Forensic Science International: Digital Investigation 37 (2021) 301191

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2021 USA - Proceedings of the Twenty First Annual DFRWS USA

Chip chop — smashing the mobile phone secure chip for fun and digital forensics

Gunnar Alendal [a, *], Stefan Axelsson [a, b], Geir Olav Dyrkolbotn [a]

[a] Norwegian University of Science and Technology (NTNU), Norway
[b] DSV, Stockholm University, Sweden

ARTICLE INFO

Article history:

Keywords:
CC EAL
Mobile security
Digital forensic acquisition
Secure element security
S3K250AF

ABSTRACT

Performing mobile phone acquisition today requires breaking—often hardware assisted—security. In recent years, Embedded Secure Element (eSE) hardware has been introduced in mobile phones, with a view towards increasing the security of critical system features and encrypted user data. The idea being that the eSE should remain secure even if the rest of the system is compromised. The eSE is set to become crucial to modern mobile phone security, challenging Digital Forensics. The eSE is designed to withstand both logical and physical attacks, including side channel attacks, and to keep the attack surface towards the rest of the system/phone small, and complexity low to minimise the risk of implementation errors.

In this paper we adapt current state-of-the-art attacks to the eSE platform and present an attack on an eSE by Samsung, recently introduced in their premium mobile phones. We show how, with limited resources, our approach discovered a vulnerability that could be exploited, leading to a complete compromise of all the eSE security goals and a full loss of future eSE trust, as mitigation of our attack in already fielded devices is challenging. This eSE is Common Criteria EAL 5+ certified and our attack exposes the gap between *intended* and *achieved* security, undermining the implied trust in such certifications.

We explain the eSE security design, the details of our attack, and discuss how a single vulnerability can have such devastating security results. The ultimate result of our research facilitates acquisition of affected devices, demonstrating use of offensive methods in advanced Digital Forensic Acquisition.

Introduction

The increased mandatory security and encryption of mobile phones is challenging digital forensics. This hindrance is discussed in the general media (Venturebeat, 2016; Focus, 2016) as well as research circles (Lillis et al., 2016). Security and encryption seem to be the major challenges in the years to come. Trusted computing (TC) in form of a stand-alone eSE HW, in addition to the existing TrustZone (ARM, 2009), is adding an extra layer of security that needs to be broken. All these security features motivate digital forensic acquisition (DFA) to turn to offensive techniques, like security vulnerability research and exploitation (Alendal et al., 2018).

Trusted computing is the concept where a system is expected to behave as intended, withstanding outside influence, and enforced by trusted, stand-alone hardware and software. The concept is not without controversy and has caused discussion of its benefits, and risks (Fournaris and Keramidas, 2014; Anderson, 2003). However, the idea is still implemented by many vendors, and to support trusted computing, several hardware (HW) solutions exist today. Intel Software Guard Extensions (SGX) (Intel, 2020), Trusted Platform Modules (TPM) (Group, 2020), Trusted Execution Environment (TEE) (Sabt et al., 2015), Hardware Security Modules (HSM) (Mavrovouniotis and Ganley, 2014) and Secure Element (SE) (Vauclair, 2011) are all examples of technology providing physical / HW assisted separation *inside* a system to provide trusted, tamper-proof and secure environments for system critical security elements. One common design principle is the need for a separate root of trust, to prevent security breaches even if the overall system is compromised (Pfleeger, 2009). This isolated system-within-the-system is to be made secure by keeping complexity low, and implementation quality high. One advantage of lower complexity is that the probability of software bugs and side-channel attacks is reduced as a consequence of the smaller code size (Hatton, 1997; Ozment and Schechter, 2006). Increased quality can be achieved by

* Corresponding author.
E-mail address: gunnaale@stud.ntnu.no (G. Alendal).

https://doi.org/10.1016/j.fsidi.2021.301191

improving development methodology, e.g. by working according to certain standards, such as those meeting Common Criteria Evaluation Assurance Level (CC EAL) certification requirements (Criteria, 2020). The intention being that a higher CC EAL level increases the reliability of the security features implemented.

In general terms, the eSE concept consists of specialised HW providing certain system critical security features to the host system without depending on that host system for any execution of code nor storage of data. This "black-box" principle means the eSE has full control of its own processor, RAM and storage. This setup is meant to prevent a compromised host system from reading the eSE embedded code and data, and to make it more difficult to perform side-channel attacks, like observing or influencing execution of eSE sensitive code.

An advantage of this physical separation is that development and production of eSE HW can be outsourced to specialised vendors with a secure production environment. The host system vendor need only to follow the documented eSE interface to incorporate it in end products. However, one major drawback is that this approach risks the introduction of a single point of failure. A failure in the eSE can have devastating effects on the operation of all systems using the eSE as a basis for their security. Another drawback is that the host system vendor needs some form of trust in the eSE HW, to certify that the eSE security features are securely implemented and working as intended. This is one of the intentions of performing a CC EAL certification.

A concept corresponding to eSE was presented in the Android Operating System (OS) version 9. Mayrhofer et al. (2019) explain Google's views on the "The Android Platform Security Model", discussing a. o. the different threat model for mobile devices. They discuss the use of a *strongbox* that "… implements the Android keystore in separate tamper resistant hardware (TRH) for even better isolation. This mitigates [T1] and [T2] against strong adversaries …" (Mayrhofer et al., 2019, p. 8). Their definition of threats [*T1*] is "Powered-off devices under complete physical control of an adversary (with potentially high sophistication up to nation state level attackers), e.g. border control or customs checks" and [*T2*] is "Screen locked devices under complete physical control of an adversary, e.g. thieves trying to exfiltrate data for additional identity theft." (Mayrhofer et al., 2019, p. 3). T1 and T2 clearly identifies the most advanced and resourceful adversaries. We will use the term eSE in place of Google's term *TRH* for consistency throughout this paper.

In this paper we present a remote attack on a state-of-the-art eSE HW utilised by the major Android mobile phone vendor Samsung. The attack is *remote* as we attack the logical interface, as opposed to *local* attacks in need of physical access. Our attack bypasses the security of the eSE, protecting sensitive encryption key material, and facilitates digital forensic acquisition (DFA) of user data. This attack will work on powered off devices, known as the *before-first-unlock* (BFU) state, with no knowledge of user credentials. We show that although placing all trust in a single, well protected, entity may be tempting, it also means the introduction of a single point of failure, and if done wrong the whole trusted computing design falls, leaving the system totally exposed. This eSE is present in Samsung's high-end mobile phone models and represents the state of the art in modern Android security. The eSE HW is CC EAL 5+ certified, and is thus expected to provide a very high level of security. Samsung uses CC EAL certifications to promote the security of their eSE (Samsung, 2020d) and also to justify the high security level of the Samsung Galaxy S20 mobile, needed in mobile eID solutions for use in Germany (Samsung, 2020b; für Wirtschaft und Energie, 2020). CC EAL certifications have been proven problematic by other authors as well (Nemec et al., 2017; Moghimi et al., 2020), and our attack shows that such certifications are no guarantee the proper security level has been achieved. Our attack demonstrates the failure of all CC EAL 5+ goals for the eSE HW. Further, our analysis shows that patching isolated eSE HW is challenging, making it hard to regain the expected CC EAL 5+ security level in already shipped mobiles.

Our contribution can be summarised as:

- The adaptation and improvement of state-of-the-art black-box attack techniques applied to the eSE HW platform. The standalone eSE significantly changes the attack path compared to conventional TEEs, like ARM TrustZone implementations.
- The discovery of previously unknown, *remotely exploitable*, security vulnerabilities that fully breaks the confidentiality and integrity of the CC EAL 5+ certified eSE HW.
- A demonstration of the gap between the *intended* and *achieved* security, and how certifications, like Common Criteria EAL, fails to deliver the needed trust in implemented solutions.
- A presentation of the full attack development and exploitation of the eSE, with example attacker use.
- Analysis of the effect of a vulnerability exploit in the eSE HW and the lack of eSE countermeasures.
- A demonstration of digital forensic goals: off-device brute force of user screen lock credential, necessary for digital forensic acquisition of encrypted user data.

The rest of this paper is organised as follows. In Section "Background" we introduce needed background and the targeted eSE. Related work is discussed in Section "Related Work" and Section "The Attack" contains the attack steps performed and the technical details on the vulnerability and its exploitation. In Section "Attack Implications" we present example implications of the attack. Finally we will present our discussion and conclusions in Sections "Discussion" and "Conclusion and Future Work".

Background

In this section we introduce some needed background material. First an introduction to the specific eSE HW targeted by our attack and its CC EAL 5+ certification. The eSE threat model is then discussed to clarify our attack approach, communicating with the logical interface using a protocol based on the "Application Protocol Data Unit" (APDU). Refer to "APDU primer" in Appendix for a brief APDU introduction.

Embedded Secure Element

The eSE HW under investigation is the Samsung S3K250AF embedded Secure Element (Samsung, 2020d). This eSE was introduced in February 2020 by Samsung, with the release of the Samsung Galaxy S20 product line. Our test devices were the Samsung Galaxy S20 Ultra 5G (SM-G988B), the Samsung Galaxy S20 (SM-G980F) and the Samsung Galaxy Note 20 Ultra 5G (SM-N986B). All these models use the Exynos SoC. The upcoming Galaxy S21 models with the Exynos SoC are also believed to include the S3K250AF, but this has not been confirmed at the time of writing, and was not part of our research.

We will mostly refer to the "S3K250AF eSE" simply as the "eSE" throughout the rest of the paper.

The S3K250AF eSE is a single chip solution, soldered to the printed circuit board (PCB) of the mobile. It has a small form factor, pictured in the Samsung promotion material (Samsung, 2020d). The eSE processor is an ARM SecurCore SC000 (Limited, 2020b), according to the NIST Cryptographic Algorithm Validation Program (CAVP) for the S3K250AF (NIST, 2020). The architecture is ARM BE8 mode (Limited, 2020a). This architecture uses little-endian for code and big-endian for data and pointers. The S3K250AF contains 252 kilobytes (kB) on-board flash storage, according to the CC EAL documents (Samsung, 2020a). Samsung promotes an eSE standard development kit (SDK) (Samsung, 2020c), but we have not evaluated this SDK as this entails us signing a non-disclosure agreement.

CC EAL

The S3K250A holds a CC EAL 5+ certification (commoncriteriaportal.org, 2020) from Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI) (de la Sécurité des Systèmes d'Information, ANSSI). The certification is accompanied by two documents. The security target (ST) document (Samsung, 2020a) by Samsung describes the S3K250A and its security requirements, and the second document (AG et al., 2014) by third parties describes the intended protection profile, which is generic and not specific to the S3K250A.

The main security goals for the eSE (SG1-SG3) ((Samsung, 2020a, p. 46)) are to maintain *integrity* of user data (SG1), to maintain *confidentiality* of user data (SG2), and to maintain *correct operation* of the services provided by the eSE (SG3). So an attacker should not be able to change any stored eSE data, read any stored eSE data (without authorisation), and not influence the operation of any of the eSE features offered.

The CC EAL 5+ certification is an aid to achieving these goals, and it states "Certification does not in itself constitute a recommendation of the product by the National Information Systems Security Agency (ANSSI), and does not guarantee that the certified product is completely free from exploitable vulnerabilities."[1] (de la Sécurité des Systèmes d'Information, ANSSI, p. 2). As such a guarantee is impossible to give, some effort has been done to lower the probability of the existence of such vulnerabilities. One such effort is the Common Criteria *Advanced methodical vulnerability analysis* (AVA_VAN) (Criteria, 2017). This vulnerability assessment aims to determine potential vulnerabilities. AVA_VAN is divided into levels ranging from 1 to 5 with "increasing rigour of vulnerability analysis by the evaluator and increased levels of attack potential required by an attacker to identify and exploit the potential vulnerabilities" (Criteria, 2017, p. 184). Level 5: "AVA_VAN.5 Advanced methodical vulnerability analysis", is the highest level. This level specifies that "A methodical vulnerability analysis is performed by the evaluator to ascertain the presence of potential vulnerabilities." (Criteria, 2017, p. 188). AVA_VAN.5 is part of the S3K250A CC EAL 5+ certification (de la Sécurité des Systèmes d'Information, ANSSI, p. 3). Thus AVA_VAN.5 is a best effort to reveal any vulnerabilities of the S3K250A. It is unclear to us what exact analysis steps were performed by the evaluator in this particular case, but AVA_VAN.5 is referenced in the certification document (de la Sécurité des Systèmes d'Information, ANSSI), assuring that sufficient analysis was performed to achieve a CC EAL 5+ certification with AVA_VAN.5.

eSE threat model

Adapting the threat model of Mayrhofer et al. (2019), we consider the eSE against threats [*T1*] and [*T2*], as these are the threats this TRH / eSE is designed to mitigate. These scenarios assume an attacker with physical control of the eSE. Attacking an isolated HW component, like the eSE, two main attack vectors present themselves: The logical interface between eSE and the host system, known as the Rich Execution Environment (REE), and (possibly HW assisted) side-channel attacks on the eSE. The logical interface between the eSE and REE uses "Application Protocol Data Units" (APDU), originally a communication protocol for smart-cards. APDU based communication, accepting attacker commands and data, could be vulnerable to design and implementation bugs. A simplified view of the logical interface between eSE and the REE is shown in Fig. 1.

Related Work

Attacks on black-box physical separation implementations are not new. Anderson et al. (Anderson et al., 2006; Bond and Anderson, 2001) discusses the security of tamper-resistant cryptographic processors. They discuss their use and attacks with focus on two different attack scenarios: attacks involving physical access and logical attacks. Attacks involving physical access are referred to as *local* attacks and most side-channel attacks fall into this category, needing some physical interaction to mount an attack. Logical attacks are referred to as *remote*: These are attacks on the logical interface and they do not require physical access, and are thus independent of the distance between the attacker and the attacked device. Anderson et al. refer to these attacks as *API* attacks, using the provided Application Programming Interface (API). Exploitation of design and code flaws fall into this category. Anderson et al. discuss several attacks, including a cryptographic API attack on the IBM 4758 cryptoprocessor. The attack demonstrated design flaws leading to information leaking via the API, which could be used to mount a brute force attack on embedded DES keys. The security of the IBM 4758 HW was rendered moot because of flaws in the software running on the device. Anderson et al. predict in their conclusions that logical attacks "..are likely to remain the weak spot of most high-end systems".

More advanced attacks via the logical interface, relevant to physical separation implementations, can be found in more recent research. Bittau et al. (2014) demonstrate how to write a remote buffer overflow attack without knowledge of the target binary. Where traditional attacks use known *gadgets* within the target binary to craft ROP attacks, Bittau et al. improve on this technique by using a so called blind ROP (BROP). The BROP technique can be used to attack closed source and unknown implementations using leaked information. Thus useful ROP gadgets can be found simply by trial and error, building a complete attack using only simple information leak oracles, like a program crash. Lee et al. (2017) use a similar approach to attack Intel SGX Secure Enclaves. Their attack, named *Dark-ROP*, uses information leak oracles from the Intel SGX to locate ROP gadgets and from there to build a functional ROP attack against selected Secure Enclaves. Dark-ROP demonstrates that critical implementation errors in secure enclave code can still be exploited by attackers, without knowledge of the target code. Van Bulck et al. (Van Bulck et al., 2018; Weisse et al., 2018) demonstrate another powerful attack, *Foreshadow*, attacking the CPU cache to retrieve secure enclave secrets. There are several published papers on the security of Intel SGX (Jang et al., 2017; Biondo et al., 2018; Schwarz et al., 2019; Nilsson et al., 2020).

Moghimi et al. (2020) recently demonstrated an attack on TPMs, some CC EAL 4+ certified. Their attack uses black-box timing analysis to reveal secret key information during signature generation based on elliptic curves. Using this attack they demonstrate retrieval of 256-bit private keys. A key element in their attack is the

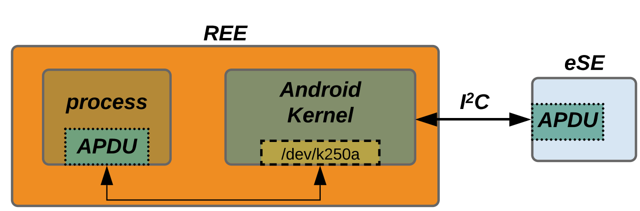

Fig. 1. eSE logical interface using APDU.

[1] Our translation from French.

magnitude of increased operating frequency of the main SoC compared to the TPM, facilitating high frequency timing of the “slow” TPM execution.

Numerous attacks exist on TrustZone implementations (Beniamini, 2016a, 2017, b; Chen et al., 2017), demonstrating that code vulnerabilities, like design and coding quality, are crucial for security, often with devastating effect on security when such vulnerabilities are found. Cerdeira et al. (2020) have summarised current security challenges of TrustZone-based TEE systems.

The Attack

The completely stand-alone eSE HW affects how an attack can be designed and performed. Compared to attacks published on other secure execution environments, discussed in the previous section, this requires a different approach. The major difference is the changed attack surface, requiring a different attack chain, with new attack oracles.

Our attack adapts elements from both BROP by Bittau et al. (2014) and Dark-ROP by Lee et al. (2017) to the physical separated black-box eSE HW, and we are, to the best of our knowledge, the first to do so. Although partly available for this particular eSE, our attack does not require knowledge of the binary (FW). We incorporate information leak oracles to aid in the attack on the eSE HW.

The attack was developed following these generic steps:

- **Information Gathering** Gain knowledge of the target eSE and how it is used.
- **Identify Attack Vectors** Gain knowledge of the eSE attack surface with potential attack vectors.
- **0-day Information Leakage** Locate at least one information leakage oracle to aid in 0-day vulnerability discovery.
- **0-day Vulnerability Discovery and Exploitation** Locate at least one new exploitable vulnerability and use the discovered vulnerability to break confidentiality (secure data exposure) and/or break integrity (writing to code or data memory).

The resulting attack on the Samsung S3K250AF eSE HW (Samsung, 2020d) will follow, with the last section discussing the technical capabilities of our attack.

Attack Assumptions

Our attack is based on the assumption that we have access to the logical interface of the chip. This logical interface is exposed by the `/dev/k250a` virtual device (see Fig. 1). Access to this device enables the attacker to communicate, using APDUs, with all exposed functionality of the eSE HW. Thus, in this case, we can operate as a privileged REE process similar to the process depicted in Fig. 1. In a test environment this is achieved simply by executing a binary we provide with system privileges. We implemented all attack functionality to communicate with the eSE. We call this tool `chip_breaker`. Our setup executed this tool through a “root” adb shell (Gunasekera, 2020), connected to test devices either with a cable or over a network connection. In a more realistic attack scenario, depending on how the attacker gains access, this can be achieved by infecting a process with system privileges and then communicating with the eSE. The next section identifies one such target process.

Our assumption seems realistic, as the design of the eSE is to withstand attacks against a fully compromised REE. Note that we do *not* require physical access to the chip, which might be a prerequisite for many side-channel attacks. Hence, our attack can even be performed remotely, over the air, assuming we have gained privileges to communicate with the eSE logical interface. So our attack can be performed using any remote, local or physical attack that gains elevated execution, like “root”, on the device. Elevated execution can be achieved without triggering user data wipe. One path is to break the secure boot of the device to introduce attacker code (Chao et al., 2020; Alendal et al., 2018). As history has shown that gaining such access is not necessarily difficult or uncommon (Google, 2020b), we do not address that problem further in this paper. Even de-soldering the eSE chip and communicating directly on the I2C lines is an option to perform our attack.

Information gathering

Several important initial information sources were identified:

- CC EAL certification documents (Samsung, 2020a; de la Sécurité des Systèmes d’Information, ANSSI; AG et al., 2014).
- An android service process, `hermesd`. This privileged process communicates with the eSE using the APDU-based logical interface and it is the only REE process with this ability (Fig. 1). Processes communicating with the eSE were revealed by observing access to the eSE virtual device, `/dev/k250a`.
- Vendor specific libraries supporting `hermesd`. The most important being `libese-grdg.so`. This library implements the low level communication with the eSE. This communication uses APDUs. APDUs are communicated over the eSE logical device `/dev/k250a`.
- FW files found to be accessed by `hermesd`: `/vendor/etc/secnvm/k250a_00000009.img` and `/vendor/etc/secnvm/k250a_00000009_dev.img`.

These files contain partly encrypted FW updates for the eSE. These files were revealed by observing files accessed by `hermesd`.

Unencrypted parts of `k250a_00000009.img` and `k250a_00000009_dev.img` revealed code in ARM THUMB mode (Limited, 2020c). The file `k250a_00000009.img` was assumed to be a “production” FW container. We refer to this as `FW_prod`. Correspondingly the `k250a_00000009_dev.img` is assumed to be a “development” FW container. We refer to this as `FW_dev`. Our research only recovered one version each of both these files, on all tested models, and analysed model FW (Appendix, Table 1).

We inspected the partially unencrypted `FW_prod` and `FW_dev`. These turned out to be container files for different “images” for the eSE. The different image names are: `BOOT`, `CRPT`, `CORA`, `CORB`, `SNVM`, and `IWEA`. We developed a simple script to parse and extract images from this proprietary container format (Appendix, Table 1). This revealed that most of the images are encrypted, while the images `SNVM` and `IWEA` are not. Images `SNVM` and `IWEA` are also signed, thus an attack on these images using simple FW modifications seems less probable. In later attack steps we recovered the encryption key to the encrypted images, and the decrypted images all included image signatures (Section “Attack Capabilities and AES Key Exposure”).

The logical eSE interface attack vector

The logical eSE interface utilises APDUs for communication (Appendix, “APDU primer”). Thus all eSE APDU communication is considered a potential attack vector and we need to expose as many eSE APDU handlers as possible. These APDU handlers are implemented by code running on the eSE ARM processor. The handlers will potentially accept attacker controlled input, which could lead to an input validation vulnerability. In addition to all APDU handlers, the APDU transport layer is an additional attack vector. Both the APDU handlers and the APDU protocol handling are part of the logical eSE interface.

All valid APDU CLA and INS values correspond to APDU handler functions within the eSE code. Observing the `hermesd` process communicating with the eSE using APDU and reverse engineering the REE library, `libese-grdg.so`, enabled us to reveal the communication logic between the REE (`hermesd`) and the eSE. The exposed eSE specific functions in `libese-grdg.so` are listed in Table 2 (Appendix) with their corresponding `grdg_*` name. These functions revealed valid APDU CLA and INS values, each communicating with different APDU handlers inside the eSE. As these functions only expose eSE features utilised by `libese-grdg.so`, additional eSE APDU handlers might exist. Some were indeed exposed by brute force of the APDU logical interface. By design, all the different APDU CLA and INS handlers inside the eSE are expected to return valid SW values, indicating success or various error states. Gkaniatsou et al. (2015) demonstrated REPROVE, a system to aid in the reverse engineering of APDUs used in smart-cards. Inspired by their work, we produced a simple brute force process shown in Listing 1, simply trying various combinations of (CLA,INS) pairs and observing returned SWs. The unknown SW response "unknown_command" classification is vendor implementation dependent and might vary from vendor to vendor, and even from CLA to CLA. However, it should be easily spotted as being the most common SW reply from a specific (valid) CLA and random (thus most probably not implemented) INS.

```
for ( all possible CLA ) {
  for ( all possible INS ) {
    SW = APDU_communicate_with_eSE(CLA, INS)
    if ( SW != unknown_command ) {
      // potential valid (CLA,INS) found
      // optional next step:
      P1_P2_Lc_Data_Le_brute_force(CLA, INS)
    }
  }
}
```

Listing 1 Simple APDU brute force pseudo code

Be warned that brute forcing valid APDU handlers might trigger an unwanted effect in the eSE if a valid (CLA,INS) pair is hit with valid P1, P2, Lc and Le values. One example could be a "factory reset" APDU, not in need of any valid P1, P2, Lc or Le values. Thus unknown (CLA,INS) pairs with SW values indicating success, `0x9000`, should be treated with some caution.

Table 2 in the Appendix lists eSE APDU handlers discovered through reverse engineering the `libese-grdg.so` library, APDU brute forcing, and confirmed by reverse engineering of the dumped eSE flash recovered later in the attack (Section "Arbitrary ash and RAM read"). Knowing the available APDU handlers for the eSE allowed us to establish communication with eSE using its own protocol. All APDU handlers could potentially be exploited to have eSE perform unintended actions and is the most important attack vector for this eSE.

0-day information leak oracles

Attacking a black-box entity like this eSE requires "blind" attack techniques, as introduced in Section "Related Work". Such attacks depend on information leaks from the device (oracles). That is, an attacker needs a way to know if an attack vector behaved unexpectedly, such as a crash. Any observable execution specific information from the eSE could potentially be a useful oracle. Potential oracles are e. g crash dumps, exception handling, page fault addresses, execution timing, etc. Such oracles could leak valuable information in the trial-and-error progress of a "blind" attack. For an eSE this could mean leaking information on code addresses, stack addresses, code content, data content, etc.

The physical separation of eSE makes such observation of (erroneous) behaviour challenging, reducing the existence of oracles. With a stand-alone eSE there is no returned crash response, no exception handler observation, no observable page faults, etc. Timing attacks can also be difficult, measuring execution time from outside the eSE. In our case we looked for logical information leak oracles, where information could be obtained through observable (mis)behavior by the normal logical interface.

We identified two information leak oracles that both play an important role in the attack.

Oracle 1

The first oracle is a common observable behaviour in black-box implementations: the lack of response. This is often the result of a crash. This is also the case with this eSE, which is expected to always reply with a status word (SW). Thus any crashing APDU handler will result in no SW being returned (a timeout error).

Oracle 2

An attacker can also try to look for logical information leak oracles in the ADPU handlers. Candidate APDU handlers are especially those that read and write data. If any of these functions can be manipulated to return more data than expected, leaked information can be used to mount a ROP attack.

Candidates are all `get`, `put`, `read` and `write` functions in Table 2 (Appendix).

The eSE handlers corresponding to `libese-grdg.so` functions `grdg_readWeaver` and `grdg_writeWeaver` were identified as an information leak oracle, when combined. We could not use the `libese-grdg.so` functions `grdg_readWeaver` and `grdg_writeWeaver` directly because of checks performed by the library before submitting the APDU to the eSE, so we re-implemented these REE functions. Our `chip_breaker` tool contains new versions of `grdg_write-/readWeaver` named `chip_breaker_write-/readWeaver`. We call the corresponding eSE APDU handlers `APDU_write-/readWeaver`. One implementation of these two eSE ADPU handler functions can be found in the `IWEA` image in `FW_dev` (Appendix, Table 1). These eSE handler functions could together become an oracle in the following way: The eSE `APDU_writeWeaver` receives two buffers of data from `chip_breaker_writeWeaver`: `CHALLENGE` and `SECRET`. `APDU_readWeaver` would send back a `SECRET` buffer from the eSE iff the caller submitted a matching `CHALLENGE`, written to on-board storage with `APDU_writeWeaver`. The oracle revealed itself by manipulating a `SECRET` length of >`32`, as this seemed to be a fixed length used inside the eSE. The `SECRET` size variable sent is only one byte, so `SECRET` length can be in the interval >`1` and <`256`. Thus `APDU_readWeaver` would return a `SECRET` buffer with up to `256` bytes of data. This leaked valuable stack data.

A stack leak from this oracle can be seen in Fig. 2.

Thus we had two information leak oracles: lack of APDU response if the eSE crashes (Oracle 1) and a stack leak from `APDU_writeWeaver`/`APDU_readWeaver` (Oracle 2).

The Oracle 2 stack leak in Fig. 2 gave us valuable information, indicating memory pointers at offsets `0x44` (`0x20001428`), `0x48` (`0x200027c0`), `0x50` (`0x20001480`), `0x5c` (`0x20001480`), and so on. Keeping in mind this is ARM BE8, memory pointers are 32 bit big-endian. This makes these point to memory locations all in the `0x2000xxxx` range. Further, code pointers can be found at offsets `0x58` (`0x000285f9`), `0x64` (`0x0002858b`), `0x78` (`0x00010423`), and so on. The reason is that they can all be interpreted as 32-bit ARM BE8 THUMB mode addresses, where the least significant bit (LSB) is always 1 to indicate THUMB mode to the processor. These code pointers are `POP`'ed from the stack during a typical ARM THUMB function epilogue: `POP {PC}`.

The leaked addresses gave valuable information both for further reverse engineering efforts and for exploitation.

```
0000   53 65 63 72 65 74 00 00   00 00 00 00 00 00 00 00
0010   00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
0020   00 00 00 01 00 00 00 D0   00 00 00 00 00 00 00 00
0030   00 00 00 00 00 00 00 00   00 00 00 01 00 00 00 05
0040   01 22 49 31 20 00 14 28   20 00 27 C0 00 00 00 00
0050   20 00 14 80 FF FF FF FF   00 02 85 F9 20 00 14 80
0060   20 00 27 C0 00 02 85 8B   00 00 00 00 20 00 0B 50
0070   00 00 00 00 FF FF FF FF   00 01 04 7F 00 00 00 00
...
```

Fig. 2. eSE stack leak using the APDU_writeWeaver / APDU_readWeaver oracle.

0-day vulnerability discovery and exploitation

Oracle 2 is crucial for both vulnerability discovery and revealing information to further understand the attack vectors, but more importantly to reveal information needed for successful exploitation, revealing memory addresses for use in for example a ROP attack.

Oracle 2 was further developed by submitting larger SECRET buffers to APDU_writeWeaver, and not only to manipulate the returned size in APDU_readWeaver. Submitting a SECRET buffer larger than 84 bytes led to Oracle 1 activating with no reply from the eSE. This indicated a crash and we assumed from the leaked stack contents in Fig. 2 that we were overwriting important stack pointers. However, since Fig. 2 shows the leaked stack from APDU_readWeaver, this did not necessarily match the stack of APDU_writeWeaver. Without knowledge of APDU_writeWeaver code, we could now implement a simple brute force attack for secret[84:88] based on the assumption that this was a code pointer and not a data pointer. If this was the case, there should be at least one address that responds with a SW, indicating an attacker controlled ROP. The leaked stack from Fig. 2 already gave valuable ranges for brute forcing. Having access to the IWEA code image extracted from FW_dev, this step can also be solved by reverse engineering the eSE APDU handlers APDU_writeWeaver and APDU_readWeaver. We manually estimated the stack use by both eSE handlers and adapted to any changes between the two. This enabled us to correctly guess the stack layout of the APDU_writeWeaver function based on observation of the APDU_readWeaver stack leak.

Analysing the trigger of Oracle 1 showed that APDU_writeWeaver suffered from a standard stack buffer overflow (One, 1996), enabling a full overwrite of the IWEA slot storage (SECRET + FOOTER) and then APDU_writeWeaver stack data. Fig. 3 shows a simplified view of the effect of the buffer overflow: The first 32 bytes are written to the normal SECRET buffer. The next 36 bytes overwrite the FOOTER and the next 16 bytes overwrite values of registers R4-R7 stored on the stack. Finally, the next 4 bytes overwrite the stored LR register, which will get POP'ed into PC when APDU_writeWeaver returns. This leads to the now well known subversion of control flow and could be used for a ROP attack.

Arbitrary flash and RAM read

The APDU_writeWeaver buffer overflow can be used to read flash and RAM memory by locating a special ROP gadget that takes an attacker controlled address as input and will return 16 bytes. The ROP gadget in Listing 2 can be set up by crafting the stack overflow with correct values of R4-R7 which are identical to those stored on the stack for APDU_readWeaver (Fig. 2). This is due to the semi-static nature of the eSE running with a 100% predictable execution and memory layout. So we control R4-R7 and PC, which is set to the address of the ROP gadget.

```
MOVS    R0, #0x10     ; size to read
STR     R7, [R4]      ; Store address
STR     R0, [R4,#4]   ; Store size
MOVS    R0, #0x90     ; SW1
STRB    R0, [R4,#8]   ; Store SW1
MOV     R0, R5        ; SW2
STRB    R5, [R4,#9]   ; Store SW2
POP     {R1-R7,PC}    ; pop and return
```

Listing 2 ROP gadget for arbitrary ash and RAM read

This simple ROP gadget can be used to read the full flash and RAM of the eSE by setting the R7 to the address to read 16 bytes from, and iterate. Indeed, we used this ROP gadget to read both the complete eSE flash and RAM. The resulting layout of the dumped 252K eSE flash can be seen in Fig. 4. The code image names, 0–8, are matched with the corresponding image names from the FW file FW_dev. The names of the secure storage data images, 9–12, are based on reverse engineering code images and their use of various secure storage addresses.

Arbitrary code execution

The APDU_writeWeaver buffer overflow can even be used to execute attacker provided code. As there is no NX or other “no execute” protection of the eSE stack memory, we can simply execute supplied shellcode. Embedding ARM code in the SECRET buffer and setting the PC to this stack address will execute arbitrary attacker controlled code in the eSE. The address of this stack buffer was located by using the ROP gadget (Listing 2) to dump the stack memory, in the range 0x20000000 - 0x200002800.

This provided us with full read *and* write control of the eSE HW flash and RAM. All code and secure storage from Fig. 4 could thus be read and written to. The use of this exploit is demonstrated in Section “Attack Capabilities and AES Key Exposure” and Section “Attack Implications”.

Our developed chip_breaker tool fully implements the exploit of this vulnerability, executing any provided shellcode on the eSE processor.

Persistence

The code images BOOT, CORA, CORB, CRPT, SNVM, and IWEA are all stored unencrypted and unsigned on the eSE flash. The only integrity checks performed on any image after flash write (as part of a FW update) are simple CRC32 and SHA256 hash verifications. These hashes are also stored on the eSE flash. This means that we can freely modify any code image on the flash and simply update the corresponding integrity check hashes. This means that the eSE has no root-of-trust and there is no secure boot present. The consequence is that there is no way the eSE can verify any code stored on the eSE flash during boot, where the eSE starts executing on-board ROM before continuing execution of BOOT. This BOOT image is writable by us, without any signature verification, and this completely breaks the code trust of the eSE. We confirmed this by developing a writeflash shellcode that was capable of modifying any code image. These changes were persistent across reboot of the device and thus reboot of the eSE. This shellcode was tested with our chip_breaker tool.

Attack Capabilities and AES Key Exposure

With the full dumping of eSE flash (Section “Arbitrary ash and RAM read”), all eSE secure storage is now readable to us (data images 9–12 in Fig. 4). Also, full reverse engineering of the eSE

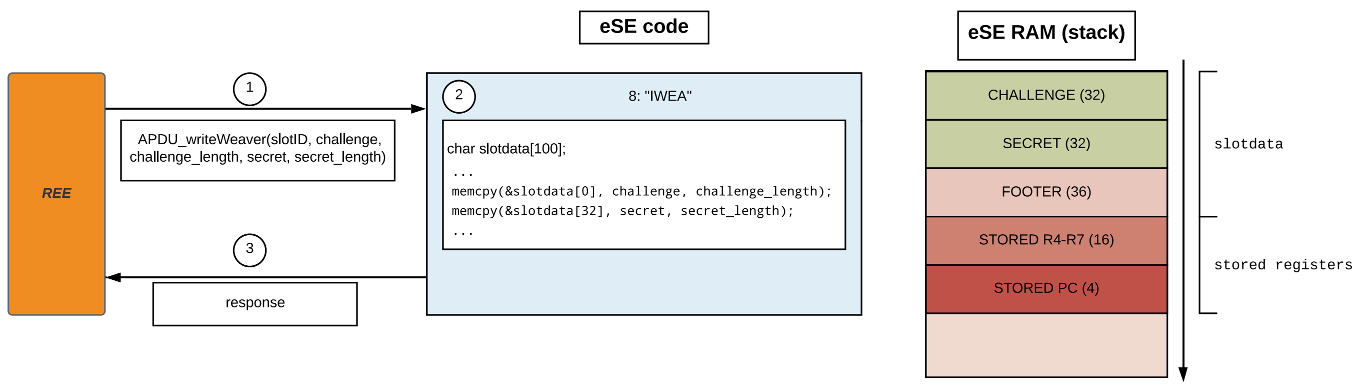

Fig. 3. Buffer overflow in eSE APDU_writeWeaver handler.

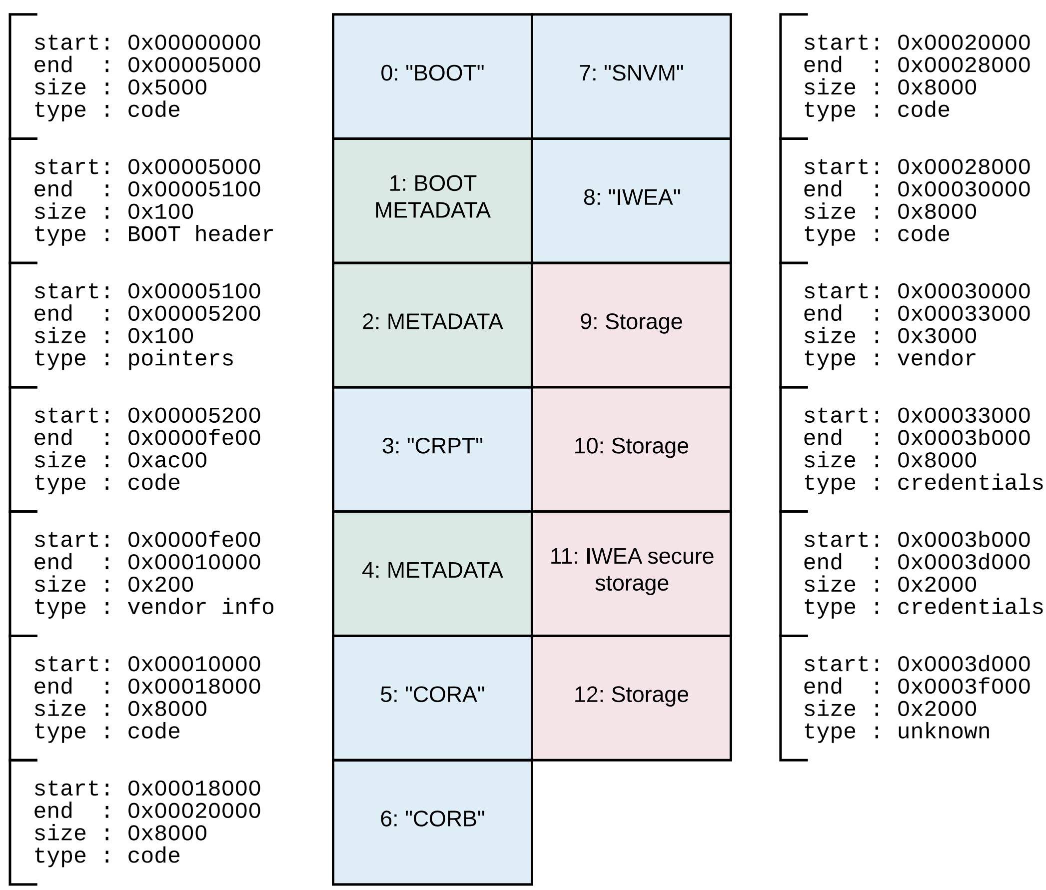

Fig. 4. Full eSE flash layout.

images BOOT, CORA and CORB is now possible. These images are not encrypted on the eSE flash, which suggests that they are decrypted as part of the FW update process. This turned out to be performed with an embedded eSE AES key and initialisation vector (IV) embedded in the dumped BOOT and CORA images. As this key is now exposed by our attack, any attacker with knowledge of this key can decrypt any previous, and future, FW updates for the eSE. We verified this by decrypting the BOOT, CORA and CORB images in the FW_dev FW file (Appendix, Table 1). Updating this pre-shared AES key and IV can thus not be done by supplying a new eSE FW update file, as part of a normal over-the-air (OTA) phone FW update, as this would leak the new key to an attacker already aware of the present one. This update can only be done by a secure update mechanism, such as physically attaching to the eSE HW at a secure vendor site.

Although our attack has fully compromised the security of the S3K250AF eSE HW, our research is by no means exhaustive. More eSE FW security vulnerabilities might exist, including the previously encrypted eSE images, now available for vulnerability research. Our research did not evaluate any side-channel attacks and such vulnerabilities might also exist, as our research identified non-constant time execution functions in the eSE FW. One example is the data dependent execution of the internal memcmp() functions, used for example in authentication functions (grdg_readWeaver in Table 2). As the S3K250AF, containing the exposed vulnerability, can be flashed with arbitrary researcher provided code, it is an ideal research platform for future research of the eSE HW and its resistance against side-channel attacks.

Attack Implications

Our full compromise of the eSE has devastating effects on the system security of affected devices. All eSE security features are made moot by our attack, as an attacker can read and write arbitrary flash and RAM, in addition to making persistent changes to any code and data stored in the on-board flash. This is trivial due to the eSE lacking security features like NX, ASLR, Stack canaries and secure boot. All code in the eSE is running in a single thread of execution, with no privilege separation. This means that a single compromise, like that demonstrated by our attack, gives access to all code and data from both flash and RAM.

In this section we demonstrate a confirmed example of a security feature that fails as a consequence of our attack. We note that Android Keymaster and device attestation also seem to be affected by a vulnerable eSE as both features seem to rely on eSE security features. However, we have not confirmed this.

The following example has been implemented and verified as working.

Android user screen lock brute force

This section demonstrates how to recover the user screen lock credential. The user screen lock credential is used, together with the encryption key material contained in the eSE secure storage, to reproduce the Credential Encrypted (CE) storage encryption key needed for Android's file-based encryption (FBE) (Google, 2020c). The CE storage contains most of the sensitive user data on the device and thus the eSE is crucial in protecting the needed key material. Recovering the screen lock credential is therefore mandatory to facilitate digital forensic acquisition (DFA) of powered-off devices and devices seized before the user has unlocked the device at least once since power on, known as the *before-first-unlock* (BFU) state.

The Android user screen lock protection supports the use of a “weaver” hardware abstraction layer (HAL). Google has documented this HAL (Google, 2020d). The documentation states that the weaver provides secure storage for secret values and that these may only be read if a corresponding key, or *challenge*, has been provided. The S3K250AF eSE provides the weaver functionality in the IWEA image (Fig. 4), accessible through the grdg_writeWeaver and grdg_readWeaver functions in libese-grdg.so (Appendix, Table 2). With our attack in Section “The Attack” an attacker can read of all the eSE secure storage, including storage belonging to IWEA (image 11 in Fig. 4). This means that sensitive IWEA storage belonging to the Android user screen lock protection can be read by an attacker without knowledge of the corresponding challenge. A fragment of the Google screen lock verification code (Google, 2020a) running on affected test devices can be seen in Listing 3.

```
// Weaver based user password
...
result.gkResponse = weaverVerify(weaverSlot,
    ↪ passwordTokenToWeaverKey(pwdToken));
if (result.gkResponse.getResponseCode() !=
    ↪ VerifyCredentialResponse.RESPONSE_OK) {
    return result;
}
...
applicationId = transformUnderWeaverSecret(
    ↪ pwdToken,result.gkResponse.getPayload());
```

Listing 3 unwrapPasswordBasedSyntheticPassword () code

A user-entered credential, a pattern, pin or password, is transformed by a key derivation function (KDF) into pwdToken, which again is transformed into a CHALLENGE by passwordTokenToWeaverKey(). This CHALLENGE is verified by the eSE using grdg_readWeaver. If the eSE successfully verifies the CHALLENGE, weaverVerify() will return the corresponding eSE stored SECRET, accessible through the call result.gkResponse.getPayload(). Both the pwdToken, derived from CHALLENGE, and SECRET are needed in the screen lock verification. These are also used to unlock the encryption keys used for the on-device file-based encryption (FBE) of user data (Google, 2020c).

As we can bypass the weaverVerify verification step and instantly retrieve the correct CHALLENGE and SECRET from the eSE, off-device brute force of user credentials can be achieved by performing a brute force attack outlined in Listing 4. The passwordTokenToWeaverKey() simply produces a SHA512 hash and KDF() is currently scrypt(). The salt can be retrieved from the on-device file /data/system_de/0/spblob/<id>.pwd, available under the same attack assumptions as before (Section “Attack Assumptions”).

```
if (passwordTokenToWeaverKey(KDF(
    ↪ passcode_candidate, salt))[0:32] ==
    ↪ eSE_CHALLENGE ) {
  // correct passcode found
}
```

Listing 4 Simplified screen lock brute force pseudo code

We successfully implemented a simple CPU based python version of this off-device screen lock brute force attack, and the results showed that an attacker could recover any four digit pin or 3×3 pattern in less than 1 h on a modest dual-core laptop. This attack could of course be highly optimised on dedicated HW to drastically improve performance.

With the user screen lock credential recovered by this brute force attack, we gain full access to the contents of the mobile device. The credential can be used to authenticate and retrieve FBE keys protecting user data. This fully breaks the confidentiality of the device and the encrypted user data.

Discussion

Our attack shows how recent (and not so recent) research in attack techniques ((Anderson et al., 2006; Bittau et al., 2014; Lee et al., 2017)) can be adapted to new areas, in this case the eSE

HW platform. This improves the probability of success by minimising the necessary knowledge of the target eSE HW and FW. Though the information gathering phase of our attack revealed some unencrypted FW code that could be analysed for security vulnerabilities, this is not a mandatory step. Thus our attack methodology, using information leak oracles from the eSE logical interface, can be applied with no prior knowledge of FW contents.

Our attack demonstrates a complete compromise of the eSE integrity, confidentiality and availability, thus all the main security goals for the eSE CC EAL (SG1-SG3) in Section "CC EAL" are violated. A *single* software security vulnerability is enough, and a single attacker can with limited resources easily discover, and exploit, this vulnerability. Our research required nothing but access to commercially available (COTS) test devices and publicly available information. Vulnerability discovery and exploit development work were done by a single person in approximately one man-month's worth of time, with no special tools required. Our attack does not require physical access to the eSE and can therefore be performed remotely, over-the-air, needing only a privilege escalation vulnerability to be able to communicate via the logical interface of the eSE. This shows that the threat model from Section "eSE Threat Model" does not match this eSE, as physical control is not required to perform our attack.

Restoring the eSE CC EAL (SG1-SG3) security goals (Section "CC EAL") and trust through an eSE FW update seems infeasible, due to the lack of a root-of-trust and secure boot. The eSE can simply not validate its own code as there does not seem to be any on-board cryptographic integrity checks. The only integrity checks performed are simple CRC32 and SHA256 hash comparing. These hashes can be updated by an attacker and thus have no effect on security. In addition, integrity verification of an installed eSE FW cannot be performed by the host system (REE), as the black-box design of the eSE leaves no way to perform external validation of the installed eSE code. A stealthy backdoor implementation by an attacker could be very hard to reveal, making it challenging to detect if the eSE FW has been tampered with. Our discovered vulnerability thus completely breaks any forward trust in the eSE HW. Our results should make users question the validity of this CC EAL 5+ certification.

Furthermore, the exposure of the AES key used for encrypted FW updates of the eSE secure OS and boot images, makes updating this key using normal OTA FW updates difficult, if not impossible, as an attacker with knowledge of this AES key can decrypt any attempt of additional secret sharing with the eSE, such as replacement of the AES key. The effect is that Samsung can no longer exchange confidential information with the eSE HW through FW updates, exposing any encrypted parts of previous and future FW updates (Section "Attack Capabilities and AES Key Exposure"). Samsung is of course free to change the key on newly manufactured devices, but this key cannot also be used in updated firmware for already shipped devices, as that would leak it.

To be able to regain trust in the eSE HW on already shipped devices, the authors believe the only secure option is a physical replacement of the eSE HW, which is probably unreasonable.

Conclusions and future work

We have presented a remote attack on the S3K250AF eSE HW, using our discovered 0-day security vulnerability, exploitable through the logical interface. The attack contributes to the development of new DFA methods of affected devices. The attack was done by building on attacks from other security research areas and applying this to the physically separate eSE HW platform. The eSE HW is designed to withstand high level, and resourceful attackers, relying on a small code base, mostly unavailable to attackers, and resistance to side-channel attacks. Our vulnerability discovery and exploit development required no special tools or access, and the complete attack was developed with very limited resources, far from "state actor" capabilities. The attack enables an attacker to execute arbitrary shellcode to facilitate both reading and writing of both code and data, in both flash and RAM. This completely breaks all the eSE security goals stated in its CC EAL certification, and also enables an attacker to install hard-to-discover, persistent, backdoors and modifications to the eSE FW. Regaining trust in this eSE HW seems challenging, with physical replacement being the only realistically secure option. As this eSE HW is soldered to the PCB in the mobile device, such replacement is not trivial.

We conclude that one simple exploitable buffer overflow vulnerability enables full attacker takeover of the eSE HW, permanently.

Our attack facilitates digital forensic acquisition of devices in a *before-first-unlock* (BFU) state, with no prior knowledge of user credentials. With the aid of a more readily available privilege escalation vulnerability in Android, this becomes a complete solution for digital forensic acquisition of affected devices.

Our attack demonstrates the gap between the *intended* and *achieved* security level of this state-of-the-art eSE HW utilised by a major Android mobile phone vendor. The CC EAL 5+ certification gave no guarantee that the eSE was free from exploitable security vulnerabilities, only that some unidentified amount of effort had been made in an attempt to prevent them. We argue that the trust in the value certifications such as CC EAL provide, needs to be evaluated carefully on a case-by-case basis. Our attack also shows that such certifications should not discourage research into new DFA methods based on offensive techniques.

Our research is not exhaustive, and further attacks, including side-channel attacks, are left for future work. Further research is needed to reveal if other physical separation black-box solutions fall to similar attacks as the ones demonstrated in this research. A new testing methodology for logical interfaces of black-box HW can arise from our work, to potentially improve the CC Advanced methodical vulnerability analysis (AVA_VAN).

We believe our research further emphasises the challenges inherent in trusted computing, and that it demonstrates how fragile the *trust* put in such solutions is, whether this trust comes from certifications like CC EAL, or not.

Responsible disclosure

Samsung is informed of the vulnerabilities discovered in this research and the authors have collaborated with Samsung to mitigate the risks they exposed. A patch for affected devices has been released, assigned with CVE-2020-28341 (MITRE, 2020) and SVE-2020-18632 (Samsung, 2020e). We thank Samsung for their professional cooperation.

Acknowledgements

The research leading to these results has received funding from the Research Council of Norway programme IKTPLUSS, under the R&D project Ars Forensica grant agreement 248094/O70.

Appendix

APDU primer

ISO/IEC 7816 is an international standard for smart-cards. This standard is divided into 15 sub-parts specifying different aspects of smart-card characteristics. ISO 7816−4 (ISO/IEC, 2020) describes security aspects, and commands to communicate with smart-cards. This includes a protocol specification, using what's called an "Application Protocol Data Unit" (APDU). An APDU defines the structure used to send/receive commands, and data. All communication is of the request–reply form, and is always initiated by the host. The smart-card never initiates communication. An APDU consists of a mandatory 4 byte header with the elements: *CLA*, *INS*, *P1* and *P2*, each one byte long. The CLA is referred to as the "class", often tied to the logical handler of the INS: the "instruction" or simply *command*. Inter-industry commands (INS) are defined in CLA `0`. ISO 7816–4 (ISO/IEC, 2020) defines a whole range of standard INS commands, all belonging to the CLA `0`. Vendors are free to implement vendor specific commands using for example CLA `0x80`. P1 and P2 are parameters for use in the specific (CLA,INS) pair and can thus be viewed as normal function parameters to the (CLA,INS) handler on the smart-card.

Additional APDU fields are optional, giving the possibility of appending any necessary data required by the specific (CLA,INS) pair: *Lc*, *DATA* and *Le*. Lc defines the size of appended DATA, and Le is the size of the expected data returned from the smart-card.

The smart-card is expected to reply with a valid return value, *SW* (Status Word). The SW is a 16-bit value consisting of bytes *SW1* and *SW2*, respectively. This reply is mandatory even if the requested (CLA,INS) pair is not implemented by the smart-card. SW is used to give informative error values back to the caller. Although the ISO 7816-4 defines some standardised error codes, these can be vendor defined for all proprietary CLA values. If the (CLA,INS) successfully completes the request, it is expected to return the SW value `0x9000` (SW1 = `0x90`, SW2 = `0x0`).

Table 1
Analysed eSE FW images

Image filename	Image name	Size	Version int / string	SHA256sum
k250a_00000009.img	(whole file)	33280	0 × 47000101 / "191128145540"	638dad7cbf79ede847331516c118ff5b / d27002818ab35356987dcfa498e3262c
k250a_00000009.img	SNVM	33024	0 × 47000101 / "191128145539"	cc8349038e84313a3354459285deade2 bc1aa9ccb6eb3eef9c3a38689d46320b
k250a_00000009_dev.img	(whole file)	198808	0 × 100 / "191120090956"	9914566a795b081fee 2040c1f530fba0 / 3ec1b57521d4dd4b1352a86600ffde5a
k250a_00000009_dev.img	BOOT*	20992	0 × 100 / "191120090947"	3e64599b94e36ed6746a7b15cb48859b a2ec0d419ce954e0b285e104ea711bc6
k250a_00000009_dev.img	CRPT	43928	0 × 100 / "191120090947"	37fcff13acda015611d6d0c3ec8576b4 52642e509b5bd799d83c716356ea8a57
k250a_00000009_dev.img	CORA*	33792	0 × 100 / "191120090947"	622c75c652d637775f92e9b37c03c2fc 0c00494c52d14fa61bd6229d05df4328
k250a_00000009_dev.img	CORB*	33792	0 × 100 / "191120090947"	71967cacde8d30d8f5597eba808e6d97 39025a4cc434c8c137ba42b4d9b4dedc
k250a_00000009_dev.img	SNVM	33024	0 × 100 / "191120090955"	aa82c67c9b545b35f6da05baebfb3549 1402cf27f99e27f81a12a1fdf0f028dc
k250a_00000009_dev.img	IWEA	33024	0 × 100 / "191120090955"	57c0a4e9033d0c9106bb0cde3af6ff82 5a79a4e8efc0de6b445ab7dd14e61c11

* Image encrypted with eSE embedded AES key + IV.

Table 2
eSE attack vectors: Exposed valid eSE APDU CLA and INS

CLA	INS	libese-grdg.so function	Comment
0 × 0	0 × 82	grdg_provisionAK	Provisioning
0 × 0	0xea	grdg_updateFW	FW Update (APP)
0 × 0	0xb1	grdg_getInfo / grdg_updateFW / grdg_updateCrypto	Retrieves various eSE info
0 × 0	0xf1	grdg_selfTest / grdg_IOTest	eSE on-board testing
0 × 0	0xf7	<not exposed>	Unknown
0 × 0	0xf8	<not exposed>	Unknown
0 × 0	0xf9	<not exposed>	Unknown
0 × 0	0xfa	grdg_updateFW	FW Update (CORA/CORB)
0 × 0	0xfb	grdg_updateFW / grdg_updateCrypto	FW Update (CRYPT)
0 × 0	0xfd	<not exposed>	FW Update (BOOT)
0 × 0	0xa4	grdg_snvmInit	APP (SNVM) Init
0 × 80	0xe4	grdg_factoryReset	APP (SNVM) Factory reset
0 × 80	0 × 84	grdg_getRandomNonce	APP (SNVM) Get 32 random bytes
0 × 80	0xb1	grdg_getAppInfo	APP (SNVM) Retrieve APP version info
0 × 80	0xdb	grdg_putCredential / grdg_putPersistentCredential	APP (SNVM) Put credential data
0 × 90	0xdb	grdg_putCredential / grdg_putPersistentCredential	APP (SNVM) Put credential data (large)
0 × 80	0xcb	grdg_getCredential / grdg_getPersistentCredential	APP (SNVM) Get credential data
0 × 80	0xee	grdg_deleteCredential / grdg_deletePersistentCredential	APP (SNVM) Erase credential data
0 × 1	0xa4	grdg_iweaverInit	APP (iWEAVER) Init
0 × 81	0xb1	<not exposed>	APP (iWEAVER) Retrieve APP version info
0 × 81	0 × 35	grdg_getWeaverConfig	APP (iWEAVER) Retrieve configuration
0 × 81	0xdb	grdg_writeWeaver	APP (iWEAVER) write credential slot (secret/challenge)
0 × 81	0xcb	grdg_readWeaver	APP (iWEAVER) read credential (secret) using challenge
0 × 81	0xdf	grdg_updateWeaver	APP (iWEAVER) Update all slots throttle data

References

Ag, I.T., Secure, I., GmbH, N.S.G., STMicroelectronics, 2014. Security ic platform protection profile with augmentation packages. URL: https://www.commoncriteriaportal.org/files/ppfiles/pp0084b_pdf.pdf. (Accessed 26 August 2020).

Alendal, G., Dyrkolbotn, G.O., Axelsson, S., 2018. Forensics acquisition—analysis and circumvention of samsung secure boot enforced common criteria mode. Digit. Invest. 24, S60–S67.

Anderson, R., 2003. Cryptography and competition policy: issues with 'trusted computing'. In: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, pp. 3–10. https://dl.acm.org/doi/abs/10.1145/872035.872036.

Anderson, R., Bond, M., Clulow, J., Skorobogatov, S., 2006. Cryptographic processors-a survey. Proc. IEEE 94, 357–369. https://ieeexplore.ieee.org/document/1580505.

ARM, 2009. Trustzone model. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0301h/Chdebaee.html. (Accessed 12 January 2021).

Beniamini, G., 2016a. Qsee privilege escalation vulnerability and exploit (cve2015-6639). https://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html. (Accessed 1 December 2020).

Beniamini, G., 2016b. War of the worlds - hijacking the linux kernel from qsee. https://bits-please.blogspot.com/2016/05/war-of-worlds-hijacking-linux-kernel.html. (Accessed 1 December 2020).

Beniamini, G., 2017. Trust issues: exploiting trustzone tees. https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploitingtrustzone-tees.html. (Accessed 1 December 2020).

Biondo, A., Conti, M., Davi, L., Frassetto, T., Sadeghi, A.R., 2018. The guard's dilemma: efficient code-reuse attacks against intel {SGX}. In: 27th {USENIX} Security Symposium ({USENIX} Security 18), pp. 1213–1227. https://www.usenix.org/conference/usenixsecurity18/presentation/biondo.

Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D., Boneh, D., 2014. Hacking blind. In: 2014 IEEE Symposium on Security and Privacy. IEEE, pp. 227–242. https://ieeexplore.ieee.org/abstract/document/6956567/.

Bond, M., Anderson, R., 2001. Api-level attacks on embedded systems. Computer 34, 67–75. https://ieeexplore.ieee.org/abstract/document/955101/.

Cerdeira, D., Santos, N., Fonseca, P., Pinto, S., 2020. Sok: understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, Los Alamitos, CA, USA, pp. 1416–1432. https://doi.org/10.1109/SP40000.2020.00061. URL:

Chao, C.Y., Su, H.C., Wu, C.Y., 2020. Breaking samsung's root of trust: exploiting samsung s10 secure boot. Black Hat USA. https://www.blackhat.com/us-20/briefings/schedule/breaking-samsungs-root-of-trust-exploiting-samsung-s-secure-boot-20290.

Chen, Y., Zhang, Y., Wang, Z., Wei, T., 2017. Downgrade attack on trustzone. arXiv:1707.05082. https://arxiv.org/abs/1707.05082.

commoncriteriaportalorg, 2020. Certified products. URL: https://www.commoncriteriaportal.org/products/. (Accessed 26 August 2020).

Criteria, C., 2017. Common criteria for information technology security evaluation - part 3: security assurance components. URL: https://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R5.pdf. (Accessed 26 August 2020).

Criteria, C., 2020. Publications. URL: https://www.commoncriteriaportal.org/cc/. (Accessed 1 September 2020).

de la Sécurité des Systèmes d'Information (ANSSI), A.N., 2019. Rapport de certification anssi-cc-2019/61. URL: https://www.commoncriteriaportal.org/files/epfiles/anssi-cc-2019_61fr.pdf. (Accessed 26 August 2020).

Focus, F., 2016. Current challenges in digital forensics. URL: https://articles.forensicfocus.com/2016/05/11/current-challenges-in-digital-forensics/. (Accessed 30 November 2020).

Fournaris, A.P., Keramidas, G., 2014. From Hardware Security Tokens to Trusted Computing and Trusted Systems. Springer International Publishing, Cham, pp. 99–117. https://doi.org/10.1007/978-3-319-00663-5_6. URL:

für Wirtschaft und Energie, B., 2020. Optimos 2.0. URL: https://www.digitale-technologien.de/DT/Redaktion/DE/Standardartikel/SmartServiceWeltProjekte/Wohnen_Leben/SSWII_Projekt_OPTIMOS_20.html. (Accessed 14 September 2020).

Gkaniatsou, A., McNeill, F., Bundy, A., Steel, G., Focardi, R., Bozzato, C., 2015. Getting to know your card: reverse-engineering the smart-card application protocol data unit. In: Proceedings of the 31st Annual Computer Security Applications Conference, pp. 441–450. https://dl.acm.org/doi/pdf/10.1145/2818000.2818020.

Google, 2020a. Android code search. URL: https://cs.android.com/android/platform/superproject/+/master:frameworks/base/services/core/java/com/android/server/locksettings/SyntheticPasswordManager.java. (Accessed 21 September 2020).

Google, 2020b. Android security bulletins. URL: https://source.android.com/security/bulletin. (Accessed 6 October 2020).

Google, 2020c. File-based encryption. URL: https://source.android.com/security/encryption/file-based. (Accessed 21 September 2020).

Google, 2020d. Google git. URL: https://android.googlesource.com/platform/hardware/interfaces/+/refs/heads/master/weaver/1.0/IWeaver.hal. (Accessed 21 September 2020).

Group, T.C., 2020. Tpm 1.2 main specification. URL: https://trustedcomputinggroup.org/resource/tpm-main-specification/. (Accessed 13 October 2020).

Gunasekera, S., 2020. Rooting Your Android Device. Apress, Berkeley, CA, pp. 173–223. URL: https://doi.org/10.1007/978-1-4842-1682-8_8.

Hatton, L., 1997. Reexamining the fault density-component size connection. IEEE Softw 14, 89–97. https://doi.org/10.1109/52.582978. URL: https://ieeexplore.ieee.org/abstract/document/582978/.

Intel, 2020. Intel software guard extensions. URL: https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html. (Accessed 6 October 2020).

ISO/IEC, 2020. Iso/iec 7816-4:2020 identification cards – integrated circuit cards – part 4: organization, security and commands for interchange. URL: https://www.iso.org/standard/77180.html. (Accessed 26 August 2020).

Jang, Y., Lee, J., Lee, S., Kim, T., 2017. Sgx-bomb: locking down the processor via rowhammer attack. In: Proceedings of the 2nd Workshop on System Software for Trusted Execution, pp. 1–6. https://dl.acm.org/doi/abs/10.1145/3152701.3152709.

Lee, J., Jang, J., Jang, Y., Kwak, N., Choi, Y., Choi, C., Kim, T., Peinado, M., Kang, B.B., 2017. Hacking in darkness: return-oriented programming against secure enclaves. In: 26th {USENIX} Security Symposium ({USENIX} Security 17), pp. 523–539. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk.

Lillis, D., Becker, B., O'Sullivan, T., Scanlon, M., 2016. Current challenges and future research areas for digital forensic investigation. In: The 11th ADFSL Conference on Digital Forensics, Security and Law (CDFSL 2016).

Limited, A., 2020a. Arm compiler toolchain linker reference. URL: https://developer.arm.com/documentation/dui0493/g/linker-command-line-options/–be8. (Accessed 26 August 2020).

Limited, A., 2020b. Securcore sc000. URL: https://developer.arm.com/ip-products/processors/securcore/sc000-processor. (Accessed 26 August 2020).

Limited, A., 2020c. The thumb instruction set. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/CACBCAAE.html. (Accessed 26 August 2020).

Mavrovouniotis, S., Ganley, M., 2014. Hardware Security Modules. Springer New York, New York, NY, pp. 383–405. URL: https://doi.org/10.1007/978-1-4614-7915-4_17.

Mayrhofer, R., Stoep, J.V., Brubaker, C., Kralevich, N., 2019. The android platform security model. arXiv preprint arXiv:1904.05572. https://arxiv.org/abs/1904.05572. (Accessed 4 September 2020).

MITRE, 2020. CVE-2020-28341. Available from MITRE, CVE-ID CVE-2020-28341. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-28341.

Moghimi, D., Sunar, B., Eisenbarth, T., Heninger, N., 2020. Tpm-fail:{TPM} meets timing and lattice attacks. In: 29th {USENIX} Security Symposium ({USENIX} Security 20), pp. 2057–2073. https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-tpm.

Nemec, M., Sys, M., Svenda, P., Klinec, D., Matyas, V., 2017. The return of coppersmith's attack: practical factorization of widely used rsa moduli. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1631–1648. https://dl.acm.org/doi/abs/10.1145/3133956.3133969.

Nilsson, A., Bideh, P.N., Brorsson, J., 2020. A survey of published attacks on intel SGX. Technical report. Tech. rep. https://arxiv.org/abs/2006.13598.

NIST, 2020. Cryptographic algorithm validation program. URL: https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/details?product=11431. (Accessed 26 August 2020).

One, A., 1996. Smashing the stack for fun and profit. Phrack 7. http://www.phrack.com/issues.html?issue=49&id=14.

Ozment, A., Schechter, S.E., 2006. Milk or wine: does software security improve with age?. In: Proceedings of the 15th Conference on USENIX Security Symposium, vol. 15. USENIX Association, USA. https://www.usenix.org/legacy/events/sec06/tech/full_papers/ozment/ozment.pdf.

Pfleeger, C.P., 2009. Security in Computing. Pearson Education India.

Sabt, M., Achemlal, M., Bouabdallah, A., 2015. Trusted execution environment: what it is, and what it is not. In: 2015 IEEE Trustcom/BigDataSE/ISPA, pp. 57–64. https://ieeexplore.ieee.org/abstract/document/7345265. (Accessed 13 October 2020).

Samsung, 2020a. S3k250a/s3k232a/s3k212a 32-bit risc microcontroller for smart card with optional at1 secure libraries including specific ic dedicated software. URL: https://www.commoncriteriaportal.org/files/epfiles/anssi-cible-cc-2019_61en.pdf. (Accessed 26 August 2020).

Samsung, 2020b. Samsung, bsi, bundesdruckerei and telekom security partner to bring national id to your smartphone. URL: https://www.samsungmobilepress.com/pressreleases/samsung-bsi-bundesdruckerei-and-t-systems-partner-to-bring-national-id-to-your-smartphone. [Online; accessed 14-September-2020].

Samsung, 2020c. Samsung ese sdk. URL: https://developer.samsung.com/ese/overview.html. (Accessed 26 August 2020).

Samsung, 2020d. Samsung introduces best-in-class data security chip solution for mobile devices. URL: https://news.samsung.com/global/samsung-introduces-best-in-class-data-security-chip-solution-for-mobile-devices [Online; accessed 26-August-2020].

Samsung, 2020e. SVE-2020-18632. November 2020, SVE-2020-18632. URL: https://security.samsungmobile.com/securityUpdate.smsb.

Schwarz, M., Weiser, S., Gruss, D., 2019. Practical enclave malware with intel sgx. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 177–196. https://link.springer.com/chapter/10.1007/978-3-030-22038-9_9.

Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R., 2018. Foreshadow: extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In: 27th {USENIX} Security Symposium ({USENIX} Security 18), pp. 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

Vauclair, M., 2011. Secure Element. Springer US, Boston, MA, pp. 1115–1116. https://doi.org/10.1007/978-1-4419-5906-5_303. URL: (Accessed 1 September 2020).

Venturebeat, 2016. Apple vs. fbi: a timeline of the iphone encryption case. URL: http://venturebeat.com/2016/02/19/apple-fbi-timeline/. (Accessed 30 November 2020).

Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T.F., Yarom, Y., 2018. Foreshadow-ng: breaking the virtual memory abstraction with transient out-of-order execution. https://lirias.kuleuven.be/2089352?limo=0.