uses one to give the display controller read-only access to all memory pages so that the display can show the framebuffer (but not leak its content). It also uses other IOMMUs to prevent any access by the GPU and other DMA-capable devices to these protected pages. Therefore, the operating system cannot abuse these devices to access the framebuffer pages containing protected text. §5 will elaborate on these issues.

# 4. OBLIVIOUS RENDERING

In SchrodinText, we enable the operating system, which controls the full graphics stack, to perform the text rendering without access to the text itself. We refer to the set of techniques that make this possible as *oblivious rendering*. As mentioned in §2.1, rendering comprises of three stages: rasterization, layout, and compositing. All of these stages face important challenges in oblivious rendering. For rasterization, our solution is to have the operating system rasterize all printing ASCII characters. For layout, the operating system determines the location of the n-th character on the display. Finally, for compositing, the operating system composites all the content excluding the text protected by SchrodinText. The monitor then composites the text on top of the rest of the content. Next, we will elaborate on these three techniques. More specifically, in this section, we mainly discuss rasterization and layout. In the next section, we address compositing.

**Straw-man solution 1:** One option is to implement the rasterization and layout in the cloud. For this, the application can inform the cloud backend of the font type, size, and color, as well as the size of the UI. The cloud backend can then rasterize the glyphs and lay them out on a buffer and send the encrypted pixels to the application, which can then use the monitor to composite that on the screen. Such a solution has important drawbacks. First, it complicates the cloud backend implementation as it must now be able to lay out and rasterize the text exactly as it would be done by the mobile device. This would, for example, require access to the right fonts on the target platform. The diversity of mobile devices in practice will make this challenging for the cloud backend. Second, this approach increases the network bandwidth consumption. In this case, texts will consume network bandwidth as much as images do.

**Straw-man solution 2:** The second option is to move the graphics stack to the monitor, e.g., to the TrustZone secure world, so that rasterization and layout (as well as compositing) are fully performed in a trusted environment. This approach has important shortcomings as well. First, it significantly increases the size of the TCB. Second, if only a minimal portion of the graphics stack were to be used in the monitor (to keep the TCB small), the content rendered by the monitor would not be visually well-integrated with the rest of the content, which are rendered by the complete graphics stack in the operating system. Third, this solution makes updates to the graphics stack more difficult. For example, new fonts from untrusted sources cannot be used.

## 4.1 Glyph Rasterization

One of the key components of our oblivious rendering technique is its rasterization. Given that the operating system does not have access to the text, the key idea behind our solution is to have the operating system rasterize all the character glyphs given the font type and other metadata, e.g., size and color. The operating system can then pass all of these glyphs to the monitor using a novel memory abstraction, called glyph-book. Depending on the plaintext, the monitor can then resolve the right glyphs on the desired location on the framebuffer (the address of which is determined by the operating system in layout and provided to the monitor).

It is important to note that the glyphs resulting from rasterization in SchrodinText are different from those resulting from normal Android's rasterization (§2.1). In the latter, the glyphs simply contain the alpha channel information (one byte per pixel). The colors are then applied by the GPU at compositing time. In SchrodinText, however, compositing is performed by the monitor (see §5), which merely implements simple alpha-blending. Therefore, we apply the color in the rasterization phase resulting in fully colored glyphs (4 bytes (RGBA) per pixel).

### *4.1.1 Glyph-Book*

The glyph-book API is at the core of oblivious rendering in SchrodinText. It is implemented by the monitor and exposed to the operating system through three hypercalls. We have abstracted this low-level API in a C++ class for the convenience of integrating them with the graphics libraries. The methods supported by this class are as follows:

1) *Constructor*: `GlyphBook::GlyphBook(`
`unsigned int glyphSize, unsigned int numGlyphs,`
`void *ciphertext, unsigned int ciphertextSize,`
`int keyHandle);`

The constructor allocates `numGlyphs` buffers of size `glyphSize` each. Each buffer will hold one of the character glyphs rasterized by the operating system. The number of buffers is the number of printing ASCII characters (ASCII characters

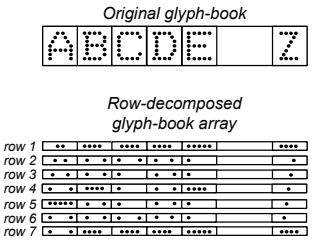



**Figure 3: Row-decomposed glyph-book array.**

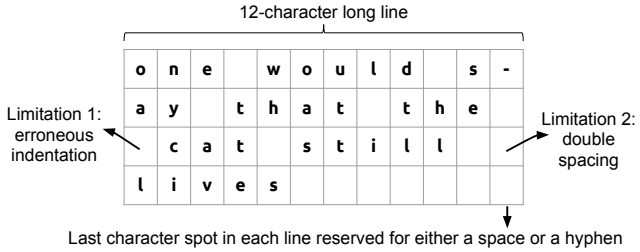



**Figure 4: Oblivious line-breaking demonstration.**

32 to 126, a total of 95 characters). The constructor also receives the ciphertext (`ciphertext`), the size of the buffer holding the ciphertext (`ciphertextSize`), and a handle to a pre-established key to be used for decrypting the ciphertext in the monitor (`keyHandle`).

2) *Getter*: `uint8_t *GlyphBook::getBuffer( unsigned int i);`

The getter simply returns a pointer to the i-th buffer in a glyph-book. Note that the getter is handled fully in the operating system and does not need to interact with the monitor. The rendering library uses the getter method to populate the glyph-book.

3) *Resolver*: `int GlyphBook::resolve( uint8_t *dstAddr, unsigned int textPos, bool conditional);`

The resolver blends the right glyph for the `textPos`-th character of the text into the destination (`dstAddr`) address. The right glyph depends on the text, which is known to the monitor. The monitor uses the ASCII value of the `textPos`-th character to compute the index into the array of buffers containing the glyphs and alpha-blends that to the destination. The last argument of this API (`conditional`) is used to ask the monitor for conditional resolve, which will be discussed in §4.2.

For example, assume that the ciphertext encrypts the string "CODE=1230". Once the glyph-book is shared with the monitor, the operating system calls the `resolver` API 9 times, once per character. At first, it calls the API for the first character; the monitor then uses the plaintext, retrieves the ASCII value of the first character (67 for C), and uses that as an index in the array[3] to retrieve the glyph, and blends it onto the framebuffer address provided by the operating system.

Note that the resolver first protects the destination framebuffer pages before blending the content onto them. The protection is done using the multi-view pages technique (§5.1). It also adds this destination address to a list, needed by the destructor.

4) *Destructor*: `GlyphBook::~GlyphBook();`

The destructor iterates through the aforementioned list of protected pages, zeros out the content copied by the resolver, and unprotects them.

### 4.1.2 Row-Decomposition of the Glyph-Book

It is critical to keep the implementation of the glyph-book API minimal in the monitor. Therefore, we implement the `resolve` API to perform alpha-blending on a contiguous memory buffer (determined by a start address (`dstAddr`) and size (`glyphSize`)). However, blending a character glyph into the framebuffer cannot simply be performed using a contiguous memory buffer as the glyph comprises of multiple rows, and these rows are not contiguous on the framebuffer.

The first solution that we considered was to update the implementation of the `resolve` API so that it could take a glyph, iterate over its rows, and blend them onto the right addresses in the framebuffer (derived from a single address pointing to the location of the first row of the glyph in the framebuffer and the widths of the framebuffer). However, we rejected this solution since it would complicate the monitor's implementation.

To address this challenge, we decompose the glyphs into their rows and store them in separate glyph-books. In this case, the operating system creates one glyph-book consisting of the first rows of all glyphs, another including the second rows, and so on. The operating system then issues multiple resolve API calls on all of these glyph-books passing the destination address of corresponding rows in these calls. Since the pixels in a row are contiguous in memory, the monitor's resolve API blends them on a contiguous destination address in the framebuffer. Figure 3 illustrates this technique.

## 4.2 Layout

In SchrodinText, we also perform the layout in the operating system without access to the text. Note that the operating system knows the number of the characters in the string to be shown in the widget as this is one of the arguments of `setCiphertext()` in `SchrodinTextView`. Moreover, we limit the `SchrodinTextView` to monospaced fonts. With monospaced fonts, the operating system can compute the location of each character relative to the previous one without knowing the exact characters since all character glyphs have equal widths.

However, even with monospaced fonts, we face an important challenge: determining the *line-breaks* and *consequent hyphenation*. The first straw-man approach that we considered was to inform the operating system of the lengths of the words in the text, e.g., a 4 character word followed by a 5 character word for a total of 10 characters (including the space). This way, the operating system could determine appropriate line breaks and hyphenation. However, this approach would enable the operating system to gain side-channel information about the text using word-length frequency analysis.

Our solution to solve this challenge is called *oblivious line-breaking* that determines the line breaks without word-length information. In this solution, the operating system reserves the last character spot in the line for either a space or a hyphen. For that spot, it issues a *conditional resolve* to the monitor (§4.1.1). More specifically, it asks the monitor

[3]More specifically, (67-32) is the index since the first glyph in the glyph-book corresponds to space with ASCII value of 32. The previous 32 ASCII values are non-printing.

to leave that spot empty if either the preceding or proceeding characters in the plaintext is a space (ASCII #32) and to fill it with a hyphen (ASCII #45), otherwise.

Figure 4 illustrates a sample text laid out over 4 lines using this technique. As can be seen, this technique has two limitations: *erroneous indentation* and *double spacing*. The former happens when the first character in the line is a space. The latter happens when the character before the last in the line is a space. While less elegant compared to more advanced layout algorithms, SchrodinText performs adequately well, resulting in fully readable text (the errors are at most two character glyphs wide).

Finally, note that SchrodinText's layout does not support tab spaces or newlines. For the former, multiple spaces must be used. For the latter, separate `SchrodinTextViews` need to be utilized.

# 5. SECURE COMPOSITING

In this section, we focus on compositing in SchrodinText and its challenges. Once rasterized glyphs are ready and their locations are determined, the next stage is to composite the right glyphs on top of other graphics layers, e.g., background image in the app, and generate the framebuffer to be displayed to the user. Compositing the protected text in SchrodinText must meet the following requirements: the text on the framebuffer must be visible to the display controller so that it can be shown to the user. It, however, must not be available to the operating system for read-back (either with direct access or through DMA). Also, the operating system must be able to composite all the unprotected content with the GPU as it normally does for performance.

We use two techniques to meet these goals: *multi-view pages* and *two-stage compositing*. The former technique uses the CPU MMU and various IOMMUs in the ARM SoC to provide different views of the framebuffer pages containing protected text pixels. The latter technique allows the operating system to perform its own compositing using the GPU and enables the monitor to composite the protected text in CPU. Below, we describe these two techniques in more details.

## 5.1 Multi-View Pages

Normally, a memory page (determined by its physical address from the operating system's perspective) is available to the operating system and all I/O devices for access. That is, reading a page either directly from the operating system (using CPU's load instructions) or reading it using I/O devices (with DMA) will return the same values. A *multi-view page* changes this paradigm. With this technique, access to the same page by the operating system and I/O devices can return different values.

We implement this technique using the nested page tables and IOMMUs in the ARM SoC. Using the nested page tables and IOMMUs, the monitor can map a physical address seen by the operating system and DMA-capable devices, respectively, to different pages in memory. Figure 5 shows an overview of this technique. For a given page of interest, the monitor maintains two underlying pages (i.e., *views*), a *protected view* and an *unprotected view*. The two have similar content except that the protected text pixels are blended on the protected view only. The monitor then maps the protected view in the IOMMU in front of the display controller; it maps the unprotected view for the operating system (using the MMU) and the rest of DMA-capable devices (using the IOMMUs in front of them). When calling the `resolve` API of the glyph-book abstraction on a given page for the first time, the monitor creates the protected view, copies the content of the unprotected view into it, blends the right glyph buffers to the protected view, and then maps it to the display's IOMMU at the same physical address. For the following `resolve` API calls on the page, the monitor simply performs alpha-blending on the protected view, but not the unprotected one.

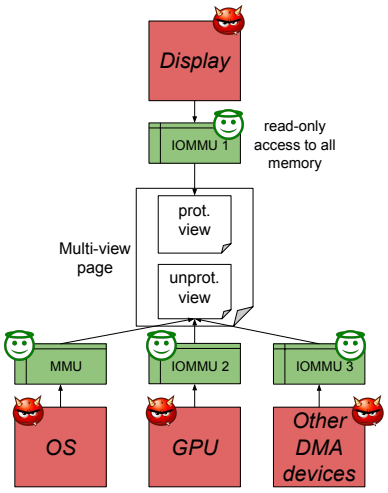



**Figure 5: Multi-view pages.**

## 5.2 Two-Stage Compositing

In Android and on all modern smartphones equipped with GPUs, the compositing is performed by the GPU for best performance. GPU compositing creates an important challenge for SchrodinText as it requires the GPU to have read-access to the resolved text glyphs. Given that the GPU is programmed by the operating system, GPU compositing would create a channel for the operating system to access the resolved glyphs.

We considered different solutions for this problem. The first option was to perform all compositing by the GPU and perform sanity checks in the monitor on the GPU instructions issued by the operating system to guarantee that the resolved glyphs are only copied to the framebuffer (which is protected from the operating system). However, this solution would significantly bloat the monitor's TCB as it would require it to parse and understand the GPU instructions. Moreover, this solution would make the monitor highly specific to the GPU model, which would make it challenging to port to other devices with different GPUs. The second solution that we considered was to execute the compositing fully by the CPU. This solution would however significantly degrade the graphics performance even for contents that are not protected.

We thus adopted a third solution, called *two-stage compositing*, which is a middle ground between these two solutions. In this solution, in the first stage, the operating system composites the unprotected layers using the GPU. In the second stage, the monitor composites the protected text using the CPU. Figure 6 illustrates this approach.

Once the application's UI surface is moved out of the vis-

ible region, e.g., pushed to the background or obscured by the dragged notification bar, the operating system must replace the content of the framebuffer including the protected pages in it. To do this, the operating system can simply call the destructor of the glyph-books. The destructor will wipe the rendered text and unprotect the protected page (visually resulting in the text disappearing from the display). In order to improve the performance, if the surface will be used in the future, the operating system can keep the current framebuffer in memory, replace it with another framebuffer, and replace it back when the previous surface needs to be shown. We have not implemented this optimization.

Compositing requires the monitor to implement alpha-blending. We currently implement this in software in the monitor with less than 20 lines of code. Note that while simple, our current alpha-blending implementation incurs noticeable computational overhead as it loops over all the pixels in the character glyphs and blends them on top of the background pixels. In §8.5, we evaluate the CPU overhead of SchrodinText, which includes the computational overhead of alpha-blending. In the future, we plan to investigate using alpha-blending hardware support in the display controller to accelerate this operation in the monitor [49].

## 5.3 Security Analysis

**Display controller and its driver.** The display has access to the protected view of the page. This raises an important concern: since the display controller is programmable by the operating system (through the display controller driver), it can be compromised by the operating system and then used to access the protected view. To address this concern, we map all the memory pages as read-only in the IOMMU of the display. Therefore, the display hardware is not able to leak out the protected content to any other locations in the memory. Moreover, a compromised display controller driver cannot access the content using CPU instructions since it is limited by the nested page tables. It cannot trigger a DMA request on its own either; it needs to program the display controller to do so, which is then limited by the IOMMU.

**GPU.** One might wonder whether the GPU can be abused by the operating system to access the protected views. To protect against this, the monitor maps the unprotected view of the pages into the IOMMU in front of the GPU. That is, SchrodinText programs this IOMMU to prevent the GPU to have any access (whether read or write) to the memory pages holding the protected text. This IOMMU is set in this restricted mode before the protected text is composited on the framebuffer by the monitor and is disabled after zeroing out the pages holding the text.

# 6. APPLICATION INTEGRATION

To demonstrate practicality and ease-of-use of SchrodinText, we have used it in a messaging application. More specifically, we have modified the open source `Xabber` application (an XMPP client) [17] so that all the received messages are displayed to the user using the `SchrodinTextView` widget. This only required us to modify a few lines of code, in order to use `SchrodinTextView` instead of `TextView`.

# 7. IMPLEMENTATION

We have implemented SchrodinText on an ARM Juno r0 development board [20]. To the best of our knowledge, ARM

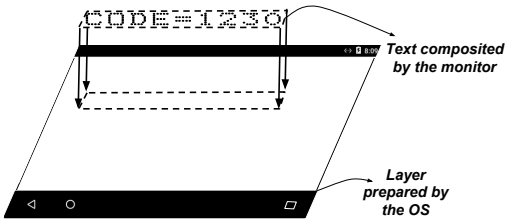



**Figure 6: Secure two-stage compositing.**

Juno boards are the only devices that allow us to program both the hypervisor and the TrustZone secure world runtime, both of which are needed for our prototype. The Juno r0 development board incorporates 6 CPUs (2 Cortex-A57 and 4 Cortex-A53 in the big.LITTLE architecture).

In our prototype, we use Android Marshmallow (version 6.0-16.04) for the normal world operating system, which uses the Linux kernel version 3.18. The operating system is configured with 1 GB of memory. We use Xen version 4.6 for the hypervisor and Open-TEE version 16.04 for the secure world runtime. We use AES encryption for the ciphertext and use the mbed TLS library [11] in the secure world to decrypt it.

Our multi-view page approach requires controlling the IOMMUs (i.e., SMMUs in ARM) for various DMA-capable devices. On the Juno board, we leverage four SMMUs: one MMU-400 in front of the GPU, one MMU-401 in front of the display, one MMU-401 in front of the DMA engine, and one MMU-401 in front of the USB devices. Note that the board incorporates SMMUs for other components such as the PCI interface. However, we disable these components in our prototype.

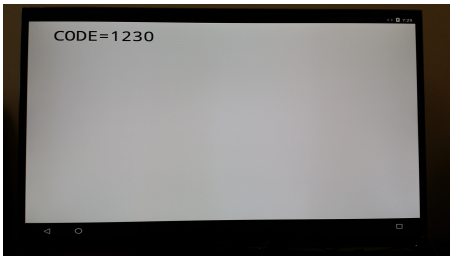
We implement our oblivious rendering in two parts. First is the implementation of the glyph-book abstraction in the hypervisor. We also implement a proxy layer in the operating system kernel to enable the user space rendering libraries to invoke the hypercalls provided by the glyph-book in the hypervisor. Second is the implementation of the `SchrodinTextView` in Android. This requires modifications to the text rendering libraries in Android including the Java-based implementation of the text widget and its corresponding native layer, the Minikin layout library, and the hardware-accelerated UI rendering library.

# 8. EVALUATION

In this section, we evaluate the protection and TCB reduction provided by SchrodinText as well its rendering overhead. More specifically, for the latter, we measure the added memory usage and latency compared to using existing `TextView` widgets supported by Android. Note that we conduct our experiments in a freshly installed Android operating system with no applications installed other than the default ones. Moreover, we reboot the operating system before every experiment. In practice, however, the overhead of SchrodinText can affect other apps running concurrently in the system as well. To capture such an effect, we also measure and report the CPU usage incurred by SchrodinText.

## 8.1 Protection

As mentioned in §2.2, there are various ways for an attacker to extract other applications' textual content. In this section, we experimentally demonstrate how SchrodinText




(a) What the user sees on the display

(b) What the operating system sees in the framebuffer

**Figure 7: SchrodinText protects the text against framebuffer read-back attacks.**

will protect against framebuffer read-back attack (e.g., taking a screenshot). SchrodinText's protection against the rest of the attacks is simple to explain as it never decrypts the ciphertext in the operating system. As a result, extraction through a compromised graphics stack (including text rendering libraries) and memory dumping is ineffective.

SchrodinText protects against framebuffer read-back attack due to its protection of the framebuffer pages (using the multi-view page technique). To evaluate this, we use `SchrodinTextView` to show a text on the application's UI, and then attempt to extract the text by taking a snapshot of the framebuffer content (we have root privilege, mimicking an attacker with such a privilege). Figure 7 illustrates the outcome of this experiment. Figure 7 (a) is a photo of what is seen on the display, taken by an external camera. Figure 7 (b) is a screenshot resulting from reading the framebuffer in the operating system. As can be seen, the text shows up on the screen but not on the screenshot. Everything else is the same including the white background, Android's notification bar on the top, and Android's virtual buttons on the bottom of the screen. The size of the screen is large since our Juno development board uses an external display.

## 8.2 TCB Reduction

SchrodinText eliminates large swaths of code that would otherwise have to be trusted. Without SchrodinText, we would have to trust a large portion of the operating system including its kernel (which itself contains various device drivers such as the GPU driver) and the user space graphics stack (including UI related frameworks and libraries). We tried to measure the size of the latter. The components that we identified contain about half a million lines of code (not including closed source libraries such as OpenGL).

SchrodinText does add additional code, but this codebase is small. Notably, it requires a hypervisor, which is much smaller than the operating system kernel. Moreover, the hypervisor is not only specific to SchrodinText and can be used for other security purposes in the system as well [26,41]. The part of the hypervisor that we developed for SchrodinText is only approximately a thousand lines of code. Finally, SchrodinText does not add any new functionality to the TrustZone secure world other than what is already available in it in commodity devices, i.e., key management and decryption of a ciphertext (§3.4).

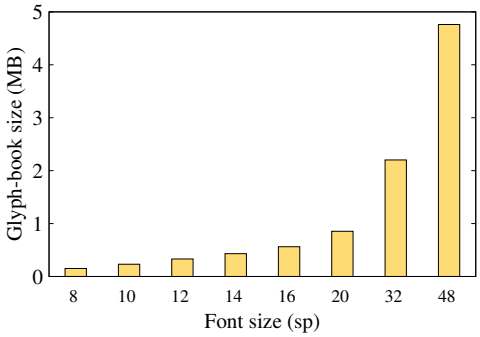



**Figure 8: Size of glyph book for varying font sizes. The font size unit is in sp (scale-independent pixel).**

## 8.3 Memory Usage

There are two sources of memory overhead in SchrodinText: memory used for the glyph-book (§4) and for multi-view pages (§5.1).

*Glyph-book.* The operating system needs to render the colored glyphs for all the characters given the font type, size, and color. We measure the size of the glyph-book for the default monospaced font in Android installed on our board (Droid Sans Monospaced) for varying font sizes. Figure 8 shows the results. It shows that the size of the glyph-book is small for normal font sizes (a few hundreds of kB, e.g., 150 kB for font size 8 sp). The glyph-book size does however become exponentially large for larger fonts, e.g., about 4.8 MB for font size 48 sp. The exponential growth is due to glyphs being two dimensional.

*Multi-view pages.* The monitor needs to create a protected view of every framebuffer page that contains protected text pixels. The amount of memory used for this depends on the number of characters and their size. A single character glyph affects several memory pages. For example, a single glyph in Droid Sans Monospaced font at sizes 8 sp and 48 sp affect 26 and 146 pages, respectively, resulting in 104 kB and 584 kB of memory needed for the protected views. However, two consecutive glyphs will not use twice as much as they mostly affect the same memory pages on the framebuffer.

At most, multi-view pages can result in as much memory as the size of the framebuffer, which depends on the screen size and its resolution. For example, the size of the framebuffer in the Juno board (which is connected to a 27" display) is about 15.8 MB. The size of the framebuffer is however often smaller on commodity devices. For example on a Nexus 5 smartphone, the framebuffer size is about 8 MB.

## 8.4 Rendering Latency

SchrodinText increases the rendering latency. We measure this latency from the time that the text widget (either

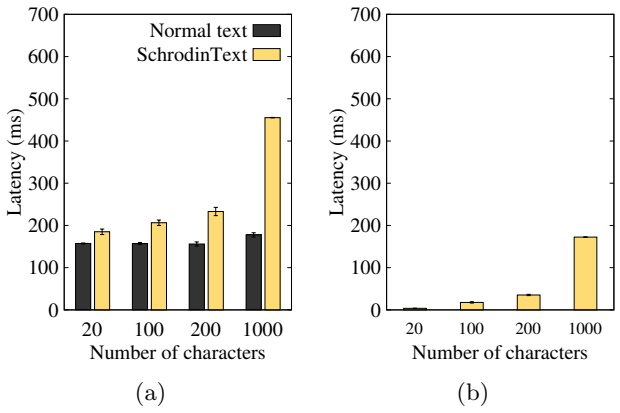



**Figure 9: (a) Overall text rendering latency. (b) Monitor compositing latency in SchrodinText.**

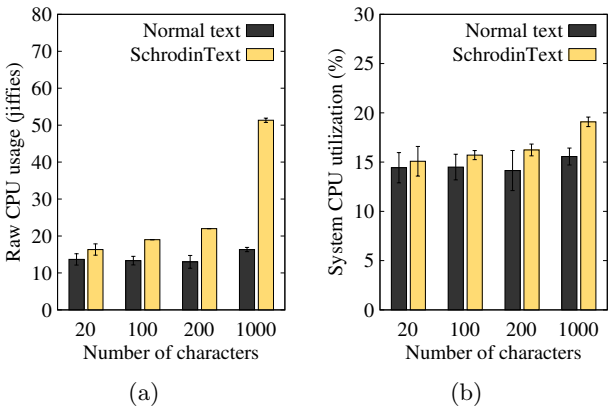



**Figure 10: (a) Raw CPU usage. (b) System CPU utilization during rendering.**

`SchrodinTextView` or `TextView`) is created until when the renderer is finished rendering the text. We perform experiments with application's UI pages with a varying number of characters in them. We use four different numbers: 20 characters representing a small text such as SSN, account information, and password; 100 and 200 characters representing messages; and 1000 characters representing a UI full of text. We use font size 14 sp in the experiments and divide the text into maximum 35 character lines, each line in a separate widget. A major contributor to the latency is the time needed for monitor-based compositing, which results in one hypercall per row of each glyph used in the text. We measure this latency as well.

## 8.5 CPU Usage

Figure 9 shows the results. We have repeated each experiments three times and shown the average and standard deviation. There are three observations. First, SchrodinText adds modest latency to the overall text rendering (about 30 ms to 270 ms for 20 to 1000 protected characters). This impacts the latency incurred when a UI surface is first created, e.g., at application's launch time. We believe that this latency is usable given that mobile application's launch latency has an average of 2 seconds [42] and can be as high as 20 seconds [52]. Moreover, predictive app preloading can be used with SchrodinText to further hide this latency [42, 52]. Second, for small number of characters on the screen, the latency is small. Only when the screen is full of text, e.g., with 1000 characters, the latency becomes noticeable. Finally, SchrodinText can support scrolling text pages with adequate framerate. More specifically, the scrolling performance is determined by the compositing latency. As the figure shows, SchrodinText's compositing incurs small latency (4 ms to 170 ms for 20 to 1000 protected characters). At 200 characters, compositing takes about 35 ms, which can support scrolling at near 29 (= 1000/35) Frames Per Second (FPS). At 1000 characters, however, compositing takes about 170 ms, which would cap the scrolling framerate to about 5 (= 1000/170) FPS. As can be seen in Figure 9, the latency in the monitor increases exponentially with the number of characters. This is due to the alpha-blending in the monitor, which operates on each pixel, resulting in exponential latency increase for two dimensional character glyphs.

We also measured the time needed to construct the glyph-book for various font sizes. Regardless of the font size, constructing the glyph-book does not take more than 30 ms, often much less. Although small, this time can be removed from the rendering latency by proactively creating the glyph-book or caching it for future use.

We measure the CPU usage overhead of rendering in SchrodinText. We measure the CPU usage overhead by measuring the amount of CPU used in the system from the beginning to the end of the text rendering period. Some of the CPU usage in this period might be due to other processes running in the background. We make sure to minimize this effect by not installing any other apps than the one under experiment and a few default ones available in the system. Also, the application itself is simple and does nothing but showing the text.

Figure 10 shows the results. Figure 10 (a) shows the raw CPU usage in jiffies (a unit of time used in Linux). It shows that the total amount of CPU usage has increased by 1-3×. This is because SchrodinText incurs computational overhead when creating the glyph-book as well as blending all the pixels in software (§5.2). In contrast, normal text in Android is blended fully in GPU, reducing the CPU usage.

Despite the overall increase in CPU usage, SchrodinText does not saturate the CPUs available in the system. Figure 10 (b) shows the CPU utilization in the system (the ratio of time when CPU is used to when it is idle). The SchrodinText's rendering components run mostly in a single thread incurring no more than 19% CPU utilization during the rendering period, slightly more than that of the normal text rendering.

# 9. RELATED WORK

## 9.1 Untrusted Operating System

Research on untrusted operating systems shares our motivation: modern commodity operating systems are too large and complex to be safe. Hence, many systems have attempted to protect the applications from the operating system assuming that the operating system can easily gets compromised by other malicious applications. Overshadow [25] and Inktag [35] were some of the early such systems. They

use the hypervisor to protect the application. Recently, Intel added the Software Guard Extensions (SGX) [40] allowing the application to to create an isolated execution environment, i.e., an enclave, and protect itself, e.g., its memory, from the operating system. Unfortunately, the job of full protection of an application from its operating system is a daunting task. Checkoway et al. showed that the operating system has many ways to attack an application, e.g., by controlling the random number generator [23].

Dealing with untrusted operating systems in mobile devices has been an active line of work as well. Mobile solutions often use the TrustZone hardware for this purpose. They allow sensitive frameworks to be offloaded to the secure world and hence isolated from the operating system. For example, AdAttester [36] pushes the display and touchscreen drivers to the secure world allowing it to provide guarantees for ad providers, e.g., that the ad is shown correctly and that the click on the ad is not fake. TLR [46] provides a managed runtime (based on .NET) to run the security sensitive parts of application in the secure world. Trusted sensors [33,34,39,47] use a trusted hardware component (either TrustZone or Mobile Trusted Module (MTM) [16]) to provide authenticity guarantees for sensor readings. More specifically, the trusted sensor framework in [39] moves the sensor driver to the secure world and use the cryptographic operation support in it to sign the sensor readings. And YouProve [34] uses the trusted hardware to provide signed statements about transformed sensor readings generated in the device. Finally, SIMlet [43] uses the TrustZone to provide a trustworthy framework to monitor a mobile device's network traffic in order to enable content providers and mobile users to co-pay for the cost of network traffic.

In pocket hypervisors [26], Cox et al. envisioned the hypervisor to be used for providing security services in mobile devices. SchrodinText is one realization of this decade-old mobile computing vision.

There exists solutions for building a trusted path between an application (or an Internet service) and the user, assuming that the operating system is untrusted [37, 55]. Zhou et al. [55] use a hypervisor to give an application direct, exclusive, and protected access to I/O devices, e.g., keyboard and display. In contrast, SchrodinText does not give any applications exclusive access to the display. It allows the operating system to manage the display as it normally does. It, however, protects selected textual content that the application shows to the user. Li et al. [37] use the secure world to protect the framebuffer and enforce background/foreground colors for it (to match the LED colors also enforced by the secure world) in order to protect against a framebuffer overlay attack. Their work, however, does not protect selected textual content on the UI.

Virtual Machine Introspection (VMI) uses the hypervisor for intrusion detection [30–32] and for detection of malicious activities in the operating system. SchrodinText is related to VMI as it uses the hypervisor to protect the textual content on the UI against a malicious operating system.

Finally, unlike these systems, SchrodinText leverages both the hypervisor and the TrustZone secure world.

## 9.2 Digital Rights Management

Digital Rights Management (DRM) is a framework that allows mobile devices to protect the digital media (such as video and audio) from unauthorized access. This framework allows the application's cloud backend, e.g., Netflix cloud service, to encrypt the content and share them with the application. Current implementation of DRM in Android decrypt the content in user space system services and hence are also vulnerable to the attacks by a compromised operating system. Future devices, however, are starting to embrace full hardware implementation for DRM, such as ARM Mali-V500 video processor [3]. In such a device, the encrypted content is delivered to a video processor hardware (integrated with the TrustZone secure world), which decodes the content and passes it to the display, all hidden from the operating system. It is however important to note that protecting digital video in this manner is relatively simpler than textual content for two reasons: First, the video does not need to be rendered; it already comes with all the pixel information available. All the video decoder hardware has to do is to decrypt the content and show it on the display. Hence, the operating system role is minimal. Second, the video is often full-screen and not fully integrated with the app UI. In contrast for text, the exact pixels to be rendered depend on the selected font type, size, and color as well as the size of the UI, text widget, and the display, most of them are not known to the cloud backend. Moreover, textual content is embedded in the app UI and highly integrated with it. However, we note that once such video processors are available in commodity devices, SchrodinText can use them for accelerating its software-based compositing.

## 9.3 Others

Unfortunately, protecting the UI integrity in Android has been challenging allowing an attacker to hijack the UI, which has triggered research into finding mitigations [24,44,45]. As mentioned in §3.6, SchrodinText does not provide integrity guarantees. Instead, it provides strong confidentiality guarantee for selected textual content on the UI.

CAPTCHA [6] attempts to identify human users (vs. computers) by showing a distorted text on the display and asking the user to type the text (a task difficult for a computer to do). Unlike SchrodinText, CAPTCHA does not protect the confidentiality of content shown on the display.

# 10. CONCLUSIONS

We presented SchrodinText, a system solution for strongly protecting application's sensitive textual content from malware that has compromised the operating system. SchrodinText leverages modern security hardware features on ARM platforms to protect and display the text. Our key contribution was oblivious rendering, a set of novel techniques to allow the operating system to perform the rendering without having access to the text, allowing us to keep the monitor, and hence the TCB, small. Our evaluation on the ARM Juno development board shows that SchrodinText incurs noticeable overhead but that its performance is usable. We believe that SchrodinText can be used to safely display various sensitive textual information to mobile users.


## Acknowledgments

The author would like to thank the anonymous MobiSys reviewers and the paper shepherd, Rajesh Krishna Balan, for their insightful feedback. He would also like to thank Stefan Saroiu for the helpful and constructive discussions regarding this paper.

## 11. REFERENCES


[1] Android Keystore System. https://developer.android.com/training/articles/keystore.html.

[2] Android's Font Renderer. https://medium.com/@romainguy/androids-font-renderer-c368bbde87d9#.rvl02m0u0.

[3] Arm mali-v500.

[4] ARM SMMU. http://www.arm.com/files/pdf/System-MMU-Whitepaper-v8.0.pdf.

[5] ARM TrustZone. http://www.arm.com/products/processors/technologies/trustzone/index.php.

[6] CAPTCHA. http://www.captcha.net/.

[7] Droid sans fonts.

[8] Droid sans monospaced font.

[9] Google Authenticator. https://play.google.com/store/apps/details?id=com.google.android.apps.authenticator2.

[10] LetMeSpy. http://www.letmespy.com/.

[11] mbed TLS. https://tls.mbed.org/.

[12] Microsoft Authenticator. https://play.google.com/store/apps/details?id=com.azure.authenticator.

[13] Roboto fonts.

[14] Roboto monospaced fonts.

[15] TheTruthSpy. http://thetruthspy.com/.

[16] Trusted computing group mobile specifications.

[17] Xabber: open source xmpp client for android.

[18] ARM Security Technology, Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2004.

[19] TrustZone: Integrated Hardware and Software Security: Enabling Trusted Computing in Embedded Systems. In *ARM White Paper*, 2004.

[20] ARM. Juno ARM Development Platform SoC, Revision r0p0, Technical Overview. *ARM DTO*, 0038A (ID040516), 2014.

[21] A. M. Azab, K. Swidowski, J. M. Bhutkar, W. Shen, R. Wang, and P. Ning. SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM. In *Proc. ACM MobiSys*, 2016.

[22] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy. Return-Oriented Programming without Returns. In *Proc. ACM CCS*, 2010.

[23] S. Checkoway and H. Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proc. ACM ASPLOS*, 2013.

[24] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proc. USENIX Security*, 2014.

[25] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: a Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proc. ACM ASPLOS*, 2008.

[26] L. P. Cox and P. M. Chen. Pocket Hypervisors: Opportunities and Challenges. In *Proc. IEEE/ACM HotMobile*, 2007.

[27] C. Dall and J. Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proc. ACM ASPLOS*, 2014.

[28] N. V. Database. Vulnerability summary for cve-2015-6639.

[29] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[30] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proc. IEEE Security and Privacy (S&P)*, 2011.

[31] Y. Fu and Z. Lin. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *Proc. IEEE Security and Privacy (S&P)*, 2012.

[32] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. Internet Society NDSS*, 2003.

[33] P. Gilbert, L. P. Cox, J. Jung, and D. Wetherall. Toward Trustworthy Mobile Sensing. In *Proc. ACM Workshop on Mobile Computing Systems & Applications (HotMobile)*, 2010.

[34] P. Gilbert, J. Jung, K. Lee, H. Qin, D. Sharkey, A. Sheth, and L. P. Cox. YouProve: Authenticity and Fidelity in Mobile Sensing. In *Proc. ACM SenSys*, 2011.

[35] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proc. ACM ASPLOS*, 2013.

[36] W. Li, H. Li, H. Chen, and Y. Xia. AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone. In *Proc. ACM MobiSys*, 2015.

[37] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C. Chu, and T. Li. Building Trusted Path on Untrusted Device Drivers for Mobile Devices. In *Proc. ACM Asia-Pacific Workshop on Systems (APSys)*, 2014.

[38] A. Lineberry. Malicious Code Injection via /dev/mem. *Black Hat Europe*, page 11, 2009.

[39] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software Abstractions for Trusted Sensors. In *Proc. ACM MobiSys*, 2012.

[40] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proc. Inter. Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, page 10, 2013.

[41] S. Mirzamohammadi and A. Amiri Sani. Viola: Trustworthy Sensor Notifications for Enhanced Privacy on Mobile Systems. In *Proc. ACM MobiSys*, 2016.

[42] A. Rahmati, C. Shepard, C. Tossell, L. Zhong, and P. Kortum. Practical Context Awareness: Measuring and Utilizing the Context Dependency of Mobile Usage. *IEEE Transactions on Mobile Computing*, 14(9):1932–1946, 2015.

[43] H. Raj, S. Saroiu, A. Wolman, and J. Padhye. Splitting the Bill for Mobile Data with SIMlets. In *Proc. ACM HotMobile*, 2013.

[44] C. Ren, P. Liu, and S. Zhu. WindowGuard:

Systematic Protection of GUI Security in Android. In *Proc. Internet Society NDSS*, 2017.

[45] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards Discovering and Understanding Task Hijacking in Android. In *Proc. USENIX Security*, 2015.

[46] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to Build a Trusted Language Runtime for Mobile Applications. In *Proc. ACM ASPLOS*, 2014.

[47] S. Saroiu and A. Wolman. I Am a Sensor, and I Approve This Message. In *Proc. ACM Workshop on Mobile Computing Systems & Applications (HotMobile)*, 2010.

[48] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proc. ACM CCS*, 2007.

[49] Texas Instruments. Architecture Reference Manual, OMAP4430 Multimedia Device Silicon Revision 2.x. SWPU231N, 2010.

[50] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proc. ACM CCS*, 2016.

[51] J. Winter. Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms. In *Proc. ACM Workshop on Scalable Trusted Computing (STC)*, 2008.

[52] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu. Fast App Launching for Mobile Devices Using Predictive User Context. In *Proc. ACM MobiSys*, 2012.

[53] H. Zhang, D. She, and Z. Qian. Android Root and its Providers: A double-Edged Sword. In *Proc. ACM CCS*, 2015.

[54] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE Symposium on Security and Privacy*, 2012.

[55] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2012.