Rushmore: Securely Displaying Static and Animated Images Using TrustZone

Chang Min Park [1,3], Donghwi Kim [2], Deepesh Veersen Sidhwani [1], Andrew Fuchs [1], Arnob Paul [1], Sung-Ju Lee [2], Karthik Dantu [1], Steven Y. Ko [1,3]

[1]University at Buffalo, {cpark22, deepeshv, afuchs2, arnobpau, kdantu}@buffalo.edu
[2]KAIST, {dhkim09, profsj}@kaist.ac.kr
[3]Simon Fraser University, steveyko@sfu.ca

ABSTRACT

We present `Rushmore`, a system that securely displays static or animated images using TrustZone. The core functionality of `Rushmore` is to securely decrypt and display encrypted images (sent by a trusted party) on a mobile device. Although previous approaches have shown that it is possible to securely display encrypted images using TrustZone, they exhibit a critical limitation that significantly hampers the applicability of using TrustZone for display security. The limitation is that, when the trusted domain of TrustZone (the secure world) takes control of the display, the untrusted domain (the normal world) cannot display anything *simultaneously*. This limitation comes from the fact that previous approaches give the secure world exclusive access to the display hardware to preserve security. With `Rushmore`, we overcome this limitation by leveraging a well-known, yet overlooked hardware feature called an IPU (Image Processing Unit) that provides multiple display channels. By partitioning these channels across the normal world and the secure world, we enable the two worlds to simultaneously display pixels on the screen without sacrificing security. Furthermore, we show that with the right type of cryptographic method, we can decrypt and display encrypted animated images at 30 FPS or higher for medium-to-small images and at around 30 FPS for large images. One notable cryptographic method we adapt for `Rushmore` is visual cryptography, and we demonstrate that it is a light-weight alternative to other cryptographic methods for certain use cases. Our evaluation shows that in addition to providing usable frame rates, `Rushmore` incurs less than 5% overhead to the applications running in the normal world.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; *Trusted computing*.

MobiSys '21, June 24-July 2, 2021, Virtual, WI, USA

ACM ISBN 978-1-4503-8443-8/21/07…$15.00
https://doi.org/10.1145/3458864.3467887

KEYWORDS

secure image display; visual cryptography; TrustZone

ACM Reference Format:
Chang Min Park [1,3], Donghwi Kim [2], Deepesh Veersen Sidhwani [1], Andrew Fuchs [1], Arnob Paul [1], Sung-Ju Lee [2], Karthik Dantu [1], Steven Y. Ko [1,3]. 2021. Rushmore: Securely Displaying Static and Animated Images Using TrustZone. In *The 19th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '21), June 24-July 2, 2021, Virtual, WI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3458864.3467887

1 INTRODUCTION

Mobile app users desire to display images and animations without compromising on their privacy or confidentiality. For example, users of a photo sharing app want to share their personal photos and videos without worrying that their privacy might be violated. Medical professionals who use a patient portal app want to view medical images without worrying that confidential patient images could be compromised. Private sector employees who use an enterprise app want to look at their company's confidential animations without worrying that they could be leaked to their competitors. Although encryption provides a way to securely deliver images to a mobile device and store them, the "last mile" of on-device protection—securely decrypting and displaying an encrypted image—is still an open problem.

Fortunately, previous approaches [11, 33–36, 56, 61, 65, 68] have shown that by leveraging a Trusted Execution Environment (TEE) [9, 32, 53, 63], more specifically ARM TrustZone [63] for mobile devices, it is possible to securely display private or confidential images. In essence, these approaches share a common mechanism to enable such a capability—they use the secure execution environment provided by TrustZone (called *the secure world*), to which they give exclusive access for display hardware. This access control prevents the software running in the normal execution environment (called *the normal world*) from accessing the content the secure world displays. As a result, it protects the integrity and confidentiality of image data displayed by the secure world from the threats posed by the software running in the normal world.

However, this common mechanism has a critical limitation that, because the secure world has exclusive access to display hardware, it is not possible for the normal world to display anything *simultaneously* when the secure world displays its image data. This limitation significantly hinders the applicability of using TrustZone for display

security—for example, it would be impossible for the aforementioned medical app to simultaneously display new content in the normal world (e.g., an incoming notification, updates for already-displayed animations, etc.) while the secure world is displaying confidential animated images (e.g., a patient's fMRI animation). One can reduce the impact of this limitation by using a technique that existing approaches have used [33, 68], where the secure world displays its content over a static "screenshot" (a pixel-wise copy) of the content displayed by the normal world. However, this technique still does not allow the normal world to actively display anything on the screen since the secure world still has exclusive access to the display hardware. The technique only captures and freezes the normal world's content when the secure world accesses the display hardware.

We present `Rushmore`, a system that overcomes this limitation by leveraging a well-known, yet overlooked hardware feature called an IPU (Image Processing Unit) [15, 20, 45, 52]. On a typical system, there is a main display channel called the frame buffer, and writing pixels to the frame buffer drives the screen to display those pixels. An IPU provides additional display channels controlled by specialized cores that can process and display pixels simultaneously from different sources (e.g., a camera). `Rushmore` uses one IPU core and one of the additional channels to display pixels from the secure world *over* the pixels from the frame buffer. By giving the secure world exclusive access to the additional channel and the IPU core, `Rushmore` protects the integrity and confidentiality of the image data displayed by the secure world. The normal world, on the other hand, can continue displaying its content simultaneously since the secure world does not use the frame buffer. In addition, a user can interact with the normal world's content (e.g., UI elements) normally without requiring any extra mechanisms in the secure world.

Moreover, `Rushmore` can decrypt and display encrypted animated images in real time at around 30 FPS (Frames Per Second) [1] or higher, depending on the image size and the cryptographic method in use. We demonstrate this by implementing and comparing four cryptographic methods: (i) software AES (Advanced Encryption Standard) that performs AES decryption entirely in software, (ii) hardware AES that uses a hardware AES accelerator, (iii) a fast stream cipher called ChaCha20 [4, 8, 31, 41] implemented in software, and (iv) an image-based cryptographic technique called visual cryptography [14, 40, 49, 64, 66]. Among these, we show that ChaCha20 provides the best frame rates to display encrypted animated images with various sizes. Displaying encrypted animated images was previously not possible due to the high overhead associated with making a pixel-wise copy (a screenshot) of the normal world's content. `Rushmore`'s use of an IPU makes this capability possible.

Lastly, we show that `Rushmore` enables a novel application for visual cryptography. Visual cryptography encrypts and decrypts confidential image data by constructing two images in such a way that overlaying one image with the other would reveal a new image with the confidential image data. `Rushmore`'s use of two display channels where one channel overlays the other provides an ideal opportunity for visual cryptography. We show that our adaptation of visual cryptography not only is functional as an cryptographic method for `Rushmore` but also provides high performance—for static, black-and-white images, our results show that `Rushmore`'s visual cryptography outperforms ChaCha20.

Overall, `Rushmore` provides low latency for displaying static images (less than 36.0 ms with ChaCha20), usable frame rates for displaying animated images (around 30 FPS for large sizes and 30 FPS or higher for medium-to-small sizes), and low overhead to the applications running in the normal world (less than 5%) compared to a baseline that runs the same workload without using `Rushmore` (thus insecure).

[1] 24 FPS is the lowest frame rate that allows a human eye to naturally perceive motion. For example, movies are shot and displayed at 24 FPS. However, we aim for 30 FPS as it provides smoother viewing experience.

Table 1: Our survey of 14 chipsets and the availability of IPUs as well as hardware crypto accelerators on them from publicly-available manuals.

Chipset (Manufacturer)	Overlay	HW crypto
TDA2x ADAS (TI) [26]	yes	yes
OMAP35xx (TI) [23]	yes	yes
i.MX 6 (NXP)* [52]	yes	yes
i.MX 6ULL (NXP) [42]	yes	yes
i.MX 28 (NXP) [51]	yes	yes
MC i.MX 51 (NXP) [18]	yes	yes
i.MX 53 (NXP) [46]	yes	yes
i.MX 21 (NXP) [43]	yes	no
i.MX 25 (NXP) [44]	yes	no
JZ4760 (Ingenic Semiconductor) [17]	yes	no
MT6797 (MediaTek) [21]	yes	no
MediaWall V (RGB Spectrum) [20]	yes	no
PXA27x (Intel) [16]	yes	no
Nios II (Intel) [22]	no	no

2 BACKGROUND

This section presents the background necessary to understand `Rushmore`'s design and introduces the terminology that we use in this paper.

2.1 ARM TrustZone

`Rushmore` uses TrustZone [63] as the foundation of its security guarantees. TrustZone is widely deployed on mobile devices thanks to the popularity of ARM CPUs. It provides a secure execution environment called the secure world and a normal execution environment called the normal world. At any given moment, a CPU core executes in the context of either the secure world or the normal world. ARM provides a privileged instruction called SMC (Secure Monitor Call) for CPU cores to make a switch between the secure world and the normal world. This operation is called a *world switch*. All hardware components, e.g., memory and I/O devices, can be configured via TZASC (TrustZone Address Space Controller) so that access is granted to a CPU core only when the CPU core executes in the secure world. This hardware-based isolation of the secure world from the normal world is the basis of the security guarantees of TrustZone.

2.2 Image Processing Unit

In addition to TrustZone, `Rushmore` uses another hardware feature called an IPU (Image Processing Unit). An IPU provides specialized

Table 2: Our survey of 5 chipsets and the availability of IPUs on them from the Linux kernel source code.

Chipset (Manufacturer)	Overlay
Exynos (Samsung) [25]	yes
MSM (Qualcomm) [24]	yes
OMAP (TI) [27]	yes
Mediatek (Mediatek) [19]	yes
Tegra (Nvidia)	no

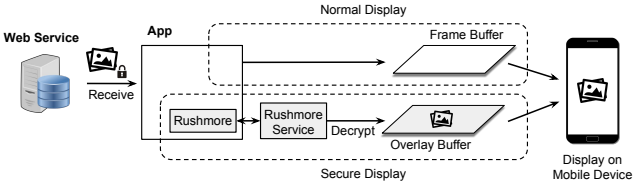


Figure 1: Typical workflow of an application to display an encrypted image using `Rushmore`.

cores and display channels that can process and display pixels coming from various sources such as a camera. The presence and the implementation of an IPU for a mobile device depends on the SoC (System-on-a-Chip) that the device uses. A well-known example is Google's Pixel Visual Core [47], which is an IPU used by Google's Pixel 2 and 3 phones.

An IPU can provide different types of display channels in addition to the main display channel (the frame buffer). However, the most basic channel is often referred to as an *overlay buffer*. Pixels in the overlay buffer are displayed over the pixels coming from the frame buffer. `Rushmore` uses this overlay buffer to display pixels from the secure world over the pixels from the normal world.

Since `Rushmore` relies on the capability of overlaying pixels, we have surveyed publicly available mobile chipset manuals to determine whether mobile chipsets provide such a capability via an IPU. We have investigated 14 chipsets as shown in Table 1 and found that every chipset except Intel Nois II provides an overlay capability via their IPU. Table 1 also shows the availability of a hardware crypto accelerator for the same chipsets as a comparison point, and only a half of the chipsets have it. As shown in Table 2, we have further investigated the Linux kernel source code of five commercially successful chipset families which lack publicly-available manuals—Samsung's Exynos, Qualcomm's MSM, TI's OMAP, Nvidia's Tegra, and Mediatek. We have found a pixel overlay feature from all of those chipsets' kernel code except Nvidia's Tegra. Our results indicate that an IPU with an overlay capability is a common feature available for many chipsets that mobile devices use. Furthermore, recent neural processing units from major mobile device vendors, e.g., Apple's Neural Engine [54] and Samsung's NPU [50], provide image processing capabilities. Although details are not publicly available, we expect that they also provide additional display channels that are paired with their image processing capabilities since competing for the frame buffer causes unnecessary difficulty for software that uses the capabilities.

2.3 Threat Model

As with other TrustZone-based systems, we assume that (i) device hardware including TrustZone is trusted, (ii) software in the normal world (including the OS and user applications) is untrusted, and (iii) software in the secure world is trusted. This applies to our own software as well, i.e., `Rushmore`'s normal world components are untrusted while secure world components are trusted. In addition, we assume that our adversaries are only interested in compromising the confidentiality of the images that we protect. Thus, we consider denial-of-service (where the normal world refuses to switch to the secure world) out of scope. Lastly, we consider the following two classes of attacks out of scope as they require specialized solutions—(i) covert or side-channel attacks [6, 7, 28, 30, 67] and (ii) physical attacks such as "shoulder surfing" [5] where an attacker peeks at the screen of a user in order to compromise images displayed by the screen.

3 RUSHMORE USAGE MODEL

`Rushmore` assumes a particular usage model as follows.

Deployment Model: Following the standard deployment model of Trusted Execution Environments (TEEs), we assume that a single party (e.g., a smartphone OEM) packages and deploys all components of `Rushmore` on a mobile device as a single secure world kernel image. The party that deploys `Rushmore` can develop and deploy their own custom services that display confidential images using the interface provided by `Rushmore`, as discussed in Section 4.3. A `Rushmore` kernel image is signed and it is verified by the bootloader during a booting process.

Key Distribution: The main functionality provided by `Rushmore` is decrypting and displaying encrypted images sent by a trusted party. Thus, we assume that images have already been encrypted when `Rushmore` receives them. `Rushmore` does not mandate any particular key distribution mechanism. It is up to each party that deploys `Rushmore` to ensure that their key distribution mechanism is safe and secure. For example, one cryptographic algorithm we implement in `Rushmore` is AES, and we assume that a secret key is shared through a separate mechanism such as pre-installation of a device key by an OEM.

Image Format: `Rushmore` supports images in the bitmap format with 16-bit RGB (RGB565). RGB565 is commonly used for embedded devices, with 6 bits for green and 5 bits each for blue and red. This choice is mainly due to performance—the bitmap format avoids format decoding and RGB565 (instead of RGB888) reduces the overhead for memory copy operations. Thus, `Rushmore` requires images in other formats, e.g., JPEG, to be converted to the RGB565 bitmap format before encryption. We further discuss the implication and limitations of this choice in Section 7.

Workflow: Figure 1 shows an example workflow of an app using `Rushmore`. Suppose that a user has an app on her mobile device that can display confidential images sent by a trusted web service. The web service and a `Rushmore` service authenticate each other with their certificates on each side. Then they exchange a symmetric key through an authenticated channel. The web service encrypts an image with the key and sends it to the app. The app receives the encrypted image and requests `Rushmore` to securely decrypt and display the encrypted image. `Rushmore` first decrypts the image and

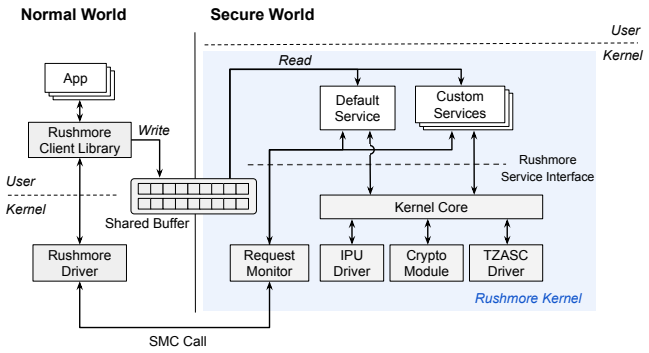


Figure 2: Overview of `Rushmore` architecture.

uses the overlay buffer on the device to display the image on top of what the frame buffer displays in the normal world. In other words, the normal world and the secure world use two separate display channels—the frame buffer (used by the normal world) and the overlay buffer (used by the secure world). Therefore, `Rushmore` does not need to make a pixel-wise copy of the normal world's content to display it unlike a technique used by previous systems [33, 68].

4 RUSHMORE ARCHITECTURE

In this section, we present the details of `Rushmore`. We start by presenting the overall architecture, followed by a description for each component in the architecture.

4.1 Architecture Overview

Figure 2 shows an overview of the `Rushmore` architecture. In the normal world, our `Rushmore` client library provides an interface for applications to send encrypted images to the secure world. Internally, the client library communicates with the `Rushmore` driver in the normal world kernel to pass encrypted images and make a world switch (using TrustZone's SMC instruction). There are two shared buffers between the normal world and the secure world. One shared buffer is for sending encrypted images from the normal world to the secure world. The other shared buffer is for passing meta data and other arguments. A naïve implementation would copy encrypted image data from the user space to the kernel space in the normal world, and copy again from the normal world kernel space to the secure world kernel space. Thus, it would result in repeated memory copy operations. We bypass both these copies by creating a shared buffer in the contiguous memory allocation area and provide access to this shared buffer to both the user space and the kernel space in the normal world. This is done by modifying the memory mapping in the kernels for both the normal world and the secure world.

In the secure world, the `Rushmore` kernel mainly provides decryption and display capabilities as well as an interface for services that manipulate and display confidential images. For decryption, `Rushmore` has drivers that implement various cryptographic alternatives. Currently, we implement and evaluate four methods—software Advanced Encryption Standard (AES) that performs the entire AES function in software, hardware AES that offloads this function to a hardware accelerator, a fast stream cipher called ChaCha20 [4, 8, 31, 41] that is part of the TLS 1.2 standard [31] and used by Google Chrome on Android [13], and an image-based cryptographic method called visual cryptography (VC) (described in Section 4.4). For display capabilities, `Rushmore` has an IPU driver that controls the overlay buffer. Using the overlay buffer, `Rushmore` displays pixels from the secure world *over* the pixels from the normal world.

Table 3: Essential subset of the `Rushmore` client library interface for client applications.

Function Name	Description
int `invoke(...)`	Invokes a particular `Rushmore` service in the `Rushmore` kernel
int `display_images(...)`	Requests the `Rushmore` kernel to display encrypted images
int `display_animations(...)`	Requests the `Rushmore` kernel to display animated images
int `remove_images(...)`	Requests the `Rushmore` kernel to remove images from the screen.

`Rushmore` kernel also contains services that manipulate and display confidential images, which we call *Rushmore Services*. By default, `Rushmore` kernel runs a service that displays encrypted images passed from the normal world and returns immediately. However, an entity that deploys `Rushmore` can develop their own custom image display services to provide extra functionality that goes beyond simple image displaying. Our use cases in Section 5 show some examples of such services. In order to distinguish different `Rushmore` services, `Rushmore` assigns a unique identifier (UID) to each `Rushmore` service. `Rushmore` assumes that these UIDs are published publicly so that mobile applications can send requests to different services. When sending a request to a particular service, a mobile application uses the service's UID to identify the service. *Request Monitor* in the `Rushmore` kernel then receives this request, looks up the UID, and forwards it to the appropriate service. This service model follows a standard practice that most Trusted Execution Environments (TEEs) use, e.g., Trusty [60], OP-TEE [57], Knox [59], QSEE [58], etc. As mentioned in Section 3, `Rushmore` does not mandate any particular key distribution mechanism. We expect each `Rushmore` service to implement its own key distribution mechanism suitable for itself.

Additionally, `Rushmore` kernel has a TZASC (TrustZone Address Space Controller) driver. TZASC allows us to control the access permissions for memory regions, and the `Rushmore` kernel core uses the TZASC driver to configure the memory layout and gives exclusive permission to the secure world for the overlay buffer, IPU registers, and hardware AES accelerator registers.

4.2 Rushmore Client Library and Driver

`Rushmore` has two components that run in the normal world—the `Rushmore` client library and the `Rushmore` driver.

Rushmore Client Library: The `Rushmore` client library exposes `Rushmore`'s features to the normal world's applications. These include (i) the ability to invoke `Rushmore` services, (ii) the ability to display images securely, (iii) the ability to display animated images securely, and (iv) the ability to remove displayed images. Internally, the client library uses the `Rushmore` driver in the normal world kernel to relay a request to the secure world via the SMC instruction of ARM TrustZone.

Table 4: Essential subset of the `Rushmore` service interface.

Function Name	Description
int `decrypt_images(...)`	Requests the kernel core to decrypt images
int `display_images(...)`	Requests the kernel core to display images
int `decrypt_and_display(...)`	Requests the kernel core to decrypt and display encrypted images
int `remove_images(...)`	Requests the kernel core to remove images from the screen.
int `invoke(...)`	A callback function that responds to a client application's `invoke()` call
int `display_images(...)`	A callback function that responds to a client application's request

Table 3 shows essential functions that the library provides. `invoke()` is used to invoke a service other than the default service. Other functions are used to invoke the default service to either display static or animated images or remove images that are currently being displayed. Although we do not show the parameters of the functions in Table 3, a client application needs to pass an appropriate set of arguments for each function. For example, the default service expects to receive encrypted images, the number of images, and their locations.

A client application can also display one or more encrypted animated images using the default service. In order to display a single animated image, a client application must pass a set of encrypted images (frames) that constitute an animated image as well as the location to display and a target FPS (using `display_animations()`). A client can pass multiple sets of images to our library to display multiple animated images simultaneously.

Rushmore Driver: When displaying an encrypted animated image, the `Rushmore` client library makes a request to the `Rushmore` kernel for each frame, which triggers the `Rushmore` kernel to decrypt and display one frame at a time. The implication of this design is that we need to make a world switch from the secure world to the normal world every time the `Rushmore` kernel finishes displaying a frame, so that our library can make another request for the next frame. An alternative design would be to pass all frames together at once so that the `Rushmore` kernel could display them in succession without switching back and forth between the normal world and the secure world. However, this alternative design has an inherent limitation that we would need to load all the frames into memory so that the `Rushmore` kernel could access them. In other words, we would always need to have enough memory to load all the frames, no matter how large a frame is and how many frames there are. In contrast, our design choice only has a minimal memory requirement since we pass one frame at a time. In our implementation, we load the next *N* frames (instead of just the next frame) to the buffer in the background while displaying in the secure world to hide memory copy latency. Section 6 shows that this design requires minimal memory while still providing low latency.

We note that the `Rushmore` client library's guarantee for a target FPS is best effort—this means that if a client application sets an aggressive target FPS, `Rushmore` may not satisfy it. Also, some cryptographic methods are inherently expensive as we show in our evaluation (Section 6), and it may be challenging to provide the requisite FPS on a given platform when we use them.

4.3 Rushmore Kernel

The `Rushmore` kernel has five sub-components that collectively provide the main functionality. These sub-components are the default `Rushmore` service, the kernel core, the cryptography module, the IPU driver, and the request monitor.

Default Rushmore Service: The default `Rushmore` service is a simple service that responds to image decryption and display requests. It receives encrypted images from client applications running in the normal world, decrypts them, and displays them. Internally, it uses the development interface provided by the kernel core, which we describe next.

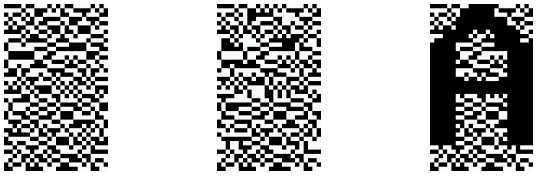
Kernel Core: The `Rushmore` kernel core has two roles. First, it handles the booting and configuration process of `Rushmore`. It uses the TZASC driver to partition the memory between the normal world and the secure world, and assign the access permissions for memory regions including the memory-mapped regions for the overlay buffer, IPU registers, and hardware AES accelerator registers.

Second, the kernel core (together with the request monitor) provides an interface for developing image display services that we call `Rushmore` services. The default `Rushmore` service uses this interface to provide the basic decryption and display functionality. In addition, organizations that deploy `Rushmore` on mobile devices can use the interface to develop and deploy their own services. Table 4 shows an essential subset of this interface. The first four functions are provided by the kernel core and mainly for decrypting and displaying encrypted images. These functions interact with the cryptography module and the IPU driver. Using these functions, a `Rushmore` service can handle client application requests and use `Rushmore`'s decryption and display functionality.

Cryptography Module: To decrypt an encrypted image, the `Rushmore` kernel uses a pluggable cryptography module. Our current implementation has four cryptographic methods—software AES, hardware AES, ChaCha20, and visual cryptography. Since we adapt and optimize visual cryptography in `Rushmore`, we discuss the details in Section 4.4. For software AES, we use an implementation from OP-TEE [57]. For ChaCha20, we use the Network Security Services (NSS) library [62]. For hardware AES, we use a driver for the hardware accelerator module present in the development board that we use (Nitrogen6Q SABRE Lite) called CAAM (Cryptographic Accelerator and Assurance Module). This design can be easily extended to other boards with cryptographic accelerators by using a driver for that accelerator.

IPU Driver: To display decrypted images, the `Rushmore` kernel uses an IPU driver. This driver controls the overlay buffer and copies image pixels into the overlay buffer according to their locations. The pixels in the overlay buffer essentially overwrites the pixels in the frame buffer (controlled by the normal world kernel). Thus, unless a pixel is transparent, it hides the pixel underneath displayed by the frame buffer. The IPU driver has exclusive access to the IPU, protecting the image data displayed by the IPU. Many chipsets provide two or more cores and multiple overlay channels in their IPU and `Rushmore` requires exclusive access to only one IPU core as well as the top-most overlay channel. `Rushmore` leaves other IPU cores and channels available for the normal world.

Request Monitor: The request monitor is a small component that receives a request from a client application, finds the correct `Rushmore`



(a) Key image (b) Encrypted image (c) Decrypted (revealed) image

Figure 3: Examples images for visual cryptography.

service for the request, and invokes a callback function implemented by the `Rushmore` service (the last two functions in Table 4). It also makes a world switch back to the normal world once the callback function is complete.

4.4 Visual Cryptography

A notable cryptographic method that `Rushmore` supports is visual cryptography (VC) [14, 40, 49, 64, 66]. We believe `Rushmore`'s use of the overlay buffer presents a compelling application of VC. `Rushmore` supports VC mainly as a light-weight cryptography option, e.g., for low-end devices without hardware cryptography support or powerful CPU cores.

Overview: VC encrypts and decrypts images by leveraging the insight that we can construct two images in such a way that they "reveal" a new image when we overlay one on top of the other. Figure 3 shows an example where overlaying one image (Figure 3a) on another image (Figure 3b) reveals a new image (Figure 3c). The example uses two colors, transparent and black. The image in Figure 3a has randomly distributed transparent and black pixels. We call this a *key image* since it effectively works as a shared secret key that is difficult for an adversary to guess. We then construct the other image (Figure 3b) by calculating pixel-wise XOR between the key image and the image that needs to be encrypted (more details below). We call this an *encrypted image* as it encodes our image data but is not decryptable without the key image. Finally, by overlaying the key image with the encrypted image, we construct the final image that contains the actual image data. We also call this a *decrypted image*.

A nice property that our implementation of VC provides is that a decrypted image does not exist in any part of the software system, even in the secure world. The decrypted image is only visible to a user. In our example shown in Figure 3c, the decrypted image reveals the alphabet `A`, the image we encode, as well as some background noise inherited from the key image. We later discuss how we minimize the background noise.

Rushmore's VC: To understand how we adapt and implement VC in `Rushmore`, consider a scenario where a server wants to send an encrypted image to a mobile device that runs `Rushmore`. `Rushmore` requires that both the server and the mobile device use the same pseudo-random number generator (PRNG) and the same seed, where the seed is a shared secret between the two parties. When encrypting, the server first constructs a key image by (i) generating a random bitstream using the PRNG (and the seed), and (ii) mapping the 0s and 1s in the bitstream to transparent and black pixels. The server then constructs an encrypted image by calculating pixel-wise XOR between the key image and the image that needs to be encrypted.

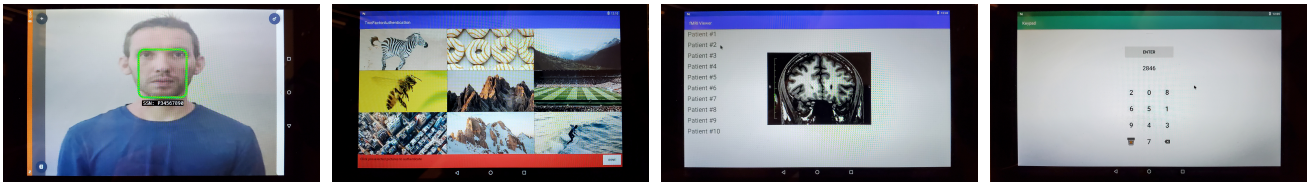
(a) Decrypted pixel-wise VC image (b) Decrypted pair-wise VC image

Figure 4: Decrypted 454×418 VC images with different key generation schemes. `Rushmore` uses a pair-wise random key generation scheme that looks less noisy.

When decrypting, the `Rushmore` client library running in the normal world displays an encrypted image directly on the normal world frame buffer. At the same time, it sends a request to the `Rushmore` kernel running in the secure world to display the corresponding key image on the secure world overlay buffer at the same position as the encrypted image. When the `Rushmore` kernel receives the request, it uses the same PRNG and the seed used by the server to generate the corresponding key image (i.e., generating a random bitstream and mapping the 0s and 1s in the bitstream to transparent and black pixels). It then displays the key image on the overlay buffer in the secure world. This process overlays the key image and the encrypted image (effectively calculating pixel-wise OR), which reveals the original image.

Although VC supports color images in general, we only support black and white images in `Rushmore` currently. This is because we discovered that the decryption performance of VC for images that support 8 colors is similar to that of ChaCha20, which supports 16-bit color images in our current implementation. In addition, we currently do not support animated images with VC. This is due to synchronization—in order to decrypt an image, VC needs to display the key image and the encrypted image exactly at the same time. If the timings do not align, there is a brief period where VC shows a random image with noise. With a static image, this is tolerable and mostly unnoticeable since it is brief. However, with an animated image, we have observed that this occurs at every frame transition and appears quite noisy as a whole. Due to this difficulty, we only support static images for VC. Our future research will investigate how to support animated images. Despite these limitations, VC outperforms other cryptographic methods for static, black-and-white images. Thus, it is the best choice for those images. We show the performance of VC later in Section 6.

Security Analysis: From the cryptographic point of view, VC is a stream cipher representing data in pixels instead of bits [40]. A stream cipher (e.g., ChaCha20) encrypts data by XOR-ing the data with a cryptographically secure random key bitstream. VC's encryption works almost exactly the same way. When encrypting data, VC calculates pixel-wise XOR between a key image and an image being encrypted, effectively swapping the transparent and black color of pixels whenever data encoding is necessary. When decrypting an encrypted image, VC overlays an encrypted image with a key image, effectively calculating pixel-wise OR. Therefore, the security of VC relies on the security of the key bitstream, just as all other stream ciphers do. `Rushmore` currently uses ChaCha20 algorithm for its



(a) Face recognition (b) Two-factor authentication (c) fMRI viewer (d) Randomized keypad

Figure 5: Screenshots of use case apps.

VC random key generation, which is cryptographically secure and already part of TLS 1.2 standard [31].

Minimizing Background Noise: As shown in Figure 3c, decrypted images in VC have background noise. This is because when we construct a key image, we generate a random bitstream and assign transparent and black colors for 0s and 1s from the bitstream. The resulting image has random black pixels appearing as background noise. Figure 4a shows an example decrypted image that appears quite noisy.

To reduce such background noise, we implement the following mechanism. When generating a key image, instead of doing color assignment for a single pixel, we perform color assignment for *a pair of pixels* in such a way that we make the background appear less noisy. In this mechanism, we first pick either transparent or black color for one pixel based on a random bitstream, and then for the pixel right below it, we assign either black (if the pixel above is transparent) or transparent (if the pixel above is black). Note that the size of a random bitstream is now a half of the original method's since one bit from a bitstream determines the colors for two pixels. Figure 4b shows an example decrypted image generated by our pair-wise key generation scheme. Since any part of the key has the same number of black and transparent pixels, the pair-wise random background looks more uniform and less noisy.

Optimization for VC in Rushmore: In our first implementation of VC, copying one color value pixel-by-pixel did not perform well. Thus, we replaced it with an optimized strategy that significantly improved the performance. The optimized strategy is a batch-assignment mechanism with a pre-calculated lookup table. This strategy assigns color values for 16 pixels at a time instead of a single pixel. We heuristically determined that 16 pixels works well, but our strategy does not generally depend on this number. For this strategy, we first pre-populate a table that maps a 16-bit number to a 16-pixel color assignment for every possible 16-bit number. When we read a random bitstream for key image generation, we read 16 bits at a time and look up the table to determine the corresponding color assignment for 16 pixels. In our experiment, the use of this table has accelerated the color assignment 21× improvement with 2 MiB (= $2\ bytes\text{-}per\text{-}pixel \cdot 16\ pixels \cdot 2^{16}\ rows$) memory overhead. We further optimized memory copy with ARM's single-instruction-multiple-data (SIMD) instruction set, Neon, and achieved 119× improvement from the original implementation.

4.5 Implementation

We have implemented `Rushmore` using OP-TEE [57], an open-source TEE OS. As we detail in Section 6, we have added and modified roughly 2.5K lines of code in total. In addition, we have implemented Android UI widgets that use our `Rushmore` client library, so that Android apps can seamlessly leverage `Rushmore`. Our UI widgets include buttons, image holders, and animation holders. We use these UI widgets in our use case apps that we present next.

5 USE CASES

Using `Rushmore`, we have implemented four representative use cases that require secure displaying of sensitive images. We show their screenshots in Figure 5. These use cases demonstrate the utility of `Rushmore` in deploying secure widgets in a wide range of apps on mobile devices.

5.1 Face Recognition

Modern mobile devices such as smart glasses are envisioned to be used by law enforcement in public areas such as airports, malls and sports arenas to recognize people of interest and display sensitive information about them such as names, passport number, etc. The threat model here is that attackers are interested in obtaining the sensitive information associated with the people of interest by compromising the normal world software on the mobile device. We note that the focus of this use case is *not* about protecting the faces of people of interest. Rather, it is about protecting the sensitive information regarding the people of interest, e.g., their identities. The reason is that the use case's scenarios assume public areas where everybody is openly visible and law enforcement routinely has access to security cameras already. Protecting their identities, on the other hand, is a concern.

To demonstrate this use case, we have implemented a normal world app and a `Rushmore` service. The normal world app runs a face recognition engine using Google's `ML Kit` [39] and maintains a face database. Each entry in the database is a mapping between a face of a person of interest and their anonymized ID number. When the app detects a face of a person of interest, it looks up the ID for the person in the database and sends the ID as well as the position of the face to the `Rushmore` service. When the `Rushmore` service receives an ID from the normal world app, it looks up the sensitive information for the received ID using a database it maintains, such as the name or passport number associated with the ID. The `Rushmore` service then displays the sensitive information right next to the recognized face using the face position information sent by the normal world app. `Rushmore` is particularly a good fit for this use case as the latency of displaying sensitive information is an important concern—if a person moves, the sensitive information displayed from the secure world should quickly "follow" the person.

5.2 Two-Factor Authentication

In a traditional two-factor authentication service, when a user logs in to a web service, the web service sends a push notification to the user's mobile device and the user needs to approve the login request by pressing an approval button. In our use case, we augment this by asking a user to identify one or more images that the user has pre-selected, instead of simply pressing an approval button. This means that when a user logs in to a web service, the web service sends a series of images (some are pre-selected by the user while others are not), which the user's mobile device displays. The threat model here is that attackers are interested in learning the user's pre-selected images. To protect those images, the web service needs to encrypt the images and the user's mobile device must display them securely.

To demonstrate this use case, we have implemented a normal world app as well as a `Rushmore` service. The normal world app receives a series of encrypted images and their display positions from a web service, and sends them to the `Rushmore` service. The `Rushmore` service receives the encrypted images and their positions, and then decrypts and displays them. We use nine images to display in a 3×3 grid. When a user identifies one or more images, the normal world app receives user input and sends it to the (emulated) web service for verification. Although the normal world is untrusted and hence might leak the user input, we still protect the user's pre-selected images as the secure world decrypts and displays them.

5.3 fMRI Animation Viewer

An fMRI animation viewer could use `Rushmore` to display a patient's animated fMRI that shows the changes in blood flow for brain activities. Medical images are of a confidential and sensitive nature where privacy and data security are of great concern. Thus, we have implemented this use case using `Rushmore` to securely display fMRI images.

Our implementation consists of a normal world app and a `Rushmore` service. The normal world app works like a typical image viewer except that the images are fMRI animations for patients. When a user (e.g., a doctor) selects an fMRI animation for viewing, the normal world app retrieves and passes the encrypted animation to the `Rushmore` service. The `Rushmore` service then decrypts and displays the animation. By using `Rushmore`, the normal world only sees the encrypted data. This preserves the privacy and confidentiality of the animation and medical image data in the event that the security of the mobile device's normal world is compromised. Frame rates matter in this use case to show brain activities accurately, and `Rushmore` can provide 30 FPS or higher as we show in Section 6.

5.4 Randomized Keypad

Several apps such as banking, online shopping and others require the user to enter sensitive numerical information (such as a pin or a credit card number) for authentication/use. In such cases, the app provides a virtual keypad on the screen to enter this numerical information. We target these apps by implementing a randomized keypad using `Rushmore`. This service securely displays a keypad where the locations of the keys are randomized every time the keypad is displayed.

We demonstrate this use case by implementing a normal world app and a `Rushmore` service. The normal world app receives encrypted randomized positions of keys from a web service and sends those to the `Rushmore` service. The `Rushmore` service then decrypts the positions and displays keys according to the positions. When a user provides input, the normal world app receives it and sends it to the backend for verification. This does not compromise the sensitive information being entered as we randomize the keypad every time we display it.

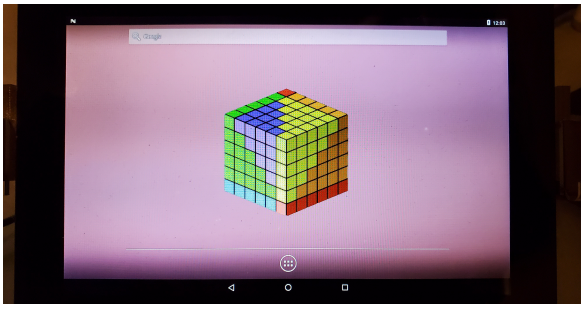


Figure 6: 400×400 **cube image on a** 1280×800 **display.**

6 EVALUATION

We evaluate `Rushmore`'s capabilities in three categories: (1) performance for static images, (2) performance for animated images, and (3) Trusted Code Base (TCB) size. We also present our security analysis for `Rushmore`.

For our experiments, we use the Boundary Devices Nitrogen6Q SABRE Lite development board (ARM Cortex-A9 quad-core CPU at 1GHz and 1GB memory) and a 1280×800 display. In the normal world, we run Android Nougat 7.1.1 with the Linux kernel version 4.1.15 and run `Rushmore` kernel implemented using OP-TEE [57]. `Rushmore` restricts access to the IPU and the hardware AES accelerator (called CAAM, Cryptographic Accelerator and Assurance Module) from the normal world, and the normal world has control and access to all the other peripherals.

We allocate 50MB of memory for the secure world and 974MB for the normal world. In the secure world, we implemented four different cryptographic methods; software AES, hardware AES, ChaCha20, and black-and-white visual cryptography (VC). For both software AES and hardware AES, we use AES-128 CBC mode (key: 32 bytes, iv: 16 bytes), and we use our board's CAAM in the asynchronous mode for hardware AES. We also use the overlay channel of our board's IPU to display in the secure world.

6.1 Image Display Performance

To evaluate end-to-end latency, we measure the delay of displaying four different cube image sizes: (1) 200×200, (2) 400×400, (3) 800×800 (larger than half of the display), and (4) 1280×800 (the whole screen). Figure 6 shows the 400×400 cube image as an example. For each size, we have 1,000 images encrypted and stored. We retrieve each image from storage, decrypt them using one of the four methods above, and display them on the framebuffer.

Figure 7 shows the end-to-end display latency breakdown of each image size. For all decryption methods except VC, the latency consists of the following four components: (1) `Memcpy:` latency for copying encrypted images to the shared buffer, (2) `Argument Copying:` latency for copying metadata arguments (e.g., image

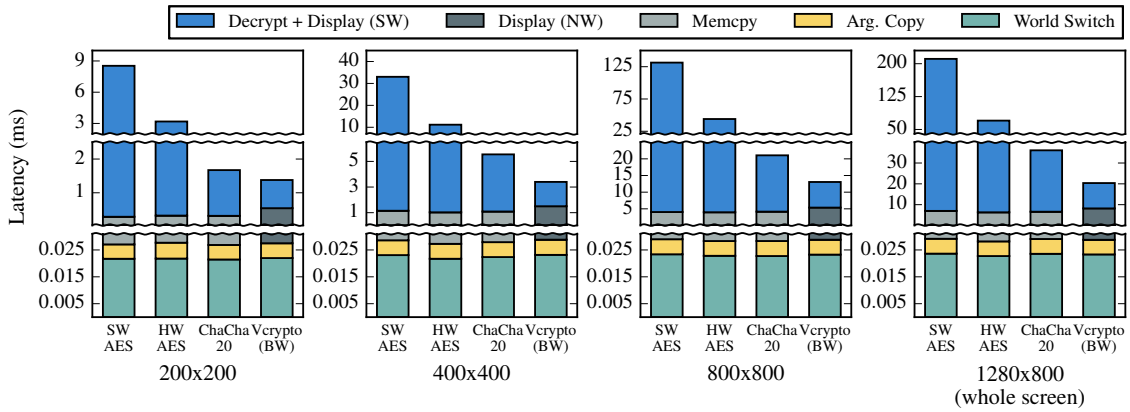


Figure 7: Latency breakdown for displaying a single image.

metadata) to the shared buffer, (3) `World Switch`: latency for the world switch from the normal world to the secure world, and (4) `Decrypt+Display (SW)`: latency for decrypting and displaying an image in the secure world.

For VC, image copying is unnecessary as the normal world directly displays an encrypted image. Instead, it has the latency of displaying an encrypted image in the normal world, which we indicate as `Display (NW)`.

`World Switch` and `Argument Passing` are stable across four cryptographic methods and all different image sizes, and they take less than 0.03 ms. The reason is that the world switch latency is consistent and an image size does not affect the latency of argument passing that only contains meta data. `Memcpy`, `Display (NW)` and `Decrypt+Display (SW)` increase when image sizes get larger as the memory size to decrypt and the amount of write on the frame buffer grow. `Memcpy` and `Display (NW)` are stable across all cryptographic methods as long as the image size is the same as both operations simply copy an image either to the shared `Rushmore` buffer or to the frame buffer.

The main latency overhead comes from `Decrypt+Display (SW)`. Even for the same sized image, the latency varies across different cryptographic methods. Decryption is a key factor in reducing the latency. On average, ChaCha20 is 6.9 times faster than software AES and 2.2 times faster than hardware AES. VC is 2.1 times faster than ChaCha20 for black-and-white images. Overall, VC shows the best performance in terms of display latency even though it only supports black-and-white images with background noise. Thus, VC would be a good option for low-end devices without a powerful CPU when trading off image quality is permissible. For colored images, ChaCha20 shows the best performance in terms of display latency.

6.2 Animation Display Performance

We now show the frame rates for simultaneously displaying 400×400 cube animations. We vary the number of animations we display from one to five. A single cube animation consists of 100 different frames. For each experiment, we display 1,000 times with software AES, hardware AES, and ChaCha20. As discussed in Section 4.4, our VC currently does not support animated images. We set the size of the shared `Rushmore` buffer between the normal world and the

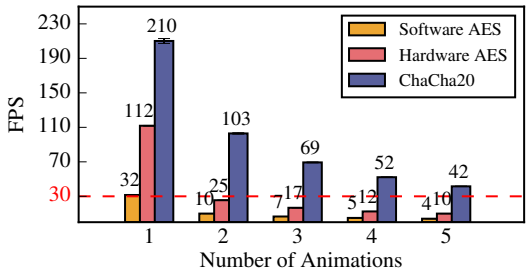


Figure 8: Frame rates for multiple animations.

secure world to 20 MB that can hold at least 10 whole screen size images. We also evaluate the impact of this buffer size. We have posted a video [2] that demonstrates animation display performance with ChaCha20.

FPS for Animations: Figure 8 shows the frame rates with different cryptographic methods. ChaCha20 provides more than 30 FPS regardless of how many concurrent animations we display. Software AES and hardware AES provide 30 FPS only when we display a single animation.

A notable result is that going from one animation to multiple animations creates a non-linear decrease in frame rates. This is an artifact of our IPU's programmability—for a single image, we can set the overlay buffer to cover the exact region where we want to display the image. However, for multiple images, we need to set the overlay buffer to cover a rectangular region that contains all the images. This means that going from one animation to multiple animations does not linearly increase the amount of write on the overlay buffer. This non-linear scaling affects the performance of all three methods, although the exact behavior is different from method to method.

Impact of the Shared Buffer Size: `Rushmore` allocates a buffer shared between the normal world and the secure world. To hide memory copy latency, `Rushmore` copies the next few frames to the buffer in the background while displaying in the secure world. To quantify the impact of the size of the shared buffer, we vary the buffer size from 512 KB to 16 MB and measure the frame rates for

[2] https://youtu.be/InkzkvpGHdU

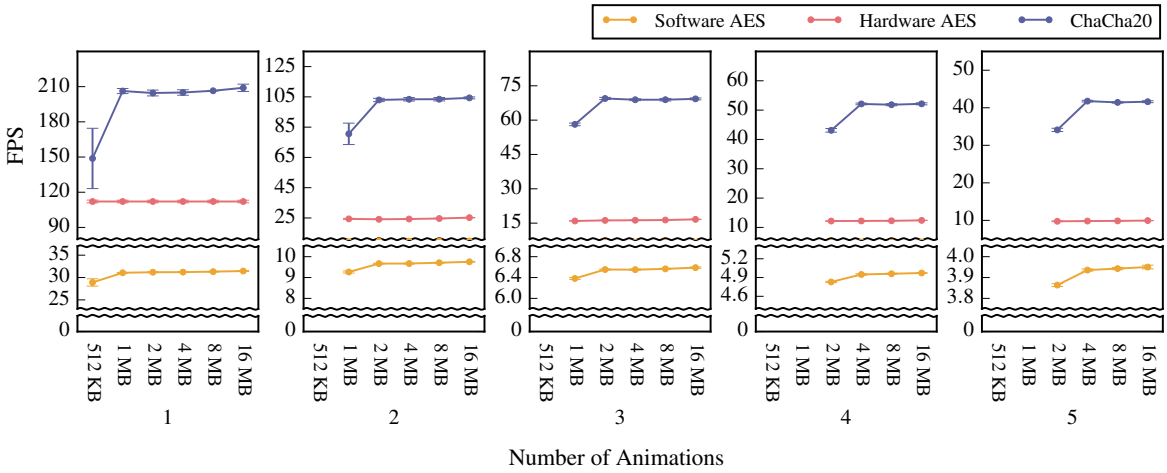


Figure 9: Frame rates for different buffer sizes.

simultaneously displaying 400×400 cube animations, varied from one to five animations. We run each experiment 100 times. For each setup of our experiments, some of the buffer sizes are not sufficient and we do not run any experiment for those cases. This is because the size of one frame for a single 400×400 animation is 312.5 KB, which means that if we display x animations simultaneously, displaying one frame for each animations requires $x \times 312.5$ KB. For example, in the case of two animations, 512 KB is not sufficient to display even one frame from two animations simultaneously, and we thus do not run experiments for that case.

As shown in Figure 9, for software AES and ChaCha20, the minimum buffer sizes that can hold just one frame (or one set of frames for multiple animations) have the worst frame rates. However, all other buffer sizes that can hold more than one frame (or more than one set of frames for multiple animations) show stable performance with less than 1% difference. This shows that as long as the shared buffer can hold at least two frames (or two sets of frames for multiple animations) for software AES and ChaCha20, it will not have much impact on the performance.

On the other hand, hardware AES's frame rates are stable for all buffer sizes because (i) it runs asynchronously in the secure world and switches back to the normal world immediately after putting a decryption request in the job queue, (ii) this asynchronous mode of operations enables the normal world to immediately buffer a new image, and (iii) our hardware AES decryption latency is larger than our image buffering latency. This means that no matter how many images we buffer, the hardware AES can only decrypt one image at a time.

Benchmark App Performance: To quantify how `Rushmore` affects other workload running in the normal world, we use an Android benchmark app called PassMark [55] available on Google Play. PassMark has 26 benchmarks in five categories; CPU, memory, disk, 2D graphics, and 3D graphics. It provides an overall system score as well as a score for each category. A higher score means better performance.

We run PassMark in the normal world along with our 400×400 cube animation workload displayed using `Rushmore`. As a baseline

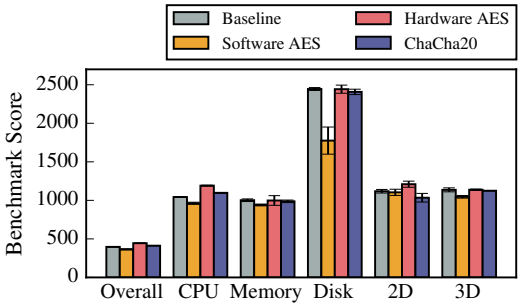


Figure 10: Benchmark scores for displaying a single animation.

comparison point, we also run PassMark with essentially the same workload (decrypting and displaying 400×400 cube animations) that runs everything in the normal world without using `Rushmore`. In this baseline workload, we use ChaCha20 for decryption as it is the fastest among the three cryptographic methods we use for animated images in `Rushmore`. We run ChaCha20 in the user space and display decrypted pixels using Linux's `/dev/fb*` interface. In all cases, we fix the frame rate at 30 FPS. We have posted a video [3] that demonstrates how PassMark works with our cube animation displayed from the secure world.

Since all cryptographic methods can support 30 FPS for a single animation (as shown in Figure 8), we first show the results of running PassMark while displaying a single animation in Figure 10. Software AES has noticeably lower performance than cryptographic methods in all benchmark categories, especially for the disk category. Since we do not know the internals of PassMark, it is not possible for us to determine the exact cause of this behavior. However, we hypothesize that software AES's heavy computation in the secure world prevents the CPU from making timely I/O requests to the disk. Hardware AES performs the best in most categories because it runs asynchronously and uses a separate piece of hardware, hence does not affect the benchmark app's performance much. Except for software AES, the

[3] https://youtu.be/JxeHUhHJH7k

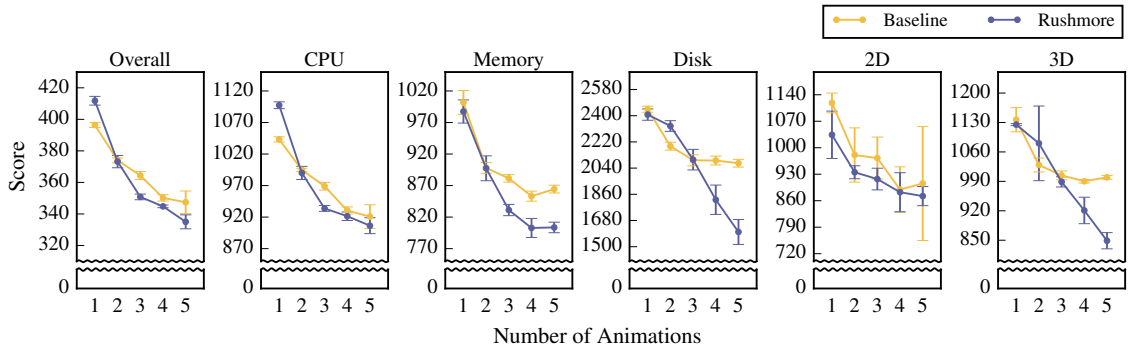


Figure 11: Benchmark scores for displaying multiple animations.

Table 5: TCB size (`Rushmore` kernel).

Type	LOC
Request Monitor	0.1K
Kernel Core	0.5K
IPU Driver	0.5K
Cryptography Module	1.3K
Default Service	0.1K
Total:	**2.5K**

overall score difference between the baseline and other cryptographic methods is less than 5%.

For ChaCha20, we run PassMark for all cases as it can always support 30 FPS. As shown in Figure 11, benchmark scores of 2D and 3D categories have high standard deviations, which indicates that `Rushmore` highly affects the performance of graphics categories. Also, the overall score for ChaCha20 is (i) higher than the baseline for a single animation, (ii) similar for 2 animations, and (iii) lower for 3-5 animations. Again, without access to the internals of PassMark, we cannot analyze the exact cause of this behavior. However, a likely cause is the differences in implementation. In our baseline implementation, ChaCha20 is implemented in the user space and pixels are written to the frame buffer using Linux's `/dev/fb*` interface. In `Rushmore`, ChaCha20 is implemented in the secure world's kernel space and pixels are written in the secure world's kernel space. The PassMark result demonstrates that with a single image, the performance of `Rushmore`'s kernel-level operations (even with frequent world switches) is better than the normal world's user-level performance. However, as we increase the number of images to display, the overhead of handling multiple images for `Rushmore` causes a steeper decrease in performance.

6.3 TCB Size

In the secure world, we implement `Rushmore` kernel using OP-TEE as described in Section 4.5. In Table 5, we present the additional lines of code required for our `Rushmore` kernel. `Rushmore` kernel consists of the request monitor, the default service, the kernel core, the IPU driver, and the cryptography module.

The majority of the TCB comes from the cryptography module (software AES, Hardware AES, ChaCha20, and VC). If we choose to use only one cryptographic method, we can further reduce the total TCB size from 2.5K LoC to 1.5K with ChaCha20 (which has 0.3K LoC) and 2.1K with VC (which has 0.9K LoC). We do not list TZASC driver in Table 5 since we mostly re-use OP-TEE's implementation (we modified 5 lines of code).

Although we cannot do apples-to-apples comparisons, our TCB size is comparable to previous systems that have used OP-TEE for their implementation. For example, VButton [34] reports additional 1.3K, TrustTokenF [69] reports additional 4.5K, and TruZ-View [68] reports additional 3.4K.

6.4 Security Analysis

In this section, we present three different types of security analysis to show how `Rushmore`'s secured display preserves confidentiality and integrity.

Confidentiality of Displayed Images: Our threat model in Section 2 assumes that adversaries are interested in compromising image confidentiality. Since our threat model also assumes that adversaries can fully compromise the normal world, they can get a hold of encrypted images that are to be displayed by `Rushmore`. Then the security guarantee for each of those encrypted images depends on the strength of the encryption scheme each uses. In our implementation, we use standard cryptographic methods that are proven to be secure, e.g., AES and ChaCha20, hence provide confidentiality for encrypted images.

In addition, `Rushmore` protects the confidentiality of images when operating in the secure world with TrustZone with TZASC. Since we give exclusive access permission to the secure world for one of the IPU cores (i.e., the registers that control the IPU core), the overlay buffer (i.e., the memory region), as well as other memory regions used by the secure world, the normal world cannot access anything that is contained within these regions including confidential image data. The normal world can only access its own memory regions and the buffer memory regions shared between the normal world and the secure world for passing image data and arguments.

Confidentiality of User Input: Previous UI protection schemes [2, 34, 36, 68] protect not only UI elements but also user input. However, `Rushmore` only focuses on securing images and does not provide user input security. Nevertheless, we can still protect user input indirectly as shown in the randomized keypad use case in Section 5.4. By randomizing a number arrangement on a virtual keypad displayed by the secure world, `Rushmore` preserves the confidentiality of user

input because adversaries cannot observe the actual numbers typed by a user.

Integrity of Displayed Images: Integrity is another crucial security aspect of the images displayed by Rushmore. There are two integrity-related attacks that we need to protect against—image overlaying and denial of service. Rushmore uses a separate overlay buffer to display images in the secure world, and pixels on the overlay buffer are always displayed over the pixels coming from the frame buffer in the normal world. Thus, a compromised OS can never overlay a different image above the secured image displayed by Rushmore. For denial of service, although our threat model considers it out of scope as described in Section 2, we can still provide a notification mechanism that turns on an LED light to indicate that the secure world is displaying confidential images. Although we do not implement it in Rushmore, previous systems [33, 68] do and show that it can be a functional solution.

7 DISCUSSIONS

Image Format: Rushmore uses RGB565 to reduce the size of an image. RGB565 uses 16 bits instead of 24 bits, which is the case for RGB888. Converting RGB888 to RGB565 is lossy and the result might show a greenish tint. However, one can get a better result by applying dithering [10].

Lack of Video Compression: Video compression is essential when streaming or sending a video with low latency. Rushmore currently does not use video compression for animated images and use raw image data in the RGB565 format. Adapting a video compression on Rushmore kernel might increase the TCB size in the secure world and incur display latency overhead because it requires decoding before decryption.

No Support for Animated Images using Visual Cryptography: We do not support animated images for visual cryptography because it is difficult to display two images (one from the normal world and the other from the secure world) exactly at the same time. This causes visual noise to occur at every frame transition. To reduce the noise, we could re-use the same key image for multiple frames instead of using a new key image for every frame. This would allow the transition noise to occur every n frames instead. However, it is possible that this would jeopardize the security of the animated images as it gives more chances to observe encrypted images with a single key image. We leave the full investigation of this as future work.

Long Term Impact Related to Processing Power Improvements: Rushmore currently supports up to 30 FPS to display whole-screen animations with ChaCha20. As processing power improves over time, Rushmore will be able to support higher frame rates and also 32-bit RGB instead of 16-bit RGB.

8 RELATED WORK

We review related work in three categories: (1) image protection, (2) visual cryptography, and (3) TrustZone research.

Image Protection: For general image protection, previous research was proposed for input image protection [1, 29, 48], physical attack protection [5], and UI protection using TrustZone [34, 36, 68]. Solutions for input image protection [1, 29, 48] first detect and recognize a confidential image from a camera frame, and then protect the image by occluding it from the camera frame. These solutions have a different focus from that of Rushmore, as Rushmore protects output images on the screen of a user device. HideScreen [5] proposes a grid-based display that limits the range of viewing angles. This prevents "shoulder surfing" where an attacker peeks at a user's screen to compromise what is displayed by the screen. In contrast, Rushmore's goal is to protect images against software running in the normal world of a device. UI protection solutions using TrustZone [34, 36, 68] also provide output image protection by providing a technique to securely display pixels. Using the main display channel (the frame buffer), they display the image of a UI element in the secure world and restrict access to the frame buffer for a short period of time until user actions are done with the UI. This means that the normal world cannot display anything on the screen while the protected UI is being used, which limits the applicability of using TrustZone for display security.

Visual Cryptography: Visual cryptography is an image-based cryptographic technique that splits an image into multiple images that, when overlaid with each other, reveal the original image. Previous research has focused on improving this technique for halftone images [64] and multi-color images [14, 66] as well as using this technique on different domains such as biometric data protection [49]. Our work uses visual cryptography as an alternative cryptography method to improve performance.

TrustZone Research: Previous research has solved various security problems using TrustZone, e.g., securely displaying text data [2], providing strong access control for sensors and peripherals on a device [33, 37], monitoring the normal world OS for a system [3], performing secure auditing [12], and analyzing a normal OS's execution [38]. These systems demonstrate the wide applicability of TrustZone for providing security in different domains.

9 CONCLUSIONS

In this paper, we have presented Rushmore that securely displays static or animated images using TrustZone. By leveraging an IPU and its overlay buffer, Rushmore allows the normal world to simultaneously display its content even when the secure world is displaying its confidential content. To the best of our knowledge, Rushmore is the first system that enables it. Furthermore, we adapt an image-based cryptographic method called visual cryptography, and show that Rushmore presents a novel application for it and it is light-weight. Our evaluation demonstrates that with the right type of cryptographic method, Rushmore can provide frame rates around or higher than 30 FPS for displaying encrypted animated images. This is also a capability previously not possible, and Rushmore's use of an IPU enables it. Lastly, we demonstrate the applicability of Rushmore's functionality by designing and implementing four use case applications.

ACKNOWLEDGMENTS

We would like to thank our reviewers and shepherd for their valuable feedback. This work was supported in part by the National Science Foundation, CNS-1618531 and CNS-1846320 (CAREER), and the National Research Foundation of Korea (NRF) grant NRF-2020R1A2C1004062.

REFERENCES

[1] Paarijaat Aditya, Rijurekha Sen, Peter Druschel, Seong Joon Oh, Rodrigo Benenson, Mario Fritz, Bernt Schiele, Bobby Bhattacharjee, and Tong Tong Wu. 2016. I-Pic: A Platform for Privacy-Compliant Image Capture. In *MobiSys '16*. Association for Computing Machinery, New York, NY, USA, 235–248.

[2] 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)*. Association for Computing Machinery, New York, NY, USA, 197–210. https://doi.org/10.1145/3081333.3081346

[3] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision Across Worlds: Real-Time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. Association for Computing Machinery, New York, NY, USA, 90–102. https://doi.org/10.1145/2660267.2660350

[4] Daniel Bernstein. 2008. ChaCha, a variant of Salsa20. *The University of Illinois at Chicago* (01 2008).

[5] Chun-Yu (Daniel) Chen, Bo-Yao Lin, Junding Wang, and Kang G. Shin. 2019. Keep Others from Peeking at Your Mobile Device Screen!. In *The 25th Annual International Conference on Mobile Computing and Networking (MobiCom '19)*. Association for Computing Machinery, New York, NY, USA, Article 22, 16 pages. https://doi.org/10.1145/3300061.3300119

[6] Haehyun Cho, Penghui Zhang, Donguk Kim, Jinbum Park, Choong-Hoon Lee, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. 2018. Prime+Count: Novel Cross-World Covert Channels on ARM TrustZone. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 441–452. https://doi.org/10.1145/3274694.3274704

[7] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern X86 Processors. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (SP '09)*. IEEE Computer Society, USA, 45–60. https://doi.org/10.1109/SP.2009.19

[8] Fabrizio De Santis, Andreas Schauer, and Georg Sigl. 2017. ChaCha20-Poly1305 Authenticated Encryption for High-Speed Embedded IoT Applications. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE '17)*. European Design and Automation Association, Leuven, BEL, 692–697.

[9] Advanced Micro Devices. 2020. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/.

[10] Dithering Cited Dec 2020. Android Bitmap.Config including RGB565 dithering. https://developer.android.com/reference/android/graphics/Bitmap.Config.

[11] Alexandra Dmitrienko, Zecir Hadzic, Hans Löhr, Ahmad-Reza Sadeghi, and Marcel Winandy. 2013. Securing the Access to Electronic Health Records on Mobile Phones. In *Biomedical Engineering Systems and Technologies*, Ana Fred, Joaquim Filipe, and Hugo Gamboa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 365–379.

[12] Nuno O. Duarte, Sileshi Demesie Yalew, Nuno Santos, and Miguel Correia. 2018. Leveraging ARM TrustZone and Verifiable Computing to Provide Auditable Mobile Functions. In *Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous '18)*. Association for Computing Machinery, New York, NY, USA, 302–311. https://doi.org/10.1145/3286978.3287015

[13] Anti-Abuse Research Lead Elie Bursztein. Cited Dec 2020. Speeding up and strengthening HTTPS connections for Chrome on Android. https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html.

[14] Young-Chang Hou. 2003. Visual cryptography for color images. *Pattern Recognition* 36 (07 2003), 1619–1629. https://doi.org/10.1016/S0031-3203(02)00258-3

[15] IPU Cited Dec 2020. Intel IPU3. https://www.kernel.org/doc/html/v5.4/media/v4l-drivers/ipu3.html.

[16] IPU Cited Dec 2020. Intel® PXA27x Processor Family Developer's Manual. https://courses.cs.washington.edu/courses/cse466/08wi/labs/l5/pxa27x_developers_manual.pdf.

[17] IPU Cited Dec 2020. JZ4760 Mobile Application Processor Programming Manual. https://dokumen.tips/documents/jz4760-mobile-application-processor-rockbox-jz4760-mobile-application-processor.html.

[18] IPU Cited Dec 2020. MCIMX51 Multimedia Applications Processor Reference Manual. https://www.nxp.com/files-static/dsp/doc/ref_manual/MCIMX51RM.pdf.

[19] IPU Cited Dec 2020. Mediatek chipset graphics driver. https://android.googlesource.com/kernel/mediatek/+/4f43e6b499c6d194030df8d2506485db9d5165bd/drivers/misc/mediatek/video/common/mtkfb.c.

[20] IPU Cited Dec 2020. MediaWall V Display Processor. https://www.syscomtec.com/_produktbereich/source/Artikel/Artikel%207693/350-11751-01_2018-10_mw-v_ug.pdf.

[21] IPU Cited Dec 2020. MT6797 LTE-A Smartphone Application Processor Functional Specification for Development Board. https://www.96boards.org/documentation/consumer/mediatekx20/additional-docs/docs/MT6797_Functional_Specification_V1_0.pdf.

[22] IPU Cited Dec 2020. Nios II Processor Reference Guide. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2cpu-nii5v1gen2.pdf.

[23] IPU Cited Dec 2020. OMAP35xx Applications Processor Introduction. https://plan9.io/sources/contrib/geoff/armdoc/ti/omap35x.intro.pdf.

[24] IPU Cited Dec 2020. Qualcomm's MSM chipset graphics driver. https://android.googlesource.com/kernel/msm/+/ad29d11ee316c7d363cb9cd4b4dffa02598d1711/drivers/video/msm/msm_fb.c.

[25] IPU Cited Dec 2020. Samsung's Exynos chipset graphics driver. https://android.googlesource.com/kernel/exynos/+/801e1de0316c0a62a4b07012de6f95562be1f926/drivers/gpu/drm/exynos/exynos_drm_crtc.c.

[26] IPU Cited Dec 2020. TDA2x SoC for Advanced Driver Assistance Systems (ADAS) Silicon Revision 2.0, 1.1 Technical Reference Manual. http://www.ti.com/lit/ug/sprui29f/sprui29f.pdf.

[27] IPU Cited Dec 2020. TI's OMAP chipset graphics driver. https://android.googlesource.com/kernel/omap/+/ecb19f44f9b0ba74cfaf303677beb7d079d4b62f/Documentation/arm/OMAP/DSS.

[28] Mohammad A. Islam, Shaolei Ren, and Adam Wierman. 2017. Exploiting a Thermal Side Channel for Power Attacks in Multi-Tenant Data Centers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1079–1094. https://doi.org/10.1145/3133956.3133994

[29] Suman Jana, David Molnar, Alexander Moshchuk, Alan Dunn, Benjamin Livshits, Helen J. Wang, and Eyal Ofek. 2013. Enabling Fine-Grained Permissions for Augmented Reality Applications with Recognizers. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 415–430. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/jana

[30] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. 2016. A High-Resolution Side-Channel Attack on Last-Level Cache. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. Association for Computing Machinery, New York, NY, USA, Article 72, 6 pages. https://doi.org/10.1145/2897937.2897962

[31] A. Langley. 2016. ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS). https://tools.ietf.org/html/rfc7905.

[32] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, 16.

[33] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. 2018. SeCloak: ARM Trustzone-Based Mobile Peripheral Control. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '18)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3210240.3210334

[34] 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 '18)*. Association for Computing Machinery, New York, NY, USA, 28–40. https://doi.org/10.1145/3210240.3210330

[35] 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 *Proceedings of 5th Asia-Pacific Workshop on Systems (APSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 8, 7 pages. https://doi.org/10.1145/2637166.2637225

[36] Dongtao Liu and Landon P. Cox. 2014. VeriUI: Attested Login for Mobile Devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications (HotMobile '14)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2565585.2565591

[37] Renju Liu and Mani Srivastava. 2018. *VirtSense: Virtualize Sensing through ARM TrustZone on Internet-of-Things*. Association for Computing Machinery, New York, NY, USA, 2–7. https://doi.org/10.1145/3268935.3268937

[38] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Krügel, and Giovanni Vigna. 2017. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *NDSS*. The Internet Society, San Diego, CA, USA.

[39] ML-Kit Cited Dec 2020. Face detection API in Google ML Kit. https://developers.google.com/ml-kit/vision/face-detection.

[40] Moni Naor and Adi Shamir. 1994. Visual cryptography. In *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer, Springer-Verlag, Berlin, Heidelberg, 1–12.

[41] Yoav Nir and A. Langley. 2018. ChaCha20 and Poly1305 for IETF Protocols. *RFC* 8439 (2018), 1–46.

[42] NXP. Cited Dec 2020. i.MX 6ULL Applications Processors for Industrial Products. https://www.nxp.com/docs/en/data-sheet/IMX6ULLIEC.pdf.

[43] NXP. Cited Dec 2020. i.MX21 Applications Processor Reference Manual. https://www.nxp.com/docs/en/reference-manual/MC9328MX21RM.pdf.

[44] NXP. Cited Dec 2020. i.MX25 Multimedia Applications Processor Reference Manual. https://www.nxp.com/docs/en/reference-manual/IMX25RM.pdf.

[45] NXP. Cited Dec 2020. I.MX355 IPU Manual. https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i-mx-applications-processors/i-mx-mature-processors/multimedia-applications-processors-based-on-arm11-core-image-processing-unit:i.MX355.

[46] NXP. Cited Dec 2020. i.MX53 Multimedia Applications Processor Reference Manual. https://www.nxp.com/docs/en/reference-manual/iMX53RM.pdf.

[47] PVC Cited Dec 2020. Pixel Visual Core: Google's Fully Programmable Image, Vision, and AI Processor For Mobile Devices. https://old.hotchips.org/hc30/1conf/1.02_Google_HC30.Google.JasonRedgrave.V01.pdf.

[48] Nisarg Raval, Animesh Srivastava, Ali Razeen, Kiron Lebeck, Ashwin Machanavajjhala, and Lanodn P. Cox. 2016. What You Mark is What Apps See. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '16)*. Association for Computing Machinery, New York, NY, USA, 249–261. https://doi.org/10.1145/2906388.2906405

[49] A. Ross and A. Othman. 2011. Visual Cryptography for Biometric Privacy. *IEEE Transactions on Information Forensics and Security* 6, 1 (2011), 70–81.

[50] Samsung Cited Dec 2020. What is the NPU in Galaxy and what does it do? https://www.samsung.com/global/galaxy/what-is/npu/.

[51] Freescale Semiconductor. Cited Dec 2020. i.MX28 Applications Processor Reference Manual. https://bootlin.com/~maxime/pub/datasheet/MCIMX28RM.pdf.

[52] Freescale Semiconductor. Cited Dec 2020. I.MX6 IPU Manual. https://people.freebsd.org/~gonzo/arm/iMX6-IPU.pdf.

[53] SGX Cited Dec 2020. Intel Software Guard Extensions (SGX). https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html.

[54] Tom Simonite. Cited Dec 2020. Apple's 'Neural Engine' Infuses the iPhone With AI Smarts. https://www.wired.com/story/apples-neural-engine-infuses-the-iphone-with-ai-smarts/.

[55] PassMark Software. Cited Dec 2020. PassMark. https://www.passmark.com/.

[56] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. 2015. TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. Association for Computing Machinery, New York, NY, USA, 976–988. https://doi.org/10.1145/2810103.2813692

[57] TEE Cited Dec 2020. OP-TEE. https://www.op-tee.org/.

[58] TEE Cited Dec 2020. Qualcomm Secure Execution Environment (QSEE). https://www.qualcomm.com/media/documents/files/guard-your-data-with-the-qualcomm-snapdragon-mobile-platform.pdf.

[59] TEE Cited Dec 2020. Samsung Knox. https://www.samsungknox.com/en/secured-by-knox.

[60] TEE Cited Dec 2020. Trusty. https://source.android.com/security/trusty.

[61] C. Tian, Y. Wang, P. Liu, Q. Zhou, C. Zhang, and Z. Xu. 2017. IM-Visor: A Pre-IME Guard to Prevent IME Apps from Stealing Sensitive Keystrokes Using TrustZone. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 145–156. https://doi.org/10.1109/DSN.2017.12

[62] TrustZone Cited Dec 2020. ANetwork Security Services (NSS) library developed by Mozilla.org projects. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/An_overview_of_NSS_Internals.

[63] TrustZone Cited Dec 2020. ARM TrustZone. https://developer.arm.com/ip-products/security-ip/trustzone.

[64] Z. Wang, G. R. Arce, and G. Di Crescenzo. 2009. Halftone Visual Cryptography Via Error Diffusion. *IEEE Transactions on Information Forensics and Security* 4, 3 (2009), 383–396.

[65] Xianyi Zheng, Lulu Yang, Jiangang Ma, Gang Shi, and Dan Meng. 2016. TrustPAY: Trusted mobile payment on security enhanced ARM TrustZone platforms. In *2016 IEEE Symposium on Computers and Communication (ISCC)*. 456–462. https://doi.org/10.1109/ISCC.2016.7543781

[66] Hirotsugu Yamamoto, Yoshio Hayasaki, and Nobuo Nishida. 2004. Secure information display with limited viewing zone by use of multi-color visual cryptography. *Opt. Express* 12, 7 (Apr 2004), 1258–1270. https://doi.org/10.1364/OPEX.12.001258

[67] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, USA, 719–732.

[68] 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 '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3292006.3300035

[69] Yingjun Zhang, Shijun Zhao, Yu Qin, Bo Yang, and Dengguo Feng. 2015. TrustTokenF: A Generic Security Framework for Mobile Two-Factor Authentication Using TrustZone. In *Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA - Volume 01 (TRUSTCOM '15)*. IEEE Computer Society, USA, 41–48.