Leveraging ARM TrustZone and Verifiable Computing to Provide Auditable Mobile Functions

Nuno O. Duarte
INESC-ID / IST, University of Lisbon
nuno.duarte@tecnico.ulisboa.pt

Sileshi Demesie Yalew
KTH Royal Institute of Technology
sdyalew@kth.se

Nuno Santos
INESC-ID / IST, University of Lisbon
nuno.santos@inesc-id.pt

Miguel Correia
INESC-ID / IST, University of Lisbon
miguel.p.correia@tecnico.ulisboa.pt

ABSTRACT

The increase of personal data on mobile devices has been followed by legislation that forces service providers to process and maintain users' data under strict data protection policies. In this paper, we propose a new primitive for mobile applications called *auditable mobile function* (AMF) to help service providers enforcing such policies by enabling them to process sensitive data within users' devices and collecting proofs of function execution integrity. We present SafeChecker, a computation verification system that provides mobile application support for AMFs, and evaluate the practicality of different usage scenario AMFs on TrustZone-enabled hardware.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**;

KEYWORDS

Verifiable computing, Trusted computing, Data protection regulations, Auditable mobile functions

ACM Reference Format:
Nuno O. Duarte, Sileshi Demesie Yalew, Nuno Santos, and Miguel Correia. 2018. Leveraging ARM TrustZone and Verifiable Computing to Provide Auditable Mobile Functions. In *EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous '18), November 5–7, 2018, New York, NY, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3286978.3287015

1 INTRODUCTION

For over 30 years [5], an important problem that has fascinated the research community is about trusting the result of computations performed on untrusted platforms. Since the advent of cloud computing, this interest has further grown in obtaining proofs of code execution on untrusted clouds.

An approach to address this problem that has garnered particular attention is *verifiable computing*. At a high-level, verifiable computing consists of studying how a prover can convince a verifier of a mathematical assertion. The main obstacle faced by this field has been practicality, as the costs of computing and verifying proofs over general computations tend to be very high [21]. Still, recent adaptations of existing theoretical protocols into real systems [20] yielded performance improvements of 20 orders of magnitude [21], moving the field into a near-practical state. In particular, the *succinct non-interactive arguments of knowledge* (SNARK) cryptographic primitive [8] provides short proofs of computation and figures among the biggest advances in the field with respect to performance efficiency.

In this paper, we argue that verifiable computing has a good application potential for mobile platforms and propose the concept of *auditable mobile functions* (AMFs). Essentially, AMFs consist of a mobile OS primitive that allows a remote verifier to obtain integrity proofs of function execution, serving as an additional mobile data protection mechanism. In fact, while we have witnessed in recent years an increase in personal data collection on mobile devices, service providers have undergone a growing pressure with respect to how that data must be handled, driven not only by the end-users' manifested privacy concerns, but also by legislation that imposes strict data protection policies like the GDPR [1].

The AMF primitive allows service providers to build mobile applications capable of *processing users' data on their devices* while obtaining *proofs of integrity of the computations* therein performed. By avoiding transferring security-sensitive data off users' devices and into their servers, service providers are able to lessen the risks of potential security breaches that might fall under their responsibility.

We present the design, implementation, and evaluation of SafeChecker, a system that provides AMF support for Android applications. A key novelty of our system's design is the incorporation of techniques from *verifiable computing* and *trusted computing*. First, SafeChecker uses SNARKs to generate proof of execution of AMFs. Second, it verifies the resulting proofs on the mobile device within a TrustZone-enabled trusted execution environment (TEE) isolated from the operating system; ARM TrustZone consists of a set of hardware security extensions featured in modern ARM processors [19]. SafeChecker keeps a log of the proof validations inside the TEE and provides the means for external services to check these results and notify the verifier. By leveraging these techniques, SafeChecker is able to provide proofs of AMF integrity without the need to trust the mobile OS. We focus on functions written in native code that can be bundled with Android apps, without loss of generality.

In building SafeChecker, it was necessary to provide for the security of the entire AMF life cycle, especially in critical steps involving application code transformation and cryptographic key provisioning. We modified the Android OS by implementing all the AMF logic necessary to support real-world Android applications and enable efficient world switch between OS and trusted environment, and we adapted existing SNARK algorithms to run inside the TEE. We built a fully operational SafeChecker implementation and tested it on TrustZone-enabled hardware for three example use cases. Our results show that the execution times of the tested AMFs sit in the order of seconds, which are acceptable numbers considering the use cases under study in exchange for the additional benefits brought about by AMFs.

Next, we provide an overview and usage scenarios for AMFs. In Section 3, we introduce our building blocks, and present the design of SafeChecker in Section 4. We proceed by presenting implementation details on our prototype in Section 5 and complement them with evaluation results and performance and security discussion in Section 6. Finally, we discuss related work in Section 7, and conclude in Section 8.

2 OVERVIEW AND SCENARIOS

In this section, we motivate the need for our new primitive, clarify the main design goals, and provide example use cases.

2.1 Auditable Mobile Functions

We propose *auditable mobile function* (AMF), a new primitive which enables a *verifier* party to obtain a proof over the correct execution of a security-sensitive function on a remote mobile device. By correct execution of a function *f*, we mean that the integrity and authenticity of *f*'s inputs, outputs, and code can be ascertained with high probability by the verifier.

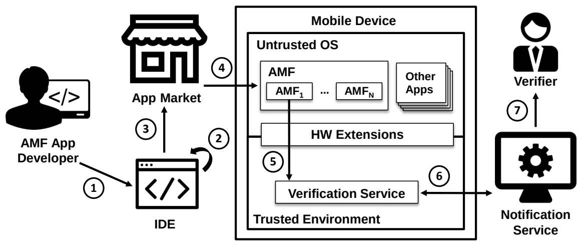


Figure 1: Lifecycle of an application with AMFs.

For a number of mobile application scenarios, service providers may need to process privacy-sensitive data collected by the end-users' devices. By sending such data to their servers, service providers must ensure that this data is correctly maintained. However, given the stringent data handling policies determined by law in many countries (notably in the EU [1]), data breaches from servers may result in the payment of huge fines. To reduce this risk, sensitive data may be processed locally on the mobile devices by letting service providers execute specific functions over the data and collect the resulting outputs. In this way, the original raw data is maintained always in the mobile devices. Depending on the usage scenario, service providers may also require function execution correctness guarantees from mobile devices. Providing such guarantees is the goal of AMFs.

Figure 1 presents a high-level view of the lifecycle of an application that implements AMFs. This lifecycle is primarily controlled by the service provider, which is responsible for implementing the application (step 1), indicating which functions must be considered as AMFs (step 2), and releasing the resulting application package to application markets, e.g., Google Play (step 3). Users then install the application on their devices as usual (step 4), and whenever an AMF is executed an accompanying process for proof generation and verification takes place (step 5). Later, by playing the verifier role, the service provider can retrieve the AMF verification outcome through some mediation system (step 6) and check whether or not the AMF has been correctly executed (step 7). This operation model can bring two important benefits for service providers. First, a service provider can prove toward a third party (e.g., a legal authority) that its application does effectively retain the users' data on their devices, thereby preserving privacy. Such proofs can be obtained by allowing the third party to inspect the application's source code and validate the correct usage of AMFs. Second, the service provider can obtain guarantees that the integrity of the AMF executed at the mobile endpoint has not been tampered with, e.g., as a result of malware interference.

2.2 Usage Scenarios for AMFs

To illustrate the applicability of AMFs, we provide three examples of usage scenarios. In all these cases, AMFs aim to enable a service provider to take the role of verifier party in order to obtain guarantees of AMF execution on remote mobile devices. Such function is part of a mobile application that users install on their devices. The concrete parties involved and their respective incentives for adopting AMF-based mobile applications depend on the specific usage scenario.

1. Digital evidence collection. Oftentimes, witnesses or victims of criminal activity use their own smartphones for taking pictures, recording videos, or capturing audio which can later be used by police authorities for forensic analysis. To preserve admissibility, it is crucial to guarantee the authenticity of the media content collected by mobile devices' sensors. Since the collected data may include sensitive information for the witnesses themselves (e.g., if the witness is a whistle-blower or a victim), it should be possible to sanitize the collected sensor data (e.g., blurring sensitive regions of a photo) before extracting it from the device while ensuring the authenticity of the obtained results. While previous work introduced the notion of trusted sensors [17] to ensure the authenticity of collected sensor data, AMFs allow police authorities to take one step further by verifying that the intended sanitization function has been correctly applied. On the other hand, by keeping the raw sensor data on mobile devices, the application is privacy friendly.

2. Gamification. Some applications reward users based on given behaviour adoption, e.g., cycling, together with certain actions such as covering a certain distance or visiting certain points of interest, e.g., shops. Given that location data constitutes privacy-sensitive information, it is in the gamification service's best interests to process this data in users' devices. With AMFs, gamification services can achieve such a goal while checking that the remotely executed functions have not been tampered with, e.g., by malware or some dishonest user. For instance, in the aforementioned cycling application, determining if the device has visited a given set of points of interest could be implemented by an AMF. The service provider would still be able to determine with high confidence that the user has taken some expected route and therefore can reliably grant him a reward. At the same time, this property would be attained without the gamification service needing to know about users' precise locations.

3. E-health. A class of e-health applications aims to provide health monitoring services. For instance, certain applications combine wearables, e.g., smart bracelets, with users' mobile devices to monitor user health indexes, e.g., heart rate, and recommend physical activity or even medication dosages. Although systems like DroidVault [15] can provide integrity protection of health records at rest, AMFs can be used as a complementary security mechanism that enables e-health service providers to check the integrity of the data processing operations on end-users' devices. By using AMFs, a service provider could, e.g., prescribe reliable medication dosages to its users or notify them about abnormal sensor readings without the need to extract the raw data from their devices.

2.3 Goals and Threat Model

The goal of our work is then to build a system to support the lifecycle of AMFs on commodity mobile devices. Our system is meant to be used by the service providers for incorporating AMFs onto their mobile applications. Developers of AMF-based mobile applications are honest. They write the mobile applications' code, in particular AMF functions, so that it satisfies the requirements laid out by the service provider (which also plays the verifier role). As a result, application developers are trusted with respect to annotating their apps, writing the code of the respective AMFs, and correctly compiling, packaging, and publishing applications.

The main security goal of our system is to preserve the *integrity* of AMFs when they are executed on the end-users' devices and generate the respective proofs. An adversary may try to interfere with the execution of AMFs to its own advantage (e.g., by changing AMF code) in a way that the resulting deviant behavior is not detected by the verification service. We model the adversary as any agent with the capability to control any application or the OS of the mobile device. In practice, the specific agent will depend on the usage scenario. In some cases, malware might subvert mobile applications or the entire OS. In other cases, the mobile user might obtain some benefit by manipulating the AMFs' execution on a rooted device, for example. Note that it is not in scope to provide confidentiality protection for the data residing on the devices (it may be a by-product of not requiring data to leave the mobile device for processing, though).

As for the building blocks of our solution, we intend to adopt verifiable computing techniques and ARM TrustZone, and we target the Android mobile platform. We assume the hardware platform is correct, and the OS is untrusted, i.e., is not part of the trusted computing base. Without loss of generality, AMFs are implemented as native functions included in the code of mobile applications.

3 BUILDING BLOCKS

This section introduces the two building blocks of our solution: verifiable computing and ARM TrustZone.

3.1 Verifiable Computing

The field of verifiable computing addresses one of the biggest issues with cloud computing, i.e., the need to trust remote machines to process privacy-sensitive data. Essentially, it

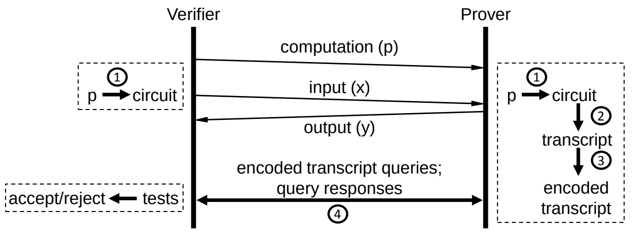


Figure 2: Verifiable computing framework [25].

allows for untrusted third parties to provide clients with computation execution correctness proofs.

Figure 2 presents the general flow associated with verifiable computing [25]. Essentially it shows a framework in which a client, acting as verifier, can assess whether, for a computation p and desired input x, the output y given by a remote party, acting as prover, over $p(x)$ is correct. Initially, both verifier and prover compile p, usually a function expressed in a high level language, e.g., C, into a boolean circuit, C (step 1). These circuits are simply networks of AND, OR, and NOT logic gates. Then, the prover executes p, obtaining a transcript for the execution of C on x (step 2). In step 3, the prover encodes the transcript in a way suitable for efficient querying by the verifier. In possession of the transcript, the verifier can then issue probabilistic queries in order to assess computation execution correctness (step 4).

Although the field has had practicality issues [21], there have been considerable performance improvements [21, 22] leading to the emergence of open source projects [2, 3] that foster the usage of verification cryptographic protocols. A particular cryptographic primitive called SNARK [8] is at the very root of these improvements. First, by leveraging arithmetic constrains, i.e., sets of equations over finite fields, instead of boolean circuits to represent computations, it significantly improves performance. And second, by providing non-interactive and zero-knowledge properties to execution proofs it further expands the range of use cases supported.

3.2 ARM TrustZone Technology

Our second building block consists of ARM TrustZone, which is a set of security extensions of ARM processors providing hardware level isolation between two execution domains: *normal world* and *secure world* [19]. The normal world is assumed to run untrusted software, usually a large OS, e.g., Android, and its applications. The secure world hosts the TEE and is supposed to run a smaller, trustworthy kernel, and security services. The context switch between worlds is controlled by the higher privileged monitor mode.

Software in the normal world can force a switch to the secure world by calling the secure monitor call (SMC) instruction. Compared to the normal world, the secure world has higher privileges, as it can access the normal world's memory, CPU registers and peripherals but not vice-versa. More specifically, each world has access to its own memory management unit (MMU) to maintain separated page translation tables. Cache memories are also TrustZone-aware, i.e., cache lines are tagged as secure and non-secure hence preventing normal world access to secure cached content. This mechanism is complemented by the ability to exclusively assign interrupts to the secure world, therefore guaranteeing peripheral world access isolation.

3.3 Combining Verifiable Computing with ARM TrustZone

In our solution, we use TrustZone to provide protection for the proof verification stage implemented by the verifier. In fact, since the proof verification requires access to the inputs and given that, as explained in Section 2.1, the inputs must never leave the end-users' mobile devices, this means that the verification process must be carried out on the mobile devices themselves. On the other hand, since the OS is not trusted, we use TrustZone to provide the necessary isolation between the OS and a trusted environment where the verification logic can execute securely; the trusted environment will be implemented inside the secure world.

An alternative design would to be to run AMFs entirely inside the secure world thereby precluding the need for verifiable computing protocols altogether. However, this approach has an important downside which we aim to avoid, namely that it requires running untrusted application code inside the secure world. Although there are TEE systems that allow for sandboxing such code [19], recent attacks to TEE systems have shown that allowing for the execution of general-purpose untrusted applications inside the secure world increases the attack surface and hence the risks of security breaches [18]. To reduce such risks, we explore a design point in which only the verification logic runs inside the secure world, therefore reducing exposure to attacks.

4 SAFECHECKER DESIGN

We present the design of SafeChecker, a system that provides AMF support for the Android mobile platform.

4.1 Architecture

Figure 3 depicts the architecture of our system, namely its components as well as most of the execution flows of the verification notification environment. For practical reasons we take the Android OS as the underlying example. Shaded boxes represent SafeChecker's original components, while pattern-filled boxes represent existing components needed to be modified. All trusted components of SafeChecker reside in the secure world, namely SafeChecker Kernel, Verification Engine and Storage Driver. The normal world runs an

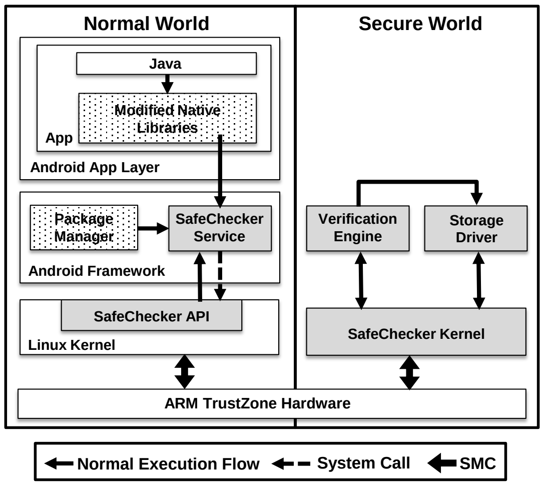


Figure 3: SafeChecker architecture.

Android stack together with a SafeChecker OS service that provides an interface to mobile apps and leverages a system call API to perform world switches via the `SMC` instruction.

All these components play specific roles in the lifecycle of AMFs. Given that auditable functions are embedded into verifiable applications' packages (Section 4.2), the OS's application installer (i.e., package manager in the figure) must be modified to handle them accordingly when installing applications on the system (Section 4.3). The execution of computation proofs happens whenever an AMF is called, which effectively triggers the execution of the corresponding function in the modified native libraries (Section 4.4).

After the proof is calculated, it is forwarded to the verification engine, leveraging SafeChecker's service and API in the normal world, and the SafeChecker kernel in the secure world (Section 4.5). Once the verification finishes, its results are stored in the trusted filesystem managed exclusively by the Storage Driver, thus preventing access from the normal world. The control is then returned to the normal world.

With respect to verifier notification, authorized notification services are able to extract proof verification results from the secure filesystem in order to notify verifiers using channels other than the ones accessible through the potentially compromised mobile device, e.g., dedicated email account (Section 4.6). In the following sections, we provide details over the different SafeChecker operations, and base our descriptions on the example given in Figure 4.

4.2 Verifiable Application Code Adaptation

Android applications are typically written in Java, but they can feature native code (C/C++ code compiled into binary code), leveraging the Java native interface (JNI). In order to implement AMFs, application developers must (1) add the `@Auditable` annotation to the intended Java native methods they wish to be verified at runtime, and (2) configure their build process to accommodate the proving logic adaptation. Note that the programming effort associated with providing AMF support for typical applications is small: developers need simply to declare which functions should be made verifiable and add the corresponding annotation to each of them.

Once a developer compiles an annotated application, there must be a proof adaptation procedure in the build process to make the application compatible with the verification mechanism. This procedure is implemented by an adaptation pipeline mechanism composed of the following four stages:

1. Method matching: Figure 4 provides an example of the complete verification environment for a certain function *f*. The code adaptation poses very little disruption, as developers can simply use the IDE to annotate the Java source code, and work on the native source code of function *f*, and other typical resources of standard mobile applications. After most of the usual build process is completed, the adaptation pipeline begins. In the first stage of the pipeline, the annotated Java methods are parsed so as to determine which native functions implement the high level methods' corresponding logic.

2. Constraint generation: After this matching, the AMFs' source code is processed and adapted to our mechanism. Instead of compiling these functions into boolean circuits (see Section 3), we use a more efficient approach in which functions are compiled into arithmetic constrains, i.e., sets of equations over finite fields [21]. Therefore, in this stage, these functions' source code is processed by an arithmetic constraint generator. This module analyzes the code, and transforms each function into a group of algebraic representations called *quadratic arithmetic programs* (QAPs). The resulting artifacts consist of a group of QAP files needed to both prove and verify computation proofs, as well as a file containing a representation of the operations a prover must follow to compute a proof over a function *f*. In Figure 4 the group of QAPs for a given function f is represented by C_f.

3. Prover compilation: Next, the prover logic is compiled. First it compiles the general prover that proves nondeterministic polynomial time (NP) statements as constraint systems represented by QAPs. Then each of the AMFs' source code is bundled with a call to the generic prover patched with the function's corresponding QAPs and prover configuration file. At that point, a native library containing those adapted functions and general prover is compiled. Ultimately, this process generates for a given function f an execution verification compatible function f' (see Figure 4).

4. APK assembly: In the last stage, the standard application package assemblage must be adapted to include the resulting prover native library, as well as the QAP files needed by the verification component. Note that, for every processed

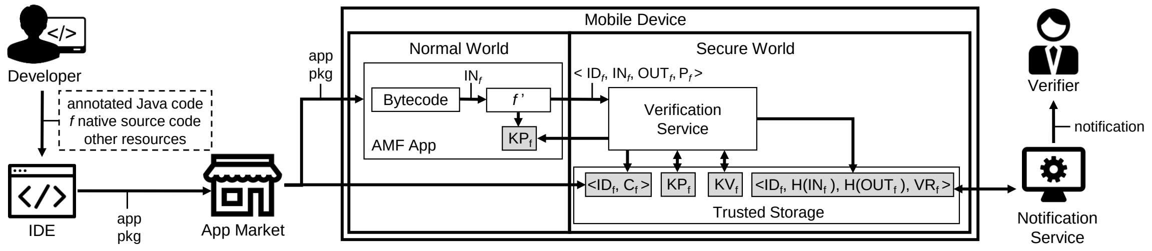


Figure 4: Auditable mobile function metadata.

auditable function f, the IDE must assign a unique id, ID_f (see Figure 4), and sign these QAP files.

4.3 Verifiable Application Installation

Since the retrofitting procedure produces additional verification artifacts, there is a need to adapt the mobile OS's application installation logic. Upon installing an application, Android normally performs operations such as creating directories to store the application's shared preferences; assess permissions by analyzing the application's manifest file. With our approach, the installation of applications also requires the installation of the QAP files, needed by the verification engine, in the secure world's filesystem. In order to do so, the application installer must forward these artifacts to the SafeChecker service while installing an application package. At this point, the service leverages the SafeChecker's system call API to allocate and include the artifacts in a memory region shared with the secure world, which the SafeChecker kernel accesses after an application installation flagged `SMC` instruction is called.

In possession of these artifacts, the SafeCheker kernel first verifies these QAP files' authenticity by checking their signatures. Then it leverages the storage driver to store these artifacts securely. Note that for a given function f these artifacts are stored as a tuple $\langle ID_f, C_f \rangle$, where ID_f is the id assigned by the IDE, and C_f is a representation of these QAP files (Figure 4). Once these artifacts are safely stored, the SafeChecker kernel issues the verification engine to generate new verification and proving keys for each of the new functions, which it again persistently stores through the storage driver. Although the verification key (KV_f) never leaves the secure world, the proving key (KP_f) must be provided to applications in order for AMFs to compute execution proofs.

4.4 Proof Computation

Because of the modified native libraries, once an application calls an AMF, it automatically issues the computation of a proof over the corresponding native function. But before actually computing the proof, the native library must first get the computation's proving key (KP_f).

In case of a first time verification, the library must first fetch the function's proving key from the secure world. Only after issuing a SafeChecker service call, which triggers a system call to the SafeChecker API and corresponding SMC to the SafeChecker kernel, is the proving key returned from secure storage. In subsequent verifications, the library may fetch the proving key from a local directory. Once the execution proof is calculated (P_f), the library again issues a SafeChecker service call, thus delivering the proof through a chain that ends in the verification engine, where the verification takes place. Apart from KP_f, AMFs must also be fed the normal inputs expected to run a native function (IN_f).

4.5 Proof Verification

To verify a proof, the verification engine first fetches the verification key (KV_f) and QAPs corresponding to a function f from the secure filesystem, and then asserts the proof leveraging them. More specifically, to verify a proof, an AMF must provide the id of the function (ID_f), the inputs and outputs from the function (IN_f, and OUT_f respectively), and the execution proof (P_f). In this case, the verification logic from the verification engine is as general as the proving logic bundled into applications' native libraries.

A limitation of our approach is the verification time. Trusted hardware solutions such as ARM TrustZone rely on isolation mechanisms that prevent execution concurrency between the normal and secure worlds. In practice this means that at any given time, only one of the worlds is executing. Therefore, although applications can leverage multithreading to compute several proofs simultaneously, verifying an execution proof in the secure world leads the normal world to halt. In a mobile device scenario this means the user interface will not be responsive until the verification step finishes and control is taken back to the normal world.

A logical workaround to this usability issue is to delay the verification of proofs until the device is on a stand-by mode, i.e., its screen is off. This means that at runtime, the switches between the two worlds concern only the secure storage of proofs, and request for first-time proving keys. Then, when the device is on stand-by, the SafeChecker service can trigger the world switch and the verification engine can process a batch of proofs. Once a verification terminates, its result is securely stored as a tuple $\langle ID_f, H(IN_f), H(OUT_f), VR_f\rangle$, where VR_f is the verification result, and $H(IN_f)$ and $H(OUT_f)$ are hashes of the inputs and outputs fed and outputted by function f respectively (Figure 4). These hashes are essential to keep sensitive information private when divulging verification information to notification services. Note however that users should be allowed to specify the information accessible by these services, so the use of these hashes should be parameterized.

4.6 Result Retrieval

To keep verifiers informed of applications' execution correctness, authorized notification services are allowed to periodically query the SafeChecker's kernel for proof verification results. In order to do so, SafeChecker's service must provide a query API usable only by authorized notification services. When the proof result query API call is issued by a notification service, SafeChecker's service must first authenticate that service, and then leverage SafeChecker's API to retrieve the proof verification results from the secure world. Whenever a query is made, the SafeChecker kernel must fetch these results from the secure filesystem, sign and copy them to a memory region shared with the normal world. After the world switch through the `SMC` instruction, the service can then forward the results to the notification service. Because the results are signed, even if SafeChecker's service is compromised, the contents of the results cannot be tampered with. Furthermore, we assume the existence of a secure channel between the notification service and SafeChecker's kernel, since we assume both the notification service and the mobile device have uniquely identifying keypairs. This is particularly admissible for ARM-based mobile devices, since they are usually shipped with such keys.

5 SAFECHECKER IMPLEMENTATION

We implemented a SafeChecker prototype for the Freescale NXP i.MX53 Quick Start Board. Our prototype features needed components to assess the costs associated to verifiable computing in mobile devices. TCB comprises the SafeChecker Kernel, the Verification Engine, and the Storage Driver.

The SafeChecker kernel resides in the secure world and is responsible for memory management, thread execution, and world switch operations. To facilitate the development of SafeChecker's kernel, we adopted the *base-hw* custom kernel from the Genode Framework [4], which comprises a 20 KLOC codebase. Additionally, we used Genode's Virtual Machine Monitor (VMM) to run a paravirtualized Android OS, which leverages a custom-made Linux kernel for Genode. This paravirtualization is needed so that certain resources (e.g., framebuffer) and signals (e.g., data abort interrupt) can be managed only by the secure world.

In order to build our Verification Engine, we leveraged libsnark [2], a library that offers several cryptographic methods for proving and verifying the integrity of computations. To preserve a small TCB, we ported only the logic associated with the *r1cs_ppzksnark* proof system, which proves NP statements as rank-1 constraint systems (R1CS), which internally are converted into QAPs. Additionally, because of compatibility issues, we ported the logic associated with the *alt_bn128* elliptic curve, an alternative to the more optimized *bn_128* curve, which also provides 128 bits of security but is only available for x86 architectures. Because libsnark uses very large integers when handling the integrity of computations, we also had to port GNU's Multiple Precision Arithmetic Library (GMP). The process of shrinking our GMP port to achieve a smaller TCB is left for future work.

With respect to the arithmetic constraint generator, we leverage Pequin's [3] frontend, which produces the QAP and prover configuration files necessary for our verification logic to be tested. We created an Android system service representing the SafeChecker service, and a kernel module that offers a verification API that triggers the necessary world switches through `SMC` invocations, and passes the necessary artifacts between worlds (e.g., proving keys). Currently we hold a few computations adapted with the verification mechanism in the SafeChecker service and have manually inserted their corresponding QAP files in the secure storage.

6 EVALUATION

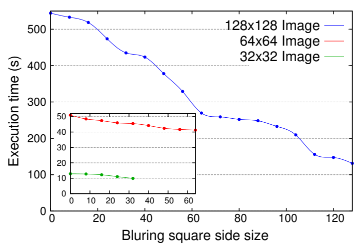
In this section, we evaluate the performance of our prototype when verifying computation proofs in the secure world.

6.1 Methodology

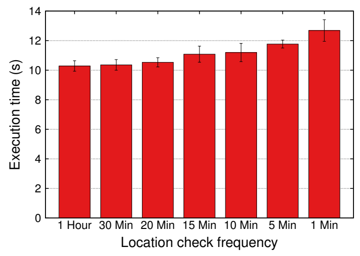
To illustrate how both the proof computation and verification performances are influenced by the computation complexity, we implemented three AMF-based applications which aim to mimic the use cases described in Section II-B:

Digital evidence collection: The first application implements an AMF for censoring parts of a picture. The AMF receives a squared `int32_t` matrix representing the image's original pixels, and outputs a copy with a censored square of pixels starting from the upper left corner. The pixels of the censored square are colored with a constant color value. For baselines we use a 32x32, 64x64 and 128x128 matrices, and increase the square size from 1x1 until the size of the whole picture.

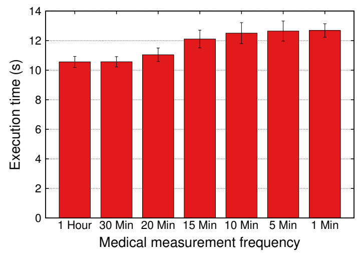
Gamification: The second application aims to test whether a user is located within a certain geographical region. It implements an AMF which receives as input two `double` vectors representing latitude and longitude GPS coordinate pairs, and eight doubles representing the GPS coordinates of a squared region. The AMF then returns the number of times a pair of coordinates falls within that squared region.



(a) **Image censorship proof.**

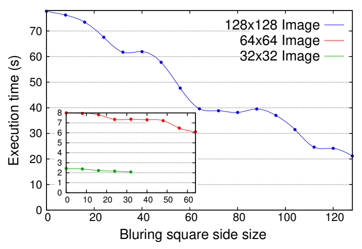


(b) **Location check proof.**

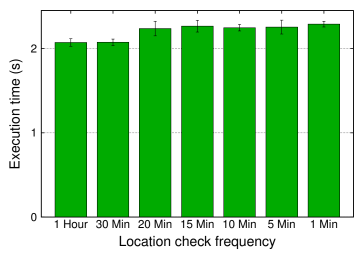


(c) **Measurement average proof.**

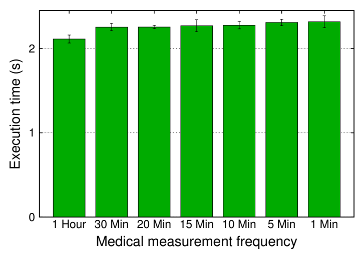
Figure 5: Proof computation performance.



(a) **Image censorship verification.**



(b) **Location check verification.**



(c) **Measurement average verification.**

Figure 6: Proof verification performance.

To measure the performance, we begin with a GPS data collecting period of one hour, and vary the frequency of collected data pairs from one per hour to one per minute.

E-health: The third application implements an AMF that computes the daily average heart rate of a user. The AMF receives a `int16_t` vector containing heart rate readings, and returns the average of all the values therein. For performance measurement, we start with a daily calculation, and vary the reading frequency from one per hour to one per minute.

We test the performance of these three applications on an i.MX53 Quick Start Board, featuring a 1 GHz ARM Cortex-A8 Processor, and 1 GB of DDR3 RAM memory. The board launched SafeChecker from a mini SD card, which was flashed with the modified Genode and Android versions. We report the mean of 5 runs, and standard deviations.

6.2 Verification Performance

We present our evaluation results separately regarding the performance of proof computation and the performance of proof verification. Note that we do not report detailed measurements of the overheads introduced by low-level system mechanisms, such as the execution times associated with the allocation of shared memory between worlds, proof, input and output copy to the memory buffer, and the world switch instruction call. The reason is that these overheads are negligible considering that the execution times of proof computation and verification take several seconds. For instance, we report a 4ms world switch cost for censoring a 32x32 pixel matrix, which takes about 10 seconds to complete.

Figure 5 depicts our proof computation performance results. The curve in Figure 5a shows the proof computation time of the image censorship application as a function of the size of the censored pixel square expressed as the number of pixels of the square's respective side. As discussed in the section below, these images can be considered as chunks of the original picture. For a 128x128 image, the execution time starts in 540 seconds and drops to 120 seconds as the size of the censored region increases from zero until it covers the entire picture. For smaller 64x64 and 32x32 images, the proof computation times shrinks considerably to the order of 50 and 10 seconds, respectively. The reason why the proof computation times tend to decrease as the censored region size increases is because the AMF performs fewer expensive operations that consist in reading the original pixels from the input matrix to write them to the output matrix; instead, the AMF simply writes zeros to it.

Figures 5b and 5c present the proof computation performance of the location checks and heart rate average calculations, respectively. In both cases, we test these functions by increasing their input sizes based on the respective sampling rates, which change between one per hour to one per minute for a one hour and one day sampling period respectively. We see that, although their input sizes are different (the medical measurements receive larger input sizes), the performance of both applications is comparable, exhibiting execution times which vary between 10 and 12 seconds. The factor that evens out their performances is a difference in the complexity of their computations: whereas the medical measurements AMF executes arithmetic computations for the average calculation, the location check AMF needs to perform more expensive nested if conditions, which are required to compare the latitude and longitude of coordinate pairs. Similar to the 32x32 image censoring application, the costs of varying the number of inputs in this range (1 to 1440 = 24h x 60 min for medical measurement) when compared with the 128x128 image censoring example (1 to 16384 = 128x128) produces reduced differences in the performance of the AMF with the different input sizes.

With respect to the proof verification performance, our main results are depicted in Figure 6. In general, they all follow the same trend as in their counterpart proof computation experiments. The main difference between them is a decrease in the total execution times by nearly one order of magnitude. We can then see that the cost of proof verification is significantly lower that the cost of proof computation.

6.3 Performance and Security Discussion

Performance discussion. Based on our experimental evaluation, the costs involved in the computation and verification of AMF proofs tend to be relatively high. Nevertheless, such numbers can be acceptable for use cases where the AMF functions are meant to be executed with a relatively low frequency and do not demand a quick response time, such as in the examples we used above. A possible optimization is to employ probabilistic verification. Instead of automatically computing proofs upon function execution, AMFs can be packaged with logic which would first query the SafeChecker service for authorization of execution proof calculation.

Regarding the performance results, we make two final remarks. First, the obtained performance numbers are the product of our raw adaptation of a state of the art computation verification protocol provided by libsnark; libsnark is a prototyped library which has not been extensively optimized. Second, we underline that it was not our goal to advance the state of the art in verifiable computing algorithms. Yet, we designed SafeChecker so that further improvements in verifiable computing may be incorporated into our system so as to improve the execution efficiency of our system.

Security discussion. SafeChecker can prevent compromised OSes or applications from interfering with the execution integrity of AMFs. For instance, buffer overflow exploits to undermine the calculation of execution proofs may be detected by the verification process. The same holds for code injection attacks aiming at modifying the original function's logic. Note that AMFs do not preclude a compromised OS from accessing application data, nor is it their original goal: AMFs are concerned about integrity verification.

We discuss some attacks that SafeChecker does not currently address, and possible solutions. SafeChecker is unable to handle DoS attacks preventing the communication between worlds, hence undermining the execution of our verification mechanism. In fact, SafeChecker is oblivious of which proofs are supposed to be verified, and if it received all verification requests sent. Nevertheless, if such attacks were to happen, they could be detected, as the resulting communication failure between notification services and SafeChecker's kernel, could at the very least lead to an abnormal behaviour notification to the verifier.

With respect to AMF code vulnerabilities, execution correctness cannot be used to infer the benign nor malicious nature of a function. If a function is malicious, a positive execution proof simply proves the function executed as intended. In that regard, SafeChecker relies on current application markets' vetting effectiveness, similarly to every mobile OS environment. Therefore, SafeChecker's runtime verification mechanism provides execution correctness assurances over previously vetted functions running on untrusted platforms.

7 RELATED WORK

We cover the related work focusing on three main areas: verifiable computing, TrustZone, and Android security.

Verifiable computing emerged 30 years ago, with proposed theoretical work that was by the time non-practical [5]. Notwithstanding, the work by Setty et al. [21, 22] has pushed the field into a near-practical state with performance improvements of 20 orders of magnitude. The recent emergence of a succinct, non-interactive, sound proof of knowledge (SNARK) cryptographic primitive [8] further evolved the field and fostered new advances in zero-knowledge proofs. ADSNARK [6] allows users to calculate execution proofs over functions handling their sensitive data, and provide those proofs as execution compliance to service providers, without revealing the data. Pequin [3] is a verifiable computing toolchain whose front-end transforms C programs into sets of arithmetic constraints, which are then fed to one of libsnark's proof systems, Pequin's backend. Zerocash [20] leverages libsnark's zero-knowledge proofs to build a privacy-preserving version of Bitcoin. More recently, Giraffe [24] implements a new probabilistic proof built on protocol T13 [23], yielding considerable performance improvements which could be incorporated in SafeChecker. In contrast to SafeChecker, none of these systems were ever used in mobile environments, as they mostly target the cloud.

ARM TrustZone has been widely used in mobile security, e.g., for enabling secure storage of sensitive data [15], providing secure authentication mechanisms [16], and safely control peripherals [14]. Other TrustZone-based systems aim to enhance the security of mobile OSes themselves, e.g., by implementing rootkit resilient OS introspection mechanisms [12], and tracing API function and system calls to detect malicious apps [27]. General purpose Trusted Execution Environment (TEE) systems [19] allow for the execution of security-sensitive application code inside a secure world sandbox isolated from the mobile OS. SafeChecker complements existing work with a new TrustZone-based service aimed to provide auditable mobile function support.

Lastly, untrusted Android applications have been secured with data shadowing [9], application sandboxing [7], or information flow control techniques [11]. Other systems refine Android's permission model [26], implement mandatory access control (MAC) [10] and enable kernel-level hooking APIs for user-level access control models [13]. All this work is complementary to ours, since we are introducing a fundamentally new OS primitive which is primarily concerned about providing integrity guarantees of code execution.

8 CONCLUSIONS

We presented SafeChecker, a computation verification system for mobile devices. AMFs allow service providers to check the integrity of client-side remote computations without requiring the migration of sensitive data off their clients' mobile devices. The design of SafeChecker combines in an original fashion techniques from two different domains: verifiable computing and ARM TrustZone. Despite the computational costs introduced by existing verification protocols, the performance overheads of SafeChecker can be acceptable for use cases where AMFs are executed infrequently.

Acknowledgements: We thank the anonymous reviewers for their comments and suggestions. This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) via projects PTDC/EEI-COM/29271/2017 (Cosmos), SFRH/BSAB/135236/2017, and UID/CEC/50021/2013.

REFERENCES

[1] [n. d.]. EU General Data Protection Regulation (GDPR). https://www.eugdpr.org/. Accessed October 2018.

[2] [n. d.]. libsnark. https://github.com/scipr-lab/libsnark. Accessed October 2018.

[3] [n. d.]. Pequin. https://github.com/pepper-project/pequin. Accessed October 2018.

[4] [n. d.]. The Genode OS Framework. http://genode.org. Accessed October 2018.

[5] László Babai. 1985. Trading Group Theory for Randomness. In *Proc. of STOC.*

[6] Michael Backes, Manuel Barbosa, Dario Fiore, and Raphael M Reischuk. 2015. ADSNARK: Nearly Practical and Privacy-Preserving Proofs on Authenticated Data. In *Proc. of S&P.*

[7] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. 2015. Boxify: Full-fledged App Sandboxing for Stock Android. In *Proc. of USENIX Security.*

[8] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *Advances in Cryptology–CRYPTO.*

[9] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. 2011. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proc. of HotMobile.*

[10] Miguel B Costa, Nuno O Duarte, Nuno Santos, and Paulo Ferreira. 2017. TrUbi: A System for Dynamically Constraining Mobile Devices within Restrictive Usage Scenarios. In *Proc. of MobiHoc.*

[11] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of OSDI.*

[12] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. 2014. SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture. In *Proc. of MoST.*

[13] Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. 2014. ASM: A Programmable Interface for Extending Android Security. In *Proc. of USENIX Security.*

[14] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. 2018. SeCloak: ARM Trustzone-based Mobile Peripheral Control. In *Proc. of MobiSys.*

[15] Xiaolei Li, Hong Hu, Guangdong Bai, Yaoqi Jia, Zhenkai Liang, and Pratiksha Saxena. 2014. DroidVault: A Trusted Data Vault for Android Devices. In *Proc. of ICECCS.*

[16] Dongtao Liu and Landon P Cox. 2014. VeriUI: Attested Login for Mobile Devices. In *Proc. of HotMobile.*

[17] He Liu, Stefan Saroiu, Alec Wolman, and Himanshu Raj. 2012. Software Abstractions for Trusted Sensors. In *Proc. of MobiSys.*

[18] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2017. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *Proc. of NDSS.*

[19] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. In *Proc. of ASPLOS.*

[20] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *Proc. of S&P.*

[21] Srinath TV Setty, Richard McPherson, Andrew J Blumberg, and Michael Walfish. 2012. Making Argument Systems for Outsourced Computation Practical (Sometimes). In *Proc. of NDSS.*

[22] Srinath TV Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J Blumberg, and Michael Walfish. 2012. Taking Proof-Based Verified Computation a Few Steps Closer to Practicality. In *Proc. of USENIX Security.*

[23] Justin Thaler. 2013. Time-Optimal Interactive Proofs for Circuit Evaluation. In *Advances in Cryptology–CRYPTO.* 71–89.

[24] Riad S Wahby, Ye Ji, Andrew J Blumberg, Abhi Shelat, Justin Thaler, Michael Walfish, and Thomas Wies. 2017. Full Accounting for Verifiable Outsourcing. In *Proc. of CCS.*

[25] Michael Walfish and Andrew J Blumberg. 2015. Verifying Computations without Reexecuting Them. *Commun. ACM* 58, 2 (2015), 74–84.

[26] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. 2011. Compac: Enforce Component-Level Access Control in Android. In *Proc. of CODASPY.*

[27] Sileshi Demesie Yalew, Gerald Q Maguire, Seif Haridi, and Miguel Correia. 2017. T2Droid: A TrustZone-based Dynamic Analyser for Android Applications. In *Proc. of Trustcom.*