From dusk till dawn: toward an effective *trusted UI*

Patrice Hameau, Philippe Thierry, Florent Valette
patrice.hameau@ledger.fr
philippe.thierry@ledger.fr
florent.valette@ledger.fr

Ledger

Abstract. Nowadays, secured embedded devices with high resolution displays, in which user interaction for critical assets is a part of the trust chain (user authentication, validation, etc.), leave the processing of the Trusted User Interface to the general purpose processor. Indeed, such displays are usually interfaced with MIPI-DSI and require an amount of reactivity and power processing a Secure Element is unable to provide. The fallback model is, for example in mobile markets, generally based on the ARM® TrustZone [22] and Trusted Execution Environment mechanisms, which imply that the Trusted User Interface relies on the very same cores of the general purpose processor as the applicative Operating System. The problem with such an architecture is that the ARM TrustZone security model has been initially designed in 2004 [22] and is not always adapted to the increased complexity of today's systems in the light of last years' new attack paths [15, 17, 18, 23, 29, 31]. In this article, we explain how we modified the management of the display by the various components of a representative mobile system including a separate Secure Element, in order to ensure the protection of the critical assets managed by the Trusted User Interface even when all the general-purpose components, including the ARM TrustZone environment, have been compromised by an attacker.

1 Introduction

The graphic subsystem, a part of the trust chain

As time goes by, user interaction is increasingly becoming an integral part of the trusted data path, making it essential to have a secured user interface in order to involve the end user in the execution of critical actions or when manipulating sensitive assets protected by a Secure Element (SE).

This user interface may require simple interactions, such as the FIDO *User Presence* [3] mechanism, or more complex ones such as Personal Identifier Number (PIN) or passphrase entries.

In both cases, the user interface used needs to be trusted enough as it is part of the global security mechanism [10], and thus is designated as a Trusted User Interface (*Trusted UI*).

When executed using the very same hardware as the untrusted system (same inputs and outputs), the Trusted UI accountability raises various questions [30].

Upon manipulation of sensitive assets, such as passphrases entries, the security requirements for the overall user interface are heightened in accordance with the level of sensitivity of the data being manipulated.

State of the art and limitations of Trusted UI

The Trusted UI problematic, and more generally the ability to execute different security levels that require user interactions is a complex problem. It has been described in multiple papers and theses [28, 30] up to a fully formal specification of the required properties a Trusted UI must support [30].
Nowadays, in the mobile market, Trusted UI is being more and more deployed in order to bring better security to user interactions when asset confidentiality, integrity or authenticity is required. This is typically the case when entering PIN entries (e.g. banking applications) or passphrases, or for validating critical actions (e.g. money transfer).

As the Trusted UI uses the device's main screen, it can't be directly managed by the SE, which has not the necessary power processing and memory resources. It is thus often delegated to the Trusted Execution Environment (TEE), based on ARM TrustZone technology. The graphic chain support model chosen by the industry is usually based on dynamic hardware *switching* of the devices required to manipulate the Trusted UI from non-TrustZone world (*normal world*) to TrustZone one (*secure world*).

Moreover, when any user-related security assets, such as PIN, is required by the SE to authenticate the user or to unlock some SE related security feature, it is performed using the Trusted UI. As a consequence, the asset confidentiality directly depends on the Trusted UI integrity, accountability and authenticity, potentially endangering the SE hosted services and assets.

The Android Open-Source Project has defined a methodology [8] based on the Keymaster [20] implementation to respond to such needs. It is still the sole protection element used in some TEE Operating System

vendors [13], despite the nowadays attack paths that have endangered this design [15,17,18,23,31].

On the other side, Apple for example uses a dedicated proprietary hardware architecture based on its own external Secure Enclave chip [6,32].

Based on the security state of the art defined above, the usual ARM TrustZone based Trusted UI model suffers limitations impacting the overall level of confidence and security that can be considered for such implementations:

— The Trusted UI security domain accountability [30]
— The dynamic sharing of the graphic chain (input, output) on which potential fault injection methods can be considered, including remote ones [7,15,23,31]
— The effective runtime protection (data confidentiality, integrity) of the TEE software [17,29]

Despite the implementation of some security best practices, such as handling power management at the Secure Monitor level in ARM architecture, this model has limitations rooted in an outdated initial design that fails to adhere to the principle of least privilege in practice: the lower secure component, in general the applicative Operating-System running in non-secure world (saying Android), has a huge amount of privileges on the hardware platform (user I/O, frequency scaling, SoC I/O mixer, etc.), impacting the higher security level Operating System (saying the TEE). This design is the source of a lot of successful attacks [15, 17, 18, 23, 31], and alternative or complementary solutions should be considered.

To address this issue, we propose a new model that adheres to the principle of least privilege, removing direct access to any potentially critical hardware component from the lower secure component (e.g. Android Operating system).
In our opinion, the para-virtualization model, as demonstrated by the Genode team in 2014 [16], which uses a virtual touchscreen and virtual frame buffers within the Android world, is a better solution that is highly easier to verify and prove in terms of security and less prone to failure.
Moreover, in such an architecture, all the hardware devices impacting the Trusted UI are dedicated to a security domain by design, with no dynamic configuration at all. This enables the use of hardware lock registers, which prohibits modification of the security domains configuration (access rights,

peripherals owned, ...) after first setup until the next full reset of the SoC.

In order to establish a reasonable security context for our proof of concept (PoC), we have formulated a simplified generic threats model that can be applied to various Android-based devices:

— **Threat 1.1** *The Android world is connected to the Internet*
— **Threat 1.2** *The Android world, including the Linux kernel, is considered as too complex to be properly secured, and as a consequence is considered as fully controlled by the attacker*
— **Threat 1.3** *Software-based hardware attacks, such as [7, 15, 23, 31] are considered*
— **Threat 1.4** *The device can be stolen and thus physical attacks can be conducted to get access to critical assets, in scenarios that make sense*
— **Threat 1.5** *Non-intrusive hardware faults (typically EM-based) are considered, in scenarios that make sense*

Considering the above threats, some hypothesis are defined for our security model:

— **Hypothesis 1.1** *The main SoC*[1] *documentation is correct*
— **Hypothesis 1.2** *The main SoC IP*[2] *behave as specified and do not hold any hidden features*
— **Hypothesis 1.3** *The main SoC brings an in-chip separated companion core (named Protected Core) and some On-Chip RAM (OCRAM) with exclusive access to this core*
— **Hypothesis 1.4** *the main SoC supports protection against any bus master for its internal memories or peripherals (integration of ARM SMMU*[3] *or equivalent)*
— **Hypothesis 1.5** *The main SoC secure boot mechanism is trustworthy, protected against non-invasive attacks*
— **Hypothesis 1.6** *The SE and the main SoC embedded software used till the Protected Core startup are considered trustworthy.*

With the above security threats and hardware assumptions in mind, the Trusted UI proof of concept we have designed aims to adhere to the following security considerations:

— **Security property 1.1** *The Trusted UI supports user accountability for managing critical assets*

[1] System On Chip
[2] Intellectual Properties, devices embedded in-chip
[3] System Memory Mangement Unit

— **Security property 1.2** *The Trusted UI component data at rest protection is guaranteed*
— **Security property 1.3** *The Trusted UI software runtime integrity must be guaranteed (using for example OCRAM*[4] *with ECC,*[5] *monitoring, ...)*
— **Security property 1.4** *The Trusted UI must resist to usual TEE exploitation paths*
— **Security property 1.5** *The Trusted UI must not be impacted by any attack or corruption of the software running in applicative security domain (e.g. Android, Linux, TEE)*
— **Security property 1.6** *The Trusted UI must not depend on hardware device(s) dynamic sharing. It shall rely on assignments of access rights and peripherals to a dedicated security domain at boot time, and the possibility to lock these assignments in hardware till next SoC reset.*

In the light of the above threats model, hardware properties and security properties that we aim to achieve, we present a novel Trusted UI architecture in Section 2. We then discuss the designed Trusted UI performances, security impacts, advantages and drawbacks in Section 3. Finally, we describe what could be made to optimize the results and optimize some limitations in Section 4.

Proof of Concept

To validate our work, we have developed a fully functional prototype internally (referenced as *PoC*). It is based on a recent version of Android AOSP running on an iMX8 SoC from NXP. It includes also a Secure Element from STM and a high resolution display of 720p driven by a MIPI DSI interface. The chosen iMX8 SoC integrates 4 Cortex-A53 cores and a Cortex-M companion core, used as the Protected Core. The SoC supports a Secure Boot mechanism, and integrates the hardware capabilities required to enforce security domains isolation (SMMU, RDC, OCRAM, ...) and locking mechanisms.

2 A new UX paradigm: Moving to three third architecture

In this section, we begin by outlining the general abstraction model we used for the Trusted UI in Section 2.1. We then expose how the security

[4] On-chip RAM
[5] Error Correction Code

architecture is deployed and locked in Section 2.2. Some time is taken in Section 2.3 to explain how the graphical output partitioning has been made, as this is the more complex part. We conclude by detailing how we completed the overall integration by including the input part in Section 2.4, and the SE [6] connection in 2.5. As a final word, we finalize the description of the overall hardening with additional elements in Section 2.6.

2.1 Virtualizing ARM64 graphic subsystem

Modern ARM64 SoCs sometimes hold a companion core, mostly in industrial fields such as automotive or medical [9]. In most of those cases, the companion core is designed for safety critical applications and is associated with a set of hardware components in order to protect it against any potential corruption from the main ARM64 core cluster.
To fully meet Properties 1.4 and 1.5 and minimize the impact of new existing threats in the design of a mobile-oriented user interface, we have decided to extract the overall graphical chain handling from the ARM64 core complex by utilizing the in-chip companion core to host those functionalities. Using a fully separated core is also considered instead of, for example hardware virtualization because:

— hardware virtualization, including ARM64 one, is not immune at all against various threats defined in Section 1, as explained in [12].
— hardware virtualization is fully-arch specific, making our architecture non-portable to, for example RISC-V based architectures

The companion core, in our usage of the i.MX8 SoC, exclusively uses in-chip dedicated memory banks and cache hierarchy that are separated from the ARM64 core complex for its own usage.
This decision comes at the cost of hosting such a core in a design commonly used in the industrial field. Based on these constraints, the goal of the Proof of concept is to evaluate the impact on security, performance and consumption of the overall solution.

Having decided to utilize the companion core to host the Trusted UI, and given that this core is inherently immune to various threats associated with the ARM64 core complex, we refer to the overall core (companion core and associated resident firmware) as the *Protected Core* in the following. The usage of a such a core for the Trusted UI gives us various advantages:

— its execution starts before any of the ARM64 core cluster OSes (TEE and Android typically)

[6] Secure Element

— The companion software execution is fully independent from the normal world, on a separated core, power domain and clock domain
— the companion software execution is not impacted by any of the ARM64 core cluster cache-based side-channel
— the Protected Core software is significantly smaller than TEE implementations. This allows us to incorporate noRTE code validation, fault-resilient implementation, control flow integrity considerations, etc.
— the Protected Core Software can be fully hosted in OCRAM (code+data), excluding the DDR for its own usage
— the Protected Core has a limited scope of operation and does not support additional applications unlike TEE OSes. As a result, it will not host any potentially buggy trustlet
— the Protected Core uses a highly simple, hardware-based, communication interface with the ARM64 core cluster
— as the Protected Core has a different architecture and mapping, there is no data pointer used, but only identifiers, avoiding attacks such as [27]

At the same time, to comply with Property 1.6, we have chosen to adopt a similar approach as the Genode team [16], by utilizing para-virtualization methodology. The main difference with the Genode proof of concept being the use of a fully separated hardware core instead of an ARM TrustZone based virtual machine monitor.

In our proof of concept, we move the overall control of the graphic chain, including the LCDIF [7] controller, the MIPI [8] bridge controller and the panel interactions to the Protected Core. Additionally, we also shift the responsibility of the touchscreen controller's low speed bus (in our case, an I2C [9] controller) to the Protected Core.
Based on this design, the Protected core then behaves as a graphical proxy between the main ARM64 core cluster and the actual physical graphic chain.

Moreover, in our proof of concept, the Protected Core also behaves as a proxy between the SE and the main SoC ARM64 core cluster. Based on this architecture, the Protected Core is able to respond in an autonomous way to highly secure-critical SE requests, and to display Trusted UI elements or

[7] LCD Interface
[8] Mobile Industry Processor Interface
[9] Inter-Integrated Circuit

to read user inputs without requiring any ARM64 core cluster execution. This is done by taking full control of the input/output interfaces that the Protected Core already manages, allowing high security user interface management (PIN requests, etc.) in direct interaction with the SE.
By doing that, we comply with our Security Model Property 1.5. As there is no hardware device cross-domain switching required in order to enable the Trusted UI interface, we also comply with our Security Property 1.6.

In order to ensure the accountability of the Trusted UI, the user must have a secure way to confirm that the effective displayed UI is managed by the Trusted UI. This is required in order to comply with Property 1.1. The key advantage of our design is the proxy model, in which the Protected Core is alway the sole component controlling the screen. Based on this principle, the Protected core can dedicate a part of the output framebuffer in order to keep a *secure bar* with a dedicated informational value, typically in the way described in [30].

Given that the graphical subsystem is fully hosted in the Protected Core, it is imperative that this core is fully protected against any direct or indirect exploitation from the main ARM64 core cluster. This includes the Protected Core itself, and all the devices used for the Trusted UI data and control plane, and the SE communication. We describe how we harden the Protected Core in Section 2.2.

Once the graphical chain is managed by the Protected Core, a fast hardware synchronization mechanism between the ARM64 core complex and the Protected Core is required in order to allow enough reactivity for graphical events, as the ARM64 software does not manage anymore any of the effective hardware involved in the graphical chain. This requires a hardware component able to communicate easily and fast between different clock domains, to abstract any synchronization complexity between the two core complexes. In our proof of concept, the i.MX8 SoC does have such an IP for this usage, denoted *MessagingUnit*. This unit is a basic four-register-based mailbox device that generates interrupt events between peers when the registers are set.

In order to efficiently communicate between core complexes with this IP, a fully synchronous simple protocol has been designed, based on the basic structure defined in Table 1.

There are multiple message-types for different requests and responses that determine the potential type and value of arguments.

register	size	type
r0	u32	message-type
r1	u32	auth-token
r2	u32	arg
r3	u32	crc32(r0,r1,r2)

Table 1. Generic communication structure between core complexes

The *MessagingUnit*-based communication protocol is, at the time of this article, a fully synchronous protocol, which is reactive enough for our performance needs, as described in 3. This also makes the protocol stack implementation highly easier to implement and to prove in terms of noRTE and functional correctness.

The overall protocol specification is not described in this article for the sake of clarity. Nevertheless, all the protocol frames respect the above specification.

2.2 Hardening the Protected Core

Authenticate peers through the MessagingUnit In the security model we have defined, the only way of communication with the Protected Core is through the *MessagingUnit* hardware interface.

The SoC datasheet specifies that:

- the *MessagingUnit* handles two registers sets per half-duplex communication pipe:
  - four write-only registers in A (resp. B) domain, that triggers B (resp. A) domain after writing the 4^{th} register
  - four read-only registers in B (resp. A) domain, that correspond to the content of A (resp. B) domain registers at interrupt time
- A and B domains are hardware-associated respectively with the ARM64 core complex and the Protected Core core complex
- write registers are write-only. Reading from them returns only `0x0`
- a given domain can't access remote domain registers

The communication is made using the basic synchronous protocol introduced in Section 2.1. In our design, two couples have to communicate through this interface:

- Android kernel <=> Protected Core
- TEE kernel <=> Protected Core

The goal here is to enable the TrustZone-based User Interface, even if, in our security model, such an interface is not considered secure enough for our needs. In our model, the effective Trusted UI security is associated to a fully independent execution of the eSE / Protected Core couple, independently of the ARM64 core complex, making the MU peer authentication corruption out of the scope.

Nonetheless, to enable such a support, the MessagingUnit is set at a given time t to a given security level (TrustZone security flag), by configuring the SoC dedicated IP: the CSU.[10] The switch, in our model, is under the responsibility of the Protected Core, and peers are informed when they are allowed to speak through the Messaging unit. This allows to naturally authenticate the Protected Core remote peer, with the help of the TrustZone hardware mechanisms.

In the meantime, we consider in our threat model that:

1. the CSU configuration of the *MessagingUnit* device may fail (bad error catching)
2. the CSU configuration of the *MessagingUnit* device may be attacked (fault injection)
3. the Protected Core context may be corrupted (run time error, single fault injection)

To increase the level of authentication of the Protected Core remote peer, an additional mechanism has been added to the communication protocol, by adding an authentication token with anti-replay protection. This is achieved by sharing an internal state seed initializer at boot time with each peer, for each half-duplex communication, which must be used as soon as the init state is terminated.
In our proof of concept, the RNG source used for this seed in the protected core is the SoC CAAM (Cryptographic Acceleration and Assurance Module) TRNG module. The random value is loaded at early Protected Core boot time, while no other software than the Secondary platform Loader is started.

In order to share this seed with each peer and associate it with the corresponding security context, we use the ARM secure boot sequence which guarantees the boot order. The platform boot order is, in our case, the following:

[10] Central Security Unit

1. bootROM startup, secure boot bootstrap
2. Secure Platform bootloader (e.g SPL), which includes Protected Core loading
3. Protected Core startup
4. Secure Monitor (e.g. ARM TF-A) startup
5. Trusted Execution Environment (TEE) OS startup
6. Normal world bootloader (e.g. U-BOOT) startup (after Secure Monitor switch from TEE)
7. Normal world OS (e.g. Linux)
8. Android boot

This allows the Protected Core to first negotiate a seed vector with the TEE first, and then with the Android kernel.

To authenticate the emitter once the initialization sequence is done, this seed is used in order to generate a predictable sequence of randomly generated numbers using the PCG32 [21] algorithm each time a frame is emitted between peers. Each generated number is written in the *auth-token* field of the communication structure described in Section 2.1, and checked by the receiver, which locally generates the very same random sequence as its remote counterpart.

The big advantage of the PCG32 algorithm is its cost: executing a PCG32 increment can be done in a few nanoseconds, in comparison with a complete cryptographic sequence such as a full HMAC algorithm.
In our performance model, such an advantage is critical to be able to keep reasonable graphical performances, as described in Section 3.

Listing and protecting all critical hardware components

Once the communication between the Protected Core and the ARM64 core complex is secured, a lot of work is still needed.
First, a lot of hardware components are required in order to properly manipulate the user interface. This requires input and output interfaces, including both high-speed (typically MIPI) and low speed (usually I2C or SPI) communication buses. In order to use these buses, the SoC I/O muxer and the GPIO(s) controller(s) to which these devices are connected also need to be involved.
Communication buses also require input clock configuration (input PLL

setting).
The communication bus (simple serial communication bus, such as SPI) with the eSE also needs to be protected, including, again, its associated GPIO controller.
Figure 1 describes all the hardware components required in order to enable a functional user interface.

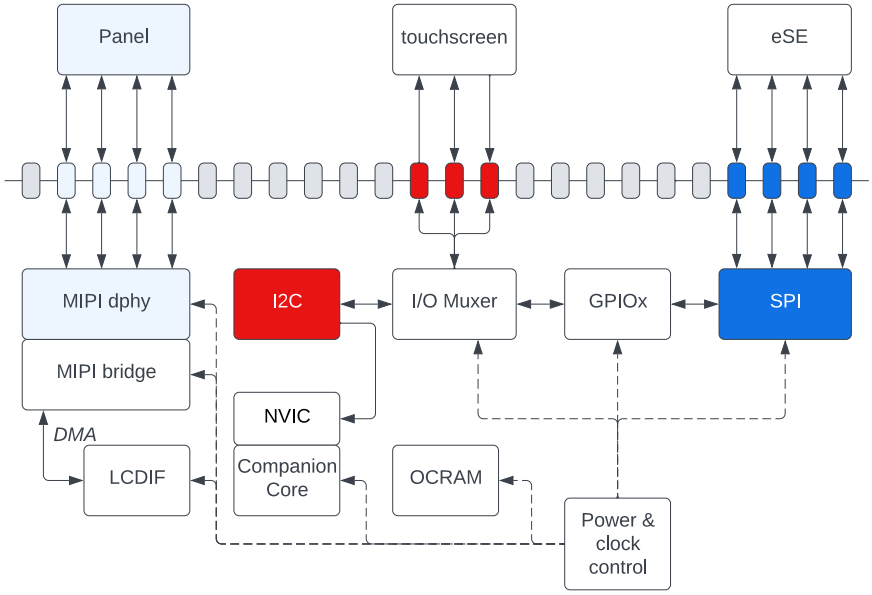

Fig. 1. Hardware components impacting the Trusted UI

In a dynamic model, in which the Trusted User Interface needs to be switched from one security level to another, the ownership of all these components needs to be transferred, and their registers need to be verified by the upper security level to validate the proper Trusted UI data plane configuration.
If not, most of them can be used in order to redirect a part of the flow, inject custom content (typically using the I/O muxer), spy the input events reception (by polling the touchscreen corresponding GPIO pins), and so on.

In our security architecture, all devices but the Power controller have been locked under the Protected Core responsibility. Therefore ownership

is clear and the configuration of registers can be locked at the earliest boot time with a quite small TCB at that time. The Power controller is a special case because efficient power agility on various devices must be kept without requiring too much effort from the Protected Core itself.
To achieve that, the Power controller control is shared between the Protected Core and the secure monitor exclusively, with a static filtering mechanism for devices that are under the Protected Core domain in the secure monitor power driver implementation. By doing that, the secure monitor never manipulates the Protected Core relative devices, Android and TEE have no access but secure monitor calls to request any power control update.[11]
Meanwhile, the Protected Core validates the overall IPs power and clock configuration each time the Trusted UI is used, detecting potential corruption.

To ensure an efficient separation of the Protected Core devices domain and the ARM64 core complex domain, we use the i.MX8 *Resource Domain Controller*. This controller is able to strictly separate all bus masters, including the core complexes themselves, in differentiated worlds, disallowing uncontrolled inter-domain communication.
The Resource Domain Controller configuration is under the full control of the Protected Core itself, and all dedicated devices are locked during the Protected Core startup at the earliest boot time.

Secure booting the Trusted UI

In order to comply with Security Property 1.2, the overall Protected Core software image must be a part of the platform secure boot process. The secure boot process is initiated by the BootROM based on in-chip eFuses holding a public key, associated with a cryptographic component used in order to successively validate the initial images. For the sake of clarity, the secure boot process is considered out of scope and is not explained in this article but is based on the standard NXP High Assurance Boot [4]. In the HAB process, the Protected Core image has been added to the initial boot image checked by the Secure Platform Loader, based on modified SPL/U-BOOT software, adding data at rest integrity and authenticity emanated from the secure boot public-key based signature check.

[11] A potential full move of the power control to the Protected Core is also analyzed to fully protect this part

Now that the platform hardening has been made and all Trusted UI security properties fulfilled, the actual implementation needs to be explained.

2.3 Plugging in all together

About MessagingUnit UI-related frames

The display pipeline is moved under full control of the Protected Core, the virtual CRTC API is exposed through *MessagingUnit* as seen in Section 2.1. Based on the protocol description in Section 1, upper layer software can emit messages (with content), emit signals (without content), receive messages (with content) and signals (without content).

All messages emitted through the MU are associated to a status response, which is a dedicated message handling a 'response' bit. This response is emitted by the peer and returns the result of the peer message handling.

Virtual CRTC API The virtual CRTC API is composed of a few endpoints in order to enable a peer to enable/disable the display pipeline and swap the scanned out framebuffer. The framebuffer may be directly rendered or composed from a secure and a non secure frame (see Section 2.1) :

- vblank event [12]
  - Message type: *Request* / Emitter: Protected Core peer
  - Description: On hardware vblank interrupt, a message is sent to peers. Thus, peers can commit the next framebuffer, if any, and start rendering any further frame.
- disable display
  - Message type: *Request* / Emitter: Cortex-A53 core peer
  - Description: On "disable display" reception, the protected core will turn off the panel and stop transferring framebuffers to LCD controller
- enable display
  - Message type: *Request* / Emitter: Cortex-A53 core peer
  - Description: On "enable display" reception, the protected core will configure the graphical pipeline, i.e. LCD controller and MIP DSI bridge, according to the current modeline. Once done, LCD controller to MIPI DMA transfer is armed and the panel is turned on.

[12] Vertical blanking is the period from the end of a framebuffer scan out and the beginning of the next frame

- update framebuffer ID
  - Message type: *Request* / Emitter: Cortex-A53 core peer
  - Description: Program the next framebuffer to display. This command will not apply any changes in order to prevent tearing. The new framebuffer commit must be done during vertical blanking. This can be handled by hardware. In our proof-of-concept, the LCD controller has a shadowed register in order to get tearing free page flip. Shadowed register will be committed at next vsync event [13] once refresh is programmed.
- refresh framebuffer
  - Message type: *Request* / Emitter: Cortex-A53 core peer
  - Description: Tells hardware to start using the previously defined framebuffer. Depending on hardware capabilities, this commands tells hardware to commit its shadowed register or emulate this behavior in software by using vblank interrupt.

Virtual Input API

- Input touch event received: position number
  - Message type: *Request* / Emitter: Protected Core peer
  - Description: On hardware input touch interrupt, the number of active position (finger(s) on screen)
- Input touch event received: position
  - Message type: *Request* / Emitter: Protected Core peer
  - Description: On hardware input touch interrupt, the touch data: positions (x/y) and state (pushed, released, kept, moved)

Integrating to the Trusted Execution Environment

In this article, we consider that today's TEE implementations are based on a basic framebuffer interface [11] that can be easily plugged over the *MessagingUnit* mechanism. On the contrary, in Android, the display ecosystem is highly more complex and requires bigger performances.
As a consequence, we decide to focus this paper on the way we virtualize the Android operating graphical chain, and leave the Trusted Execution Environment apart.

Integrating to Linux DRM subsystem

[13] Vsync is the beginning of a frame scan out

The Linux DRM [14] layer is the infrastructure that helps complex GPU [15] driver writing. Each driver can handle a wide range of features such as 3D rendering, KMS,[16] GEM [17] object allocation, etc.

Here, we are building a DRM driver for display pipeline (Figure 2) only, so we need to implement three DRM features, *ATOMIC*, *MODESET* and *GEM* [2,14]. The DRM device controlling the virtual display pipeline owns a single DRM CRTC [18] device with only a primary plane (i.e. no cursor plane nor overlay). Encoder and Connector are DRM devices of virtual type as they are not handled anymore by the DRM subsystem.

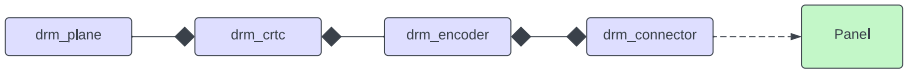

Fig. 2. Simple DRM display pipeline

DRM CRTC The CRTC controller drives the display setting and timing and is responsible for scanning the framebuffer [2]. The driver handles the following events from the display controller:

1. VBlank interrupt
2. "DRM refresh" command
3. "Display enable" command
4. "Display disable" command.

DRM PLANE A DRM plane holds a state that defines the current buffer of this plane and the bound CRTC. Due to hardware limitation, we only support one primary plane (no cursor nor overlay) and composition is done in user space [1]. On DRM MODE COMMIT action (Figure 3), the new plane state, with the next framebuffer to render is committed atomically. The corresponding plane update helper [2] sends the next framebuffer to display to the Protected Core.

[14] Direct Rendering Manager
[15] Graphics Processing Unit
[16] Kernel Mode Setting: display configuration
[17] Graphics Execution Manager
[18] Cathode Ray Tube Controller

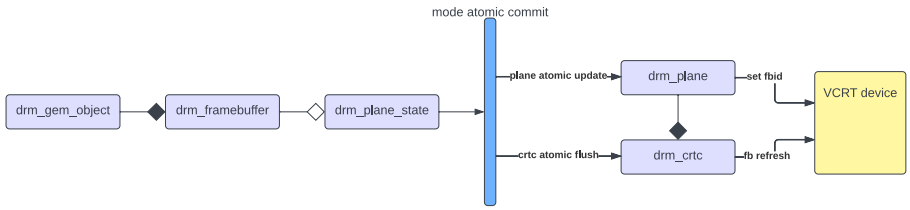

Fig. 3. DRM mode atomic commit

GEM CMA The GEM contiguous memory allocator can only allocate memory for framebuffer objects at fixed place due to security and hardening considerations discussed in Section 2.2. All rendering steps are done in the 2D/3D GPU by Android graphical stack [1] and the final layers composition targets the previously allocated framebuffer. Thus, our driver needs to support PRIME [19] buffer export, but, those design constraints imply that PRIME buffer import can't be supported.

In order to force memory location of each framebuffer, the driver needs to handle a reserved memory region [2] with shared-dma-pool and no-map attributes for each framebuffer. Each DMA pool is used to allocate a unique framebuffer, thus the base address of each buffer is known and predictable as shown in Figure 4.

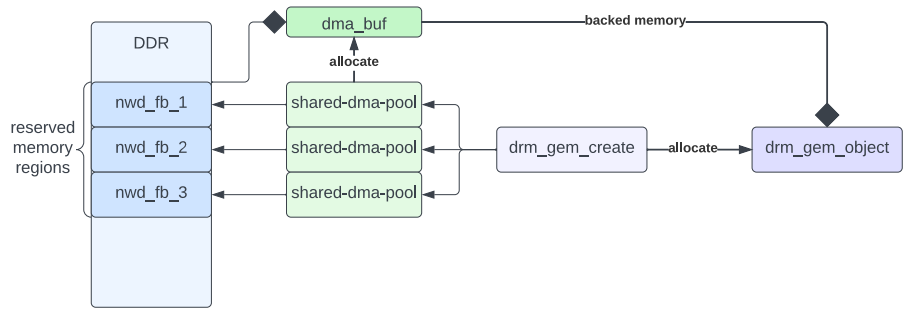

Fig. 4. DRM GEM CMA allocator

Based on the above implementation, the linux DRM-compatible graphic chain has been connected to the Protected Core through the *MessagingU-*

[19] PRIME is the cross device buffer sharing framework in DRM

nit, in association with three predefined framebuffers, in order to let the GPU3D controller manipulate the Android-level rendering and schedule. These framebuffers addresses are known at build time and shared with the Protected Core, as shown in Figure 5.

2.4 Integrating to Linux Input subsystem

The Linux Input layer is the infrastructure that unifies all input devices in order to abstract the various input hardware to an unified input types for devices such as touchscreens, mouses, and so on. On the opposite of the DRM subsystem, the Linux input subsystem is a quite easy system in which the lower driver only registers itself and triggers the upper input API.

The framing model described in the beginning of this section is sufficient in our Proof of Concept to manipulate a touchscreen through a virtualized architecture using the *MessagingUnit*. In this model, the device-specific implementation is kept in the Protected Core. This includes the interrupt handling and the associated I2C requests in order to get back the associated event information.
The positioning and state information (coordinates vector, number of fingers, touch type, etc.) is then emitted through the MessagingUnit toward the current User interface target driver, which handles the positioning information in its own context (input subsystem in Linux, etc.).
For the sake of simplicity, we have limited the input support to basic multitouch (no palm support).

2.5 Activating TrustedUI: from proxying to local looping

Now that we have a real input/output proxy held in the Protected Core between the ARM64 core complex and all the hardware IPs used in the input/output subsystem, a last brick is added to the Protected Core: a proxy integration between the SE and the TEE.

When the SE software needs to manipulate sensitive assets or perform critical actions (e.g. unlock some feature using a PIN or execute some cryptographic service), it then sends a request to activate the Trusted UI. Through its proxy position in the communication between the SE and the ARM64 core complex, the Protected Core will intercept such a request and setup the Trusted UI to perform the required secured user interaction. The ARM64 core complex may even not be informed of such requests.

As the SE hardware interface with the main SoC is a low-speed communication bus (e.g. SPI, ISO7816, etc.), such a proxy can be easily implemented in the Protected Core in a very small amount of code.

The overall architecture including both Linux kernel and Trusted Execution OS is described in Figure 5.

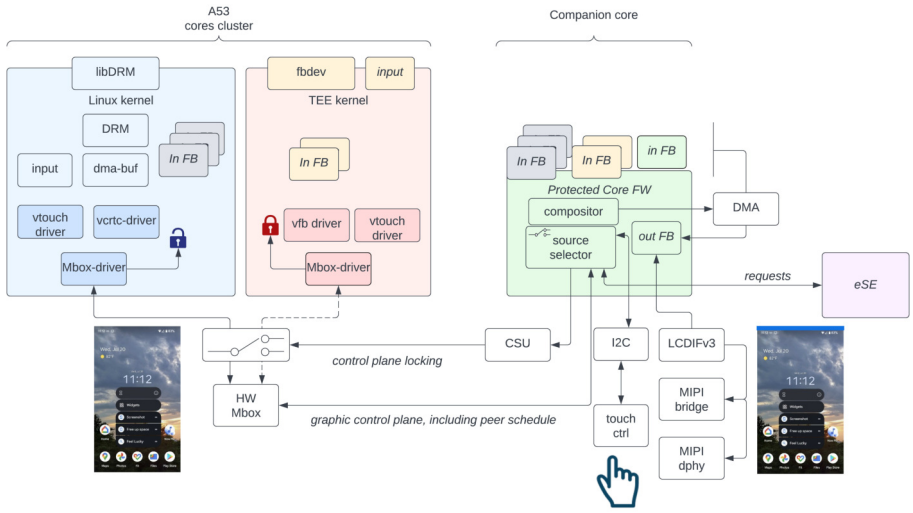

Fig. 5. General graphic proxy architecture

Managing some Trusted UI requests in the Protected Core necessitates a basic implementation of some user interface elements, such as basic graphical drawing (circle, rectangle) and character set encoding. These elements being kept simple, they can be implemented using a small amount of code.

In the end, adding a proxy with the SE allows to enforce the robustness of the Trusted UI activation, with a minimal impact on the Protected Core TCB size and complexity.

2.6 Icing on the cake: formal proof

As the Protected Core interface with the ARM64 core complex is quite simple and based on a fully specified communication protocol written using a state automaton, it has been possible to verify it using the Frama-C framework (in the same fashion as the Wookey project [5, 26]).

The communication interface has been checked against run-time errors and the state automaton correctness validated.
Supplementary protections against faults based on usual resilient coding patterns are also being used in a way that was previously analyzed in term of security [19].

3 Results and discussion

3.1 Security gains

Based on the global security architecture we have deployed, we can consider that:

1. No additional cold-boot attack path is leveraged by this architecture, in comparison with an ARM TrustZone-based security service. The Protected Core firmware is fully deployed at the earliest boot time, before the TrustZone components such as the Secure Monitor and the Trusted Execution Environment Operating System, with a very minimal TCB (SPL code only).
2. The usage of the para-virtualization paradigm instead of dynamic hardware resources control switch is a strong security enhancement, as it allows resources to be fully locked to one the defined device's security domains at boot time (the lock using the lock register bits of the hardware security controllers and remaining active till next SoC hardware reset). Using such a model, a strict delimitation between the initialization (configuration definition) and the runtime (locked configuration) phases reduces the critical temporal window to the startup sequence, during which neither the Android operating system nor the TEE is started, and at a moment where the device is not yet connected to any external parts (the SoC startup sequence being fully in-chip, using exclusively the OCRAM and the internal peripherals). Through para-virtualization methodology, the para-virtualized worlds are easier to support when in background, as there is no hardware device (un)locking that may fail the driver or the power management support, leading to potential instabilities and attack paths. Instead, the para-virtualization layer simply drops messages when the peer is in background, keeping an unified implementation without any potential complex attribution mechanism.
3. During the runtime sequence, the Protected Core is not exposed to any of the usual main cores cluster side channels based on shared

cache, shared memory and shared processor resources. However, enforcing the required isolation of the Protected Core cannot be done in a trivial and automated way: it requires a full analysis of any potential direct (mapping) or indirect (through bus masters) accesses between the defined security domains. On one hand, this allows a clear and easy way to analyze separation between the security domains and on the other hand requires from the hosting SoC the capacity to properly separate such security domains for all the devices embedded in the SoC design.

4. The graphic management proxy methodology used is an efficient way to strictly demonstrate the accountability of a given screen, when keeping a part of the output buffer under exclusive control of the Protected Core itself. Keeping such a part always displayed with an explicit informational value allows specifying which level of security is shown on the other parts of the screen (for example by using a color bar always present on one side of the display), and allows to fulfill formal accountability requirements needed in Trusted UI described in [30]. Other mechanisms more complex than a color bar can also be imagined, including the use of dedicated external LEDs aside of the display under exclusive control of the Protected Core or of the SE.
5. the proxy model is a full enabler to a fully independent eSE-controlled Trusted UI, requiring no action from neither the Android nor the TEE world.

3.2 Performances

In our PoC the overhead of graphical pipeline virtualization on performance is very limited. There is no impact on rendering in user space as the framebuffer object can be exported and thus shared between our DRM driver and GPU vendor one. Android plane composition is done by GPU 2D hardware accelerator directly in the normal world framebuffer managed by the Protected Core. Compared to direct hardware handling in Linux kernel, VBlank event notification costs an extra message exchange sequence using *MessagingUnit*, and Mode Atomic Commit two extras exchanges.

The *MessagingUnit* latency was measured by software using Linux kernel *ktime* infrastructure on continuous ping exchange with the Protected Core. We were printing the time interval every 10^5 exchanges. Table 2 shows ten consecutive measurements of one hundred thousands samples.

samples	latency for 10^5 exchanges (nanoseconds)	mean latency (microseconds)
10^5	362729875	3.627
10^5	362849250	3.628
10^5	362576750	3.625
10^5	362880375	3.628
10^5	362818625	3.628
10^5	362577500	3.625
10^5	362654250	3.626
10^5	362785750	3.627
10^5	362909125	3.629
10^5	362696625	3.626
total	latency for 10^6 exchanges (nanoseconds)	mean latency (micro seconds)
10^6	3627478125	3.627

Table 2. *MessagingUnit* latency

The cost of virtualization using *MessagingUnit*, i.e. three extras messages, is about $3 \times 3.627 = 10.88 \mu seconds$. Given the hardware characteristics of the display panel used for the PoC, an AMOLED with a 720p resolution and with a vertical sync pulse of three lines long, a line is about 795 pixels long. With a pixel clock at 60 MHz, the VBlank period is $39.75 \mu seconds$. Thus, virtualization consumes one fourth of the VBlank period.

3.3 Discussions

General gains, Limitations and restrictions

The overall security architecture based on a separated graphical proxy in a dedicated core, the Protected Core, is not linked to specificities of the ARM architecture. Applications of such graphical proxy architecture are thus possible in other system architectures, including for example RISC-V based SoCs. Yet, it implies that the used SoC includes a companion core, and the ability to have a dedicated isolated security domain for it with strictly separated access and peripherals exclusive attribution. NXP supports this on iMX8 SoCs family, as well as others such as Renesas [24, 25].

In our current PoC design some mechanisms have not been implemented or considered:

— Harmonized support of display rotation:
The display rotation, i.e. the framebuffer contents orientation, is under each Security Domain responsibility. The Android operating system uses the GPU 2D hardware accelerator to rotate its display

as it needs, the graphical proxy managed by the Protect Core having no role here in the chosen orientation. However, in our current design, the Protected Core is not informed of the chosen display orientation, which may be an issue when it manages elements such as a secure bar which will then be always set on the same side, whatever the used orientation. This limitation can be easily lifted if required by adding dedicated messages to the Protected Core to keep it informed of the orientation chosen by Android.

— Secure bar covering a part of the display:
We have identified two possible solutions to add a secure bar:

1. Declare a screen resolution to the main cores cluster peers smaller than the real display resolution (e.g. 1440x900 against 1600x900 pixels). This allows the Protected core to dedicate the height difference (e.g. 160 pixels here) to the secure bar it manages without dropping out any part of the peer framebuffer contents.
2. Keep the very same resolution and always overlay a part of the screen when generating the output framebuffer. The Android world then always has a part of its screen hidden.

The first solution seems more convenient, but is harder in practice as the effective resolutions supported by both the display and the GPU may not be versatile enough to get a proper secure bar rendering. In order to offer proper confidence level to the user, the second solution may be accompanied by a dedicated secure indicator (e.g. a LED), which is directly driven by the Protected Core or the SE. There is however no real software-related differences in term of complexity for each solution.

— Secure bar interactions:
The secure bar managed by the Protected Core can be considered as an always-on dedicated secure graphical panel with which the end user can interact, whatever the currently security domain displayed is. This is easily feasible as the input positions associated to the dedicated screen part can be used for local actions, under the responsibility of the graphical proxy implementer.

— Power management:
Specific power management actions (e.g. battery low warning) are not considered once entered in the Trusted UI in our current implementation. The Protected Core is not informed of a given battery state or potential power off risk. We consider that the Trusted UI will be used for a very short time, with a timeout if

there is no action from the user. The power state should therefore not evolve significantly during this time. The global security impact of power issue still needs to be fully analyzed. If required, Trusted UI cancellation messages sent to the Protected Core by the power management system could be implemented.

Possible evolutions

In-Chip communication with Protected Core In our current architecture, one last hardware component still requires to be dynamically switched between the normal and secured (TrustZone) worlds: the MessagingUnit. Nevertheless, in our hierarchical abstraction, it can be easily modified by moving the hardware mailbox manipulation down to the Security Monitor. Then the Security Monitor itself is responsible for handling the access from both worlds, based on the current state reported by the Protected Core.

The main advantages in implementing this evolution are:

1. The overall SoC-specific implementation is pushed back to the Secure Monitor. The virtual drivers hosted in both Linux and the TEE OS then only use the ARM `SMC` standard calls as underlying interface.
2. The Secure Monitor can keep the Protected Core synchronized with any switch between the normal and secure (TrustZone) world by sending it a dedicated message.
3. The hardware mailbox backend can be locked to the Security Monitor security domain at early stage of boot (as no switch is required).

The possible drawbacks would be:

1. As the Security Monitor is managing requests from both worlds, the peer authentication becomes more critical for the Protected Core, requiring the initial PCG32 usage to be re-analyzed with a consideration for this new design.
2. The latency would increase, as a Secure Monitor call is required for each event in the display control plane. The impact may be significant, typically upon graphic VBlank requests.

From in-Chip to external Protected Core In the case of an out of Chip eSE / Protected core couple, the communication between the main SoC software (Android, Trustzone) and the Protected Core itself would require a more hardened channel as the one described in section 2.2 which offer only limited security for anti-replay and authentication (but which is fine inside a SoC).
The idea here would be to use a secure channel with session keys generated upon each boot of the platform. The data exchanged would then be classified in two categories: the standard ones which have no impact on the security (e.g. change screen luminosity) and sensitive ones relative to the Trusted UI. Having two categories will allow to keep optimal performances in case of limited bandwidth of the communication channel. The sensitive data will always use signature (MAC), and optionally encryption, when exchanged. The secure channel will be based on a security proven but efficient protocol, as for example SCP with AES from GlobalPlatform.
The root keys used for the secure channel will be generated by the SE during the very first boot of the platform in secure production environnement. They will be transmitted to the SoC, and then stored securely in its eFuse. The idea being to use the hardware security mechanism attached to boot level that is in general available with eFuse storage in order to allow access to them only during the first stage of the boot: even if the Applicative Operating System is corrupted, access to them is impossible as hardware locked.
Additionally, to cope with possible weaknesses of the RNG of the SoC, the session key generation performed at the early stage of each boot will make sure to use also RNG values issued from the SE connected to the Protected Core.

4 Conclusion

In this article, we have described a novel architecture based on an heterogeneous in-SoC cores cluster set, in order to enhance the security of user interactions on critical assets managed by any backend Secure-Element, through a Trusted User Interface (Trusted UI) based on high resolution displays (often driven using MIPI-DSI interfaces). Our proof of concept proposes a new hardware-software hybrid architecture that aims to support security-critical user interactions performed with a Trusted UI, while significantly reducing the attack surface compared to traditional ARM TrustZone based Trusted UI architectures.
During the solution design, we strived to maintain overall concept princi-

ples independent of ARM-specific considerations, preserving the possibility of utilizing alternative architectures such as RISC-V for either the main cores cluster or the Protected Core. We have demonstrated that it is possible to fully implement a proxy of the user interface, thus maintaining complete control over user interactions in a secure way, without compromising the graphical performances.
The proof of concept we built still needs refinements in terms of power consumption analysis and user experience design for the Trusted UI part. We are however confident in the ability of the Protected Core, based on a Cortex-M CPU, to have limited power consumption impact.
During the design of our proof-of-concept, we selected the Trusted UI as the first challenging work on which a novel architecture can be designed. But our work also opens doors to implement similar proxies for other security impacting hardware components, such as gyroscopes, light sensors, etc.

References

1. Graphics | Android Open Source Project — source.android.com. `https://source.android.com/docs/core/graphics`.
2. The Linux Kernel documentation — kernel.org. `https://www.kernel.org/doc/html/v5.10/`, 2020.
3. Dirk Balfanz. Fido u2f implementation considerations. *FIDO Alliance Proposed Standard*, pages 1–5, 2015.
4. Nahom Aseged Belay. Securing the boot process of embedded linux systems. Master's thesis, NTNU, 2022.
5. Ryad Benadjila, Cyril Debergé, Patricia Mouy, and Philippe Thierry. From cves to proof: Make your usb device stack great again, 2021.
6. Dave Bullock, Aliyu Aliyu, Leandros Maglaras, and Mohamed Amine Ferrag. Security and privacy challenges in the field of ios device forensics. *AIMS Electronics and Electrical Engineering*, 4(3):249–258, 2020.
7. Yue Chen, Yulong Zhang, Zhi Wang, and Tao Wei. Downgrade attack on trustzone. *arXiv preprint arXiv:1707.05082*, 2017.
8. Janis Danisevskis. Android protected confirmation: Taking transaction security to the next level. `https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html`, 2018.
9. Simone DI BLASI. Development of a touch screen display with haptic functionality and a graphical user interface in a heterogeneous multi-core and multi-processor environment. 2021.
10. Thomas Fischer, Ahmad-Reza Sadeghi, and Marcel Winandy. A pattern for secure graphical user interface systems. In *2009 20th International Workshop on Database and Expert Systems Application*, pages 186–190. IEEE, 2009.

11. GlobalPlatform, Inc. *Trusted User Interface API v1.0*, 6 2013. software interface Specification.
12. Nathaniel Hatfield. Software-based side channel attacks and the future of hardened microarchitecture. 2021.
13. Richard Hayton. The benefits of trusted user interface (tui). `https://www.trustonic.com/technical-articles/benefits-trusted-user-interface/`, 2020.
14. Kocialkowski. Walking Through the Linux-Based Graphics Stack. In *Embedded Linux Conference Europe 2022*. ELCE, 2022.
15. Nikolaos Koutroumpouchos, Christoforos Ntantogian, and Christos Xenakis. Building trust for smart connected devices: The challenges and pitfalls of trustzone. *Sensors*, 21(2):520, 2021.
16. Genode labs. An exploration of arm trustzone technology. `https://genode.org/documentation/articles/trustzone`, 2014.
17. Ben Lapid and Avishai Wool. Cache-attacks on the arm trustzone implementations of aes-256 and aes-256-gcm via gpu-based analysis. In *International Conference on Selected Areas in Cryptography*, pages 235–256. Springer, 2019.
18. Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. {ARMageddon}: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, 2016.
19. Thibault Martin, Nikolai Kosmatov, and Virgile Prevosto. Verifying redundant-check based countermeasures: a case study. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 1849–1852, 2022.
20. René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. The android platform security model. *ACM Transactions on Privacy and Security (TOPS)*, 24(3):1–35, 2021.
21. Melissa E O'Neill. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation. *ACM Transactions on Mathematical Software*, 2014.
22. Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM computing surveys (CSUR)*, 51(6):1–36, 2019.
23. Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 195–209, 2019.
24. Renesas Electronic Corporation. *R-Car-H3ne*, 8 2022. hardware user manual.
25. Renesas Electronic Corporation. *R-Car-M3e*, 8 2022. hardware user manual.
26. Virgile Robles, Nikolai Kosmatov, Virgile Prévosto, Louis Rilling, and Pascale Le Gall. Methodology for specification and verification of high-level requirements with metacsl. In *2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 54–67. IEEE, 2021.
27. Dan Rosenberg. Qsee trustzone kernel integer over flow vulnerability. In *Black Hat conference*, page 26, 2014.
28. Joanna Rutkowska and Rafal Wojtczuk. Qubes os architecture. *Invisible Things Lab Tech Rep*, 54:65, 2010.

29. Keegan Ryan. Hardware-backed heist: Extracting ecdsa keys from qualcomm's trustzone. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 181–194, 2019.

30. Mickael Salaun. *Intégration de l'utilisateur au contrôle d'accès: du processus cloisonné à l'interface homme-machine de confiance*. PhD thesis, Evry, Institut national des télécommunications, 2018.

31. Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. {CLKSCREW}: Exposing the perils of {Security-Oblivious} energy management. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1057–1074, 2017.

32. Natalia Vizintini and Aleksandr Grek. Secure virtual payments.