Andromeda: Enabling Secure Enclaves for the Android Ecosystem

Dimitris Deyannis[1,2,3], Dimitris Karnikis[2], Giorgos Vasiliadis[2], and Sotiris Ioannidis[4,2]

[1] Sphynx Technology Solutions AG
d.ntegiannis@sphynx.ch
[2] FORTH-ICS, Heraklion, Crete, Greece
{deyannis. dkarnikis, gvasil}@ics.forth.gr
[3] University of Crete, Heraklion, Crete, Greece
deyannis@csd.uoc.gr
[4] Technical University of Crete, Chania, Crete, Greece
sotiris@ece.tuc.gr

Abstract. The Android OS is currently used in a plethora of devices that play a core part of our everyday life, such as mobile phones, tablets, smart home appliances, entertainment systems and embedded devices. The majority of these devices typically process and store a vast amount of security-critical and privacy-sensitive data, including personal contacts, financial accounts and high-profile enterprise assets. The importance of these data makes these devices valuable attack targets.
In this paper we propose Andromeda, a framework that provides secure enclaves for Android OS to mitigate attacks that target sensitive or critical code, data and communication channels. Andromeda offers the first SGX interface for Android OS (to the best of our knowledge), as well as services that enhance its security and offer protection schemes for several applications that deal with sensitive or secret data. Andromeda is also able to securely execute SGX-enabled code on behalf of external devices that are not equipped with SGX-capable CPUs. Moreover, Andromeda protects cryptographic keys from memory dump attacks with less than 16% overhead on the corresponding cryptographic operations and provides secure, end-to-end encrypted, communication and computation channels for external devices paired with the Android device.

1 Introduction

Android has become a very popular open-source operating system that targets a large set of devices [11], including mobile phones, tablets, smart home appliances, entertainment systems and embedded devices. All these devices play a core part of our everyday life and usually process and store a vast amount of privacy-sensitive data, such as personal info, financial accounts, cryptographic keys and high-profile enterprise assets. The importance of this data makes these devices a valuable target for attacks and forces enterprises and device owners to be concerned about the security of the data stored on them.

Furthermore, Android is also used as a hub for a diverse set of smaller devices, such as wearables, web-cams, sensors, control and automation systems, etc. These external devices act as data producers (e.g., image/video capturing, motion sensors, temperature/humidity sensors, activity trackers etc.), sending all of their data to a corresponding application that runs on Android. Several different application frameworks do currently exist, such as Samsung SmartThings [12] and Android Sensor API [4], that enable third-party developers to build apps that compute on such, typically sensitive, data. Even though such applications allow the user to easily access the data, still at the same time they are being posed to significant risks as data is usually left unprotected and prone to misuse/abuse by unverified processes. As such, enabling applications to compute on sensitive data that external devices generate (such as surveillance material, heart rates, activities performed, motion patterns), while preserving the integrity of data and preventing any unwanted or malicious abuse, is an important problem. The protection of sensitive data is even more difficult to be achieved in such use cases, since external and wearable devices are not equipped with trusted components. In most cases, the only option available to protect the sensitive data they produce is to use the TEE offered by other (remote) devices, if available.

To mitigate such attacks and protect user data, many operating systems or frameworks that target such devices deploy permission-based access control mechanisms, such as authentication and disk encryption. For instance, IoT frameworks, such as Bosch's IOT [6] and Amazon's AWS [1], use permission-based access control for data sources and sinks, however they do not control the flows between the authorized sources and sinks [26]. Many approaches leverage hardware-based trusted computing techniques to isolate the execution of applications [17,33,38,23]. For instance, several works utilize ARM TrustZone [14] to run security-sensitive code or protect security-critical data, such as cryptographic keys and payment information [31,39,32]. However, TrustZone is shared simultaneously by all applications since there is only one TEE provided by the hardware. Thus, by design, it can not provide isolation between the applications that utilize the TEE, as they all co-reside in the same secure space. As a result, if one of the trusted applications goes rogue, any other application that runs in the secure world can possibly be affected. This prevents it from being universally leveraged simultaneously across different applications, either in user-space (e.g., banking applications, etc.) or kernel-space (security monitors, device keystore, etc.). In addition, TrustZone does not protect against attackers with physical DRAM access. Moreover, although TrustZone is provided by almost all ARM processors, it can not be directly used by application developers; it requires control of the device and its firmware, which is not the case in many cases.

In this paper we introduce Andromeda, a framework that provides secure enclaves for Android OS so Android developers can explicitly use them for their applications, either by using the native API in C/C++ or our Java interface that provides access to the secure enclaves through JNI bindings. In contrast to previous approaches, Andromeda has the potential for multiple enclaves in a system simultaneously, making it more flexible for general-purpose security-critical

operations, offering per-application or per-function isolated secure environments. In addition, Andromeda implements popular Android services, enhanced with secure enclaves capabilities, hence securing and protecting their functionalities. We offer two representative services (i.e., a secure key management system, and a data protection scheme for data flows) that enhance the security of Android OS and offer protection schemes for several applications that deal with sensitive data (such as cipher keys, personal data, medical data, etc.). These services enable Andromeda to support an efficient and robust end-to-end encrypted data flow model in which external devices that pair with Android can securely transfer and process their data in the Android device, or even with a remote cloud-service.

We have currently implemented Andromeda prototype for Intel CPU processors with SGX support; any device that is equipped with a SGX-enabled processor can run Andromeda natively, out of the box, including handheld devices, convertibles, set-top boxes, and car entertainment units. However, we have to point out that Andromeda is not bound to Intel SGX; instead the proposed mechanisms could be implemented on top of other architectures offering secure user-level enclaves. For instance, there are approaches that implement user-level secure enclaves, compatible to SGX, either independent of the underlying CPU (such as Komodo [27]) either on top of ARM TrustZone (such as Sanctuary [20]); Andromeda is not fundamentally tight to Intel SGX and, as such, could be implemented on top of such approaches instead. Besides that, we note that a number of vendors are developing similar hardware protection mechanisms, including AMD SEV [2] and IBM's SecureBlue++ [19]. Even though these mechanisms are not identical, many of the proposed techniques of Andromeda can be adapted to use these hardware features, the need of which will increase in the future.

The contributions of our paper are the following:

- We present a systematic methodology to port the SGX framework for the Android OS, including the SGX kernel driver, the required libraries and background services needed for its operation and a custom cross-compiler (§ 5). This allows Android developers to explicitly use SGX for their applications either by using the native API in C/C++ (§ 6.2), or our proposed Java interface that provides access to the secure enclaves through JNI bindings (§ 6.3).
- We implement popular Android services, enhanced with SGX capabilities, hence securing and protecting their functionalities (§ 4.2). The SGX enclaves enable multiple secure spaces that can be used simultaneously by different applications, in contrast with other TEE ecosystems, such as ARM TrustZone, that allow only a single secure space that is shared for everyone and often times requires control of the device and its firmware.
- We implement a programming paradigm tailored for externally paired devices, that enables a robust, efficient, and trusted data flow between external devices that pair with the Android OS (§ 4.2). Such devices can securely offload data storage and computations to the Android OS in a trustworthy manner, without necessarily being equipped with TEE-enabled CPUs.

2 Background

2.1 Intel SGX

Intel SGX [8] is a technology for application developers who are seeking to protect selected code and data from disclosure attacks or modifications. Intel SGX makes such protections possible through the use of enclaves, which are trusted execution environments for applications. Enclave code and data reside in enclave page cache (EPC), which is a region of protected physical memory. Both enclave code and data are guarded by CPU access controls, and are also cache-resident. Every time the data are moved to DRAM, they are encrypted via an extra on-chip memory encryption engine (MEE), at the granularity of cache lines. For Intel Skylake CPUs [9], the EPC size is between 64 MB and 128 MB and SGX provides a paging mechanism for swapping pages between the EPC and untrusted DRAM.

Enclave memory is also protected against memory modifications and rollbacks, using integrity checking. Non-enclave code cannot access enclave memory, however enclave code can access untrusted DRAM outside the EPC directly. It is the responsibility of the enclave code, however, to verify the integrity of all untrusted data. Application code can be put into an enclave by special instructions and software made available to developers via the Intel SGX SDK. The Intel SGX SDK is a collection of APIs, libraries, documentation, sample source code, and tools that allows software developers to create and debug Intel SGX enabled applications in C and C++ and is targeted for x86_64 computer systems.

2.2 The Android OS

Android is an operating system mainly designed for small handheld smart devices, including but not limited to mobile phones, tablets and watches. It is being developed by Google LLC, was first released in 2007 and is currently the most widespread OS for smart devices[13][10]. Android's backbone is based on the Linux kernel, thus granting it extensively tested security features and stability, and also allowing developers and manufacturers alike to develop hardware drivers for a well known kernel. Google also had to make a few additions in order to provide a more customised kernel functionality for Android's requirements. A few key additions are the wakelocks, a power management component crucial for mobile devices, a unique out of memory (OOM) handling also informally known as 'Viking Killer', the ashmem, a new shared memory allocator for low-memory devices, pmem a process memory allocator and also Binder an Android specific interprocess communication mechanism and remote method invocation system essential to Android, due to the fact that it does not support the use of the Linux SysV IPC.

Android is built on top of the Linux kernel with components such as the hardware abstraction layer (HAL), which provides various standard interfaces that allow higher Java APIs and code to make use of a device's hardware components, and the Android Runtime (ART), a special virtual machine similar to Java's JVM, designed to run on low-memory devices. There are also Native

C/C++ Libraries and both HAL and ART are written in C/C++, however these native libraries do not provide the same functionality as they would in a native Linux machine. On the top layer of the Android architecture, there is the Java API Framework, which provides applications a means to access the other layers in a constant way throughout different machines. All Android applications, while able to use native C/C++ code, are developed in Java, enabling them to be executed on multiple and different devices.

The majority of cryptographic operations in Android, including encryption, decryption, message authentication (MAC), key generation and agreement, are handled by the Android Keystore [3], that also provides a central place for storing cryptographic keys for all applications. Keymaster is a part of the Android Keystore service and responsible for generating new keys for encrypting, decrypting and hashing data. It supports various cryptographic functions like AES, RSA, SHA and more. In order to generate such an encrypted key for an application and perform cryptographic operations, one has to generate a `SecretKey`, initialize a `Cipher` with the desired mode (encrypt, decrypt or other) and choose the appropriate algorithm and its properties for the current operation. Android defines an abstract programming interface that can be used for the third-party implementations, plugged in seamlessly as needed. Therefore application developers may take advantage of any number of provider-based implementations without having to add or rewrite code.

3 Threat Model and Assumptions

In this work, we assume a powerful and active adversary who has root privileges and access to the physical hardware (with the exception of the CPU) as well. The adversary can control the entire software stack, including the OS kernel and other system software. However, we explicitly exclude denial-of-service (DoS) attacks on enclaves, given that the design of SGX allows the host OS to control an enclave's life cycles anyway. As a result, an attacker can prevent or abort the execution of enclaves, but should not gain any knowledge by doing so. Moreover, side-channel attacks [21] that exploit timing or page faults or based on vulnerabilities of the application running inside the enclave are proven to be feasible on SGX enclaves. However, protecting SGX enclaves from side-channel attacks that either focus on software or hardware bugs is orthogonal to Andromeda and thus we consider that it is out of scope of our work. However, any successful attempt to protect SGX-enabled code/hardware has a direct benefit to our framework. Finally, we assume the design and implementation of SGX itself, including all cryptographic operations, is secure and does not contain any vulnerabilities.

4 Andromeda Architecture

Our objective is to offer secure enclaves for the Android OS which must protect sensitive services from the threats defined in Section 3. This will enable Android developers to explicitly leverage them for their applications. We also want to

utilize secure enclaves inside Android services that operate on sensitive data (such as Keystore), so they can be used transparently by applications. Overall, Android developers should be able to build their applications and make use of the secure enclaves as transparently as possible, ideally without writing extra code or heavily modifying existing applications.

An enclave cannot be initiated on its own but instead the Intel Launch enclave must be used to generate the appropriate launch token. In addition, an enclave's code always has to be executed in Ring-3 with a reduced set of allowed instructions and a limited amount of available memory. Thereby, we decide to build an architecture that runs solely on the user-space, providing the interface and the services that Android applications can use in an expressive and flexible way. Figure 1 gives an overview of the Andromeda architecture. It comprises of different layers that can be used by different kinds of applications for different purposes. Using these mechanisms, we enhance popular Android services, such as the Device Pairing and Keystore service, to leverage secure enclaves internally in order to increase their security in a robust and transparent way. Finally, we also implement an environment, within SGX, so external devices that have paired with Android can securely transfer and store sensitive data on the Android device. Andromeda is responsible to protect all sensitive data by encrypting them across the full path from the external device to the Android OS. Further, Andromeda optionally enables the processing of these data via functions that the data-publishing application has submitted for execution in the SGX enclaves.

4.1 Trusted Execution and Storage

Andromeda provides a trusted execution and data storage service on top of SGX. The service can be used by local Android apps, as well as from remotely paired devices, as described in Section 4.2. At the lowest level, applications can use the native API provided by the SGX runtime libraries, in order to achieve the maximum performance. The process of utilising secure enclaves in an application developed in native C/C++ code remains the same as for every other native C/C++ Android application. The developer needs to prepare and integrate the Intel SGX counterpart of the application (similar to the Linux environment) and then cross-compile the application with our custom Android tool-chain, which is able to handle the compilation of both trusted and untrusted parts of the code. Developing Intel SGX enclaves for an APK implemented using Java requires the use of JNI bindings. For this reason, we provide a Java API (described in Section 6.3, which wraps the SGX functionalities in appropriate classes. The developer needs to extend these classes with methods that will be eventually executed in the SGX enclave of the application and perform the code compilation using the Andromeda tool-chain which also provides JNI bindings for each SGX-enabled function requested. In this way, the developer can easily interface with the enclaves from the APK level. Moreover, Andromeda provides the implementation of a secure data vault system and exposes a simple Java API for Android applications. Using the data vault service, applications can securely store data inside the SGX enclaves or seal them for secure file system storage.

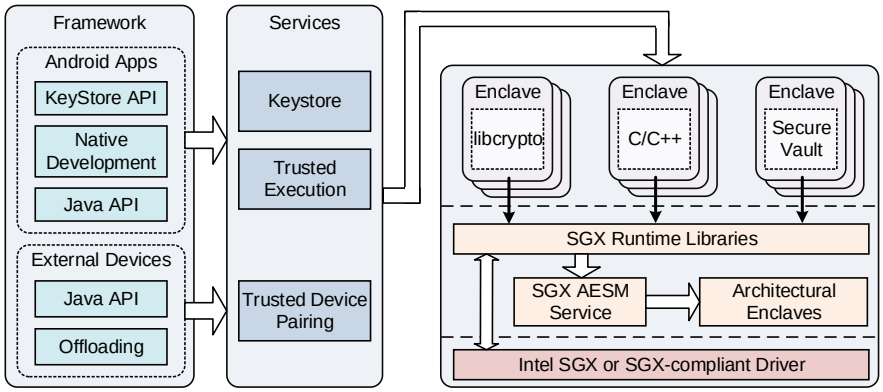


Fig. 1: Architecture of Andromeda

4.2 Andromeda services

Keystore Service The main purpose of Android Keystore is to store cryptographic keys and offer cryptographic operations in a secure container, protecting them from tampering. However, if not implemented with secure hardware support, it is vulnerable to a broad set of attacks, as described in Section 3. Having the secret and private keys stored in clear-text makes them an easy target for a malicious software running on the device. Andromeda offers the mechanisms to keep the secret keys in a protected space, within secure enclaves, thus solving and overcoming leakage scenarios.

The Keystore is implemented in C/C++ while Android uses a binder to communicate with the Java part. Internally, Android Keystore can handle different type of entries. Some of them are `PrivateKey`, `SecretKeyEntry` and `TrustedCertificateEntry`. Each one of these entries is identified by an alias name which corresponds to the Keystore entry. When generating such an entry, it is possible to choose from a range of cryptographic algorithms available in the Keystore or use the default. In this way, the Android Keystore is able to store multiple keys simultaneously, regardless of type, name and algorithm. At the same time, different running programs can utilize the Keystore and store their keys without having to deal with collisions.

An overview of our SGX-enabled Keystore operation is illustrated in Figure 2. A major advantage of Andromeda Keystore is that it can be used even by legacy apps without any code modifications or recompilation. The simplest way is to have the entire Keystore inside a single enclave. However, this design leads to a large TCB that is generally harder to review, or possibly verify, and is assumed to have more vulnerabilities. To overcome this problem, we place in secure enclaves only three core operations, which are used by the majority of cryptographic algorithms: (i) the key generation, (ii) the data encryption, and (iii) the data decryption. By doing so, we ensure that all private and secret keys reside in secure enclaves while having a small TCB that can be easily verified. The memory for

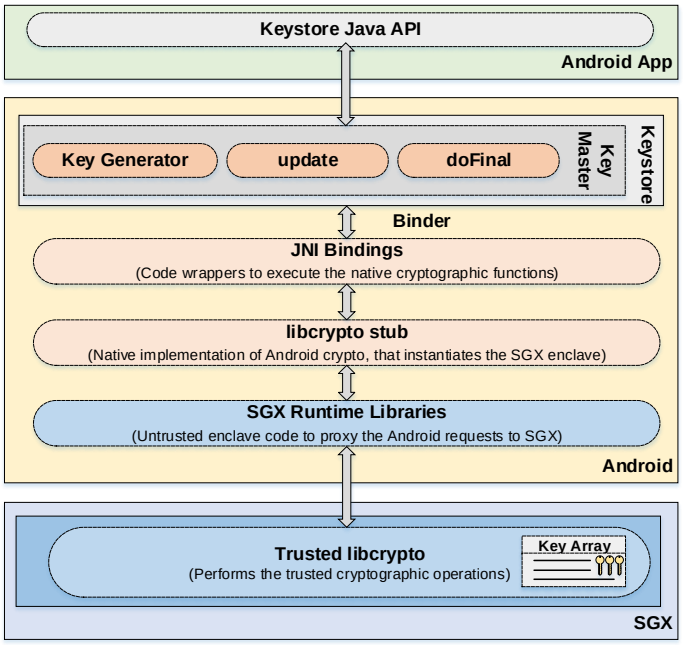


Fig. 2: The Keystore Architecture. The cipher keys are stored only in SGX enclaves. Developers can encrypt and decrypt their data using the default Keystore API, which internally redirects to Andromeda's trusted implementation.

the keys is allocated inside the SGX enclave and only their pointers are returned to the user-space, preventing any attempt to read them, extract them or modify them, even via physical access to the device's DRAM.

Our current implementation uses RSA-1024 and AES Counter Mode (AES-CTR); we note though that other modes can be easily implemented. AES divides each plain-text into 128-bit fixed blocks and encrypts each block into cipher-text with a 128-bit key. The encryption algorithm consists of 10 transformation rounds. Each round uses a different round key generated from the original key using Rijndael's key schedule. The whole encryption and decryption occurs inside the SGX enclave, ensuring that keys and all intermediate states are well protected. Similarly, we have implemented RSA encryption and decryption.

Trusted Device Pairing Andromeda provides secure device pairing between devices, even when only one (i.e., the Android device) is equipped with an SGX-enabled processor. Such scenarios are typical when small external devices, such as sensors and wearables with limited security capabilities, need to be paired with more powerful Android devices (i.e., a phone or gateway). To accomplish secure device pairing and attestation in such use cases, Andromeda offers the

functionality that enables the external devices to securely connect with the SGX-capable Android device. The main concept is that data-publishing wearable or external devices can protect their sensitive data, so it will only reside or processed within designated functions that run in SGX-provided enclaves.

First, Andromeda generates a key pair and distributes the public key to the external device and the corresponding private key to a local secure enclave. Each external device has its own secure enclave, to ensure isolation between each other. These keys can be used later to establish a session key via Diffie-Hellman. The process of establishing and storing the keys is performed entirely inside SGX enclaves in the case of the Android device. We assume that the external device runs on a minimal code base with limited I/O, thus the integrity of the key management can be attested and preserved. While this end-to-end encryption of the I/O channel ensures data protection during transfers, the need of attestation between the two devices remains a critical point in order to prevent malicious users impersonating as one of the two devices. In cases where the external device is capable to execute the Intel Remote Attestation process it is able to verify that it is indeed communicating with a secure enclave, running on SGX-capable hardware without emulation. However, in some cases, Intel Remote Attestation can not be performed due to the limited computing capabilities of many external devices. To overcome this, we utilize one-time passwords (OTP) instead, which are an essential part for our remote attestation alternative procedure. More specifically, we use Google key generator to create an arbitrary key that we can then register with a secure SGX enclave. The registration is performed at the first connection and Andromeda (optionally) prompts the user to verify the registration. Once the key has been successfully registered, the attestation procedure starts by the external device demanding a 6-digit OTP to be exchanged. The generated OTPs are based on the RFC 6238. Upon receiving the OTP, the external device calculates an OTP with the same key. If both match, the external device can be certain that it communicate with the SGX enclave, since the entire OTP process is performed inside the enclave. Once the OTP is verified, the secure communication channel is established as described above.

5 Implementation

5.1 Setting up SGX for Android

Cross-compiling Intel SGX for Android OS is a challenging task. Due to the complexity of the software and the many differences between a Linux distribution and Android, we have to split the porting process in several smaller tasks in order to constantly proving the potential and validity of our goal. For this reason we perform the Android port in the following steps. First, we compile the SGX SDK for a different Linux distribution than Ubuntu, which is the officially supported, namely Arch Linux. Since Android is also based on Linux, this process lets us understand how different compiler and library versions affect the possibility of porting SGX on Android. Second, we validate that we can build the Android Open Source Project (AOSP) form scratch and successfully install and run it on

an SGX-capable x86 machine. Finally, we integrate the SGX functionality into the AOSP source tree by cross-compiling and providing the necessary libraries for its correct operation.

The whole process of building the SGX environment for a non supported Linux distribution is a quite tedious procedure due to the kernel, compiler and library version incompatibilities. While analyzing the dependencies of SGX SDK we find the following to be essential for a standard enclave execution: (i) the SGX kernel driver, (ii) `aesm_service` which is a background daemon serving as a management agent for SGX enabled applications, (iii) the `libsgx_urts.so` and `libsgx_uae_service.so`, needed for executing enclaves in hardware mode, the `libsgx_urts_sim.so` and `libsgx_uae_service_sim.so`, needed for the software emulation mode, and finally (iv) the `le_prod_css.bin` and `libsgx_le.signed.so`. This analysis allowed us to understand the software requirements and the process of building the SGX environment for an unsupported platform.

Porting SGX on Android is an even more complicated process. First, AOSP has to be built from scratch and be installed on an SGX-enabled x86 machine. Then, porting the SGX environment is a time-consuming process since each change to the source tree requires to (i) build the Android image, (ii) flash it on the host machine and (iii) verify the correctness of each change as well as the stability of the system. The SGX SDK is designed to be build on desktop-based Linux distributions using GCC $>$ v5 while Google's NDK (Native Development Kit for Android) offers `GCC-4` and `clang` that are not able to compile the SGX source tree. For this reason, we use CrystaX NDK [7] which acts as a drop-in replacement for Google's Android NDK, offering GCC-5.3 compatibility. Also, the SGX SDK contains a group of libraries that must be compiled for Android in order for the environment to execute properly, such as `protobuf`, `ssl`, `libssp`, `curl`, `gperftools` and `libunwind`. To cross-compile them, we need to export and set the corresponding flags for the Makefile and configuration files of each project to link to the CrystaX compiler by setting the `cross_compiling` field to `true`. Then, all references to `lpthread` have to be removed from the Makefiles, since it is automatically linked at the Android version of the standard library. Moreover, due to the stripped down kernel version that is used by the Android OS, the `RDRAND` instruction that is used by `sgx_read_rand` to perform random number generation is not available. To overcome this issue we use a software based implementation for random number generation that is fully compatible with the existing API and works on Android and SGX.

After successfully cross-compiling the SGX source tree, the final step is to cross-compile the kernel driver and port it to Android. Unfortunately, there are inconsistencies between the supported kernel used by Ubuntu and the Android kernel headers and the signatures of several kernel functions are different. For this reason, some patches are required in order to build the driver which also requires to be built in-source with Android. Once the SGX porting is completed, we build a demo application that utilizes SGX enclaves in both `hardware` and `simulation` mode. Finally, in order to execute Intel SGX enclave code, the application must be signed using Intel's `sgx_sign` tool, which we rebuild and

use in order to compile Android applications as needed. The problem is that cross compiling the whole Intel SGX source developing tools (SDK) and platform software (PSW), would produce the `sgx_sign` binary that is only executable on Android; this would be quite inflexible to build an application and then sign it at the Android using the application. Instead, we rebuild the source but this time using only the Ubuntu default tools, store the `sgx_sign`, and then use it to compile our applications when needed.

5.2 Running an SGX application

An SGX application can run either in `hardware` or `simulation` mode. To make use of the underlying hardware and leverage Intel SGX as a service, we compile SGX applications using `make SGX_MODE=HW` which links against `libsgx_urts.so`. Of course, since these libraries are not available in the source tree of Android they must be provided to the `LD_LIBRARY_PATH` of the corresponding application by exporting the paths of each one of them. Apart from the required SGX dependencies, the libraries that were linked during the SDK compilation must be also provided and exported to the `LD_LIBRARY_PATH` of the given application. Additionally, we use `insmod` to load the driver and then start the `aesm_service`. The Android service system has several differences compared to Linux; editing a system service file like `init.d` is not enough for Android to deploy a new system service. Instead, a new application, marked as a service, has to be created and meet specific code requirements [5]; i.e., all native functions of `aesm_service` need to be wrapped with JNI calls for it to be accessible by the Java part.

To overcome this issue, we simply adjust the `aesm_service` source code to run as a daemon in the background and interact directly with the native part. The other solution would be to discard the whole Android application part and interact with the native part directly. By examining the source code of `aesm_service` we manage to run the application as daemon (which is essentially a service) so the app would start and stay alive. Whereas, if we start it without the specified input it would just terminate with no output. Also, the `aesm_service` requires the `le_prod_css.bin` and `libsgx_le.signed.so` binaries to properly execute so we transfer these binaries from the Intel SGX output directory to the `aesm_service` directory in Android before its execution. Finally, running an application in Android requires it to be built with the `-pie` and `-fPIE` flags. These flags instruct the linker that the program's code can be executed regardless its absolute address. After all the aforementioned requirements are met, we are able to cross-compile and execute SGX-enabled Android applications.

Enclaves can be created using the `ECREATE` instruction, which initializes an SGX enclave control structure (SECS) in the EPC. The `EADD` instruction adds pages to the enclave, which are further tracked and protected by the SGX (i.e., the virtual address and its permissions). The `EINIT` instruction creates a cryptographic measurement, after the loading of all enclave pages. The cryptographic measurement can be used by remote parties for attestation. After the enclave has been initialized, enclave code can be executed through the `EENTER` instruc-

tion, which switches the CPU to enclave mode and jumps to a predefined enclave offset. The `EEXIT` instruction causes execution to leave the enclave.

6 Andromeda Framework

The Andromeda framework is split in three parts: (i) the enclave-enhanced Android Keystore, which can be utilized transparently, (ii) the native API, used to initialize and configure SGX using native code, and (iii) the Java API, which provides a set of building blocks for APKs.

6.1 Andromeda Keystore

The Android apps can transparently utilize the Andromeda Keystore service to securely perform cryptographic operations. Private keys and other sensitive information are kept in encrypted form in an array that resides in SGX memory and cannot be accessed in any way by the host. To perform a cryptographic operation: (i) the required (encrypted) key is fetched from the array, (ii) it is decrypted inside the enclave, and (iii) the actual operation is performed on the input data. This extension of the Android Keystore, provided by Andromeda, is completely transparent to the developer. All necessary modifications are performed at the native C/C++ part of Android's Keystore while the corresponding Java API remains unmodified, rendering it completely backwards compatible with legacy applications. Persistent secure storage of keys and important metadata can be achieved using the sealing technique. The Keystore service will seal and export the contents of the secure enclaves to the specified file-system locations, protecting them during unexpected execution termination or device power-off. The exported data are encrypted and accompanied with the necessary metadata that ensure their validity. Once Keystore's enclaves need to be re-enabled, the service will repopulate them by loading and unsealing the data. If the data is invalid or tampered, the service provides the necessary exceptions.

6.2 Native Development

Using the Andromeda SGX tool-chain, developers can create their own SGX enclaves for their Android applications. To do so, native code in C/C++ has to be developed for the enclave functionality as well as the respective `ECALL`s and `OCALL`s that manipulate the data (sensitive or not) in the trusted and the untrusted part. In order to access the SGX code and functions, JNI bindings must be provided to the Java part of the APK to connect it with the native C/C++ and SGX counterpart. These JNI functions must be written in order to initialize the enclave instance, setup the environment and access the secure enclave code, functions and data. The process is quite similar with a Linux environment; the basic difference with SGX-enabled Android applications is that all native C/C++ code that implements the SGX enclaves and the native C/C++ code that handles their execution should be cross-compiled with the Andromeda Android tool-chain which handles all the steps required to build the source tree.

6.3 Andromeda Java API

In order to assist the development of SGX-enabled Android applications, Andromeda also offers an API that developers can use to offload specific parts of the code into secure enclaves. The Andromeda Java API provides a set of building blocks for APKs and automates the generation process of secure enclaves that execute only minimal parts of the application logic in the trusted environment. The Andromeda Java API are shown in Table 1 and allows the creation of enclaves, the configuration of input and output between enclaves, and the execution of user-defined functions.

Secure Execution The Java functions provided by the Andromeda API offer the following functionality: The developer can create a new secure enclave Java class instance using the `TrustedEnvironment()` constructor. To make the establishment of the trusted environment, the secure enclave Java class provides the `load()` method that passes configuration settings and user-defined configuration extensions to the enclave. This operation will generate a new enclave using the C/C++ layer of the Andromeda API and provide the necessary handles to the Java counterpart in order to interface with the enclave. The enclave and its metadata can be securely erased using the `destroy()` method, which optionally passes finalization data to the enclave. Developers can use the `run()` method to perform a trusted execution in the secure enclave. The `run()` method is extensible and includes the code that performs the desired computations inside the SGX enclave. Andromeda also provides the option to implement multiple functions to be executed in the trusted environment which can be invoked using their respective index (using the corresponding `run()` method argument). The `run()` method can be called an arbitrary number of times with different inputs.

In contrast to the manual development of SGX-enabled Android applications, when using the Andromeda Java API the Andromeda tool-chain will generate the appropriate native C/C++ SGX code that implements the functionality defined in the `run()` method. Moreover, the tool-chain will generate the enclave driver code, that handles I/O and function calling, as well as establish connection with the Java API by creating the necessary JNI bindings.

Secure Vault API The Java functions provided by the Andromeda secure vault API enable both short term and persistent secure storage functionality. The developer can use the `store()` function in order to store a data object within a secure enclave. The data object can be of any kind, such as cryptographic keys, certificates, fingerprints, tokens or any other data considered sensitive in the scope of the application. Upon successful data storage, the API will return an index which can be used to retrieve the actual data through the `retrieve()` function. Moreover, the Andromeda Java API provides access to the SGX sealing and unsealing functionality, via the `seal()` and `unseal()` methods respectively. Using the `seal()` function, the developer can encrypt the data within the enclave using a secret key derived within SGX. Once the data are sealed, they can be

Table 1: Andromeda Java API for SGX enclave utilization.

Constructor Summary	
Constructor	**Description**
TrustedEnvironment()	Creates a new secure enclave class instance
Method Summary	
Modifier and Type	**Method Description**
void	**load**(EnclaveConfig config) Initializes the secure enclave
EnclaveOutput	**run**(int index, EnclaveInput i) Performs the trusted execution
int	**store**(byte[] data) Stores the data and returns its index
byte[]	**retrieve**(int index) Retrieves the data using its index
SealedData	**seal**(Object d) Seals the enclave data and stores to file-system
Object	**unseal**(SealedData d) Unseals the data and populates the enclave
void	**pair**(ChannelConfig config) Creates a secure connection with the external device
void	**transmit**(ChannelConfig config, byte[] data) Securely transmits data to the external device
byte[]	**receive**(ChannelConfig config) Securely receives data from the external device
void	**terminate**() Disconnects the external
void	**destroy**() Destroys the secure enclave

stored in main memory or storage with assurances of integrity and authenticity and can only be unsealed using `unseal()`. These functions can also be used to periodically generate backups of the secure storage in order to prevent data loss (e.g., from unpredictable execution termination).

Secure Pairing API The secure device pairing functionality is provided by dedicated Andromeda API methods. These methods can be utilized by the Android application controlling the external device, as long as the external device includes Andromeda's connection libraries, which do not require SGX support, in its software stack. The developer is able to establish a secure communication channel with an external device using the `pair()` method. The external device can be connected either via Bluetooth or Wi-Fi. Andromeda will then perform the attestation procedure for both devices. The configuration data passed to this method indicate the device ID, the attestation procedure (Remote Attestation or OTP), the option of notifying the user with a verification pop-up and other metadata, essential for initiating the connection. Once the attestation process is completed, Andromeda will perform the communication channel establishment automatically, as described in Section 4.2. Once communication is initiated, the devices are able to exchange data using the `transmit()` and `receive()` func-

tions respectively. Finally, the developer can execute the `terminate()` function for a `TrustedEnvironment` instance in order to disconnect the external device.

7 Evaluation

7.1 Security Analysis

We now evaluate the security properties of Andromeda by describing possible attacks and showing how our proposed design protects against them.

Memory Attacks We implement Andromeda in a way that nothing but a pointer to enclave memory is ever written into host memory. The pointer's content can not be read or modified since it resides into the enclave. When Andromeda performs the desired operations, the output is transferred back to Android memory. In the meantime, we keep the enclave execution alive completely isolated from the Android system, without being affected by side effects of the OS or hardware, such as interrupt handling, scheduling, swapping, and ACPI suspend modes.

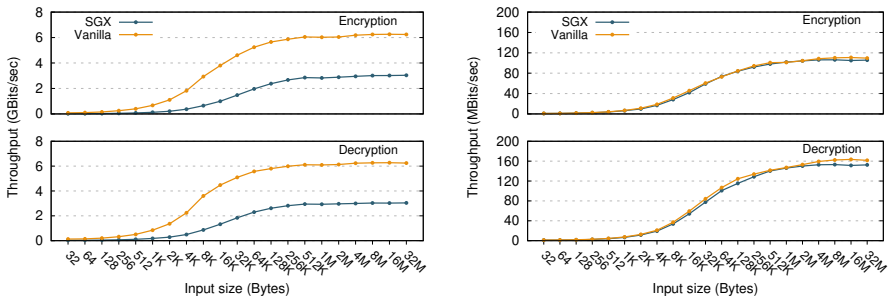
Controlling the Kernel In cases where the attackers have successfully taken full control of the Android OS kernel, any sensitive data manipulated by Andromeda is still sound and safe. Once again, even though the attackers may have full read/write/execute rights in the whole system, they cannot read/write/execute code inside the enclave. As a result, any attempt to modify or read enclave code will result in a Segmentation violation since this memory is not mappable outside the enclave code, keeping the data secured.

Integrity of data In a typical scenario, attackers can exploit software vulnerabilities and manage to inject code of their choice to a running service. Sensitive data, such as secret keys and checksums, stored in the address space of the process, can be easily acquired. In contrast, hiding sensitive data in a secure enclave prevents access even to fully privileged processes. To verify this, we attach our process with `gdb` in order to check the allocated pointers in the enclave code and trace the calls. However, no such data can be extracted since the enclave code and data are inaccessible from non-enclave code nor the function calls or memory stack. Such operations always result in Segmentation violations.

7.2 Performance Analysis

We now assess the performance of Andromeda and the extra overhead introduced for the execution of the secure enclaves. For our experiments we use an Intel NUC 8i5BEK kit with an SGX-enabled Intel i5-8259U CPU at 2.3 GHz and 8 GB of DDR4 RAM. The system is running Android x86 version 7.1.2_r33.

AES Evaluation We compare the performance of the AES-128 crypto algorithm, as achieved by the vanilla Android Keystore system, versus the SGX-enabled implementation provided by Andromeda, using a custom benchmarking tool. In each processing loop, the tool generates a random secret key and a random stream of data. The data vary in size from 32 B up to 32 MB. To avoid any potential caching effects that may result in inaccurate results, we generate a new key and data stream in each processing loop. Once an AES key and a stream of data are prepared in memory, the tool performs cryptographic operations on the data using AES-128 in CTR mode, using both the vanilla and the SGX-enhanced Keystore system, provided by Andromeda. Figure 3(a) shows the performance characteristics of the native AES code execution. We achieve this by monitoring only the AES functions found in the native C code part of the Android Keystore system. Our evaluation indicates that the overhead introduced by the SGX-enabled implementation ranges between 51% and 84% for the encryption operations and from 51% to 78% for the decryption.



(a) Microbenchmark (b) Keystore

Fig. 3: Throughput comparison between the AES-128 CTR found in Android's Keystore and the SGX-enabled version provided by Andromeda's Keystore.

In the next experiment we explore the throughput sustained in the APK scope. We achieve this by performing the same experiment but in this case we monitor the execution time of the Java cryptographic functions provided to the APK by the Keystore system (Figure 3(b)). The execution time includes the entire execution path and the overhead introduced by the various layers of the Android architecture, including the IPC, the binder and the numerous function calls until the actual cryptographic operations are performed. We notice that the sustained throughput perceived by the APK is one order of magnitude lower (compared to Figure 3(a)), due to the overhead introduced by the various layers of the software stack involved in the process (i.e., JNI, IPC, and the binder). Similarly, the perceived overhead introduced by the SGX enclaves is minimised, between 0.6% to 13% for encryption and 0.6% to 11% for decryption.

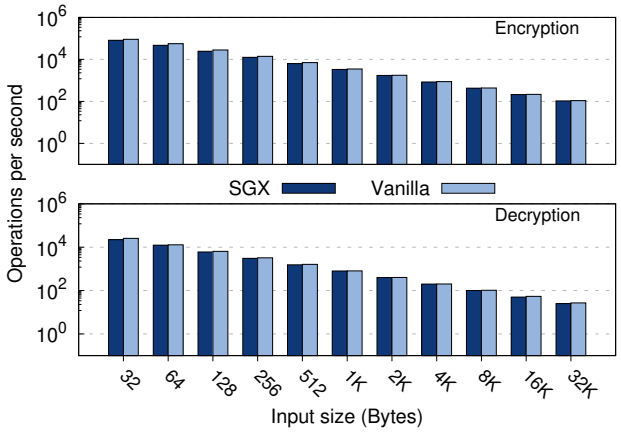


Fig. 4: Sustained throughput achieved for the vanilla and the SGX-enabled implementation of the RSA-1024 cryptographic algorithm.

RSA Evaluation We now present the performance comparison between the vanilla and our SGX-enabled implementation of the RSA algorithm. We perform the evaluation as follows. We develop a benchmarking application capable to perform RSA key generation, encryption and decryption. In each processing loop, the tool generates a new RSA key-pair and performs cryptographic operations against a set of input data. The data set consist of 10,000 random data chunks, varying in size from 32 B up to 32 KB, with each set containing chunks of the same size. We choose to generate a new set of random data in each processing loop in order to eliminate any caching effects. We execute the benchmarking application for every data set, each time monitoring the number of sustained cryptographic operations per second. The outcome of this experiment is displayed in Figure 4.

We notice that the SGX-enabled implementation introduces a maximum overhead of 16%, observed when processing 64 B long data, with the lowest introduced overhead being 2.3% during the encryption of 2 KB long data. The maximum sustained decryption rate is observed for the vanilla implementation during the encryption of 32 B long data with the introduced overhead being 12.6%. The minimum observed overhead introduced by the use of SGX enclaves is 0.9%, encountered during the decryption of 2 KB long values. For both crypto operations, we observe that the perceived overhead introduced by the I/O between the benchmarking application and the SGX-enclave is minimised due to the processing complexity of the RSA algorithm.

Computation Offloading We present the performance of three benchmarking applications, executed, using the different methods provided by the Andromeda framework, as well as the overhead introduced by executing them remotely. In

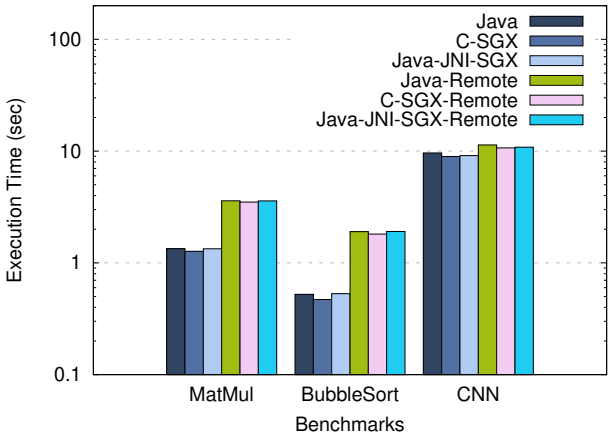


Fig. 5: Performance comparison of the different Andromeda-enabled execution methods, including offloading, against the vanilla Java versions.

particular, we compare the execution of the vanilla Java implementation against their secure implementation using C and SGX natively, compiled with our custom cross-compiler, and their implementation using the Andromeda Java API for SGX. These benchmarks consist of some typical operations that external devices or wearables may perform on sensitive data (e.g., for analytics on finance or health data, image processing, etc.) and also exhibit different performance characteristics (i.e., IO-bound, memory-intensive, computational-intensive). The first benchmark performs matrix multiplication on two tables with 10K rows and columns. The second benchmark performs bubble sort on an array of 20K random integers. Finally, the third benchmark is a convolutional neural network that performs image classification using as input images with size 800x600 pixels, generated by an external device.

As we can see in Figure 5, the vanilla Java implementation requires 5.4% to 11.2% more time to finish its execution than the respective SGX-enabled implementations (developed either in native C or using the Andromeda SGX Java API) whereas the time needed for code offloading ranges from 5.13% to 7.5% for Java. The reason for this is that in both SGX-enabled versions, the functions are executed natively using C. The overhead introduced by the I/O with the secure enclave, the JNI layer (in the SGX-enabled Java implementation) and the data offloading on the socket level are minimal in these cases and does not overshadow the speedup gained by the native execution.

8 Discussion and Limitations

Misusing Andromeda Keystore for encrypting/decrypting messages Intel SGX cannot verify whether a request for an operation has been received

from a benign or a malicious user. As a result, an attacker who has managed to gain access to the base Android system or the Keystore service could leverage Intel SGX to encrypt and decrypt messages. Still, the adversary cannot steal any key stored in secure enclaves.

Denial-of-Service Adversaries who have compromised the Android system can easily disrupt the operation of Intel SGX. For example, they can delete or modify input or output data by hooking the functions that communicate with the SGX application or even kill or suspend the execution of enclaves. As the main purpose of Intel SGX is to protect sensitive data and perform trusted operations, defending against these attacks is out of the scope of this work.

Portability Andromeda is currently implemented on Intel SGX-equipped CPUs. Even though this prevents us from adopting it to other CPU models, we note that Andromeda is not fundamentally bound to Intel SGX; instead our proposed architecture could be implemented on top of other approaches that offer secure user-level enclaves. For instance, there are recently proposed approaches that implement user-level secure enclaves, similar to SGX, either independent of the underlying CPU [27] either on top of ARM TrustZone [20]; porting Andromeda to these approaches is part of our future work.

9 Related Work

ARM TrustZone [14] enables the development of two separate environments, the trusted and the untrasted world. This split enables the execution of the rich OS (that runs in the untrusted world) and the system software that controls basic operations that must be protected and runs in the trusted world. Santos et al [33] use TrustZone for securing mobile applications, by establishing and isolating trusted components. However, their approach requires a trusted language runtime in the TCB, due to the fact that there is only a single trusted world. DroidVault [29] presents a security solution for storing and manipulating sensitive data. The data are stored in an encrypted form on the filesystem and are only processed (decrypted) in TrustZone. TZ-RKP implements a low-TCB system level safe security monitor on top of the TrustZone architecture [16] that provides a real-time OS kernel protection. The monitor routes privileged system functions through secure world for examination. Samsung KNOX [32] is a secure container framework, leveraging ARM TrustZone, that offers protection from both the software and the hardware. However, KNOX is primarily a closed-source system and its architecture is not well documented in the open literature. A major limitation of all these TrustZone-based approaches is that they do not protect against attackers with physical DRAM access. Moreover, TrustZone is not best suited to be securely shared by multiple applications, as there is only one shared TEE provided by the hardware, offering limited isolation granularity compared to SGX. This prevents it from being leveraged simultaneously by

different applications, either in user-space (e.g., banking applications, etc.) or kernel-space (security monitors, device keystore, etc.).

Intel SGX [8] offers fine-grained confidentiality and integrity at the enclave level. Haven [18] aims to execute unmodified legacy Windows applications inside SGX enclaves by porting a Windows library OS into SGX. TrustAV [25] offloads malware analysis operations within secure enclaves to shield the transfer and processing of private user data in untrusted environments. Graphene-SGX [37] encapsulates the entire libOS, including the unmodified application binary, supporting libraries, and a trusted runtime with a customized C library and ELF loader inside an SGX enclave. VC3 [34] uses SGX to achieve confidentiality and integrity for the Map Reduce framework. SCONE [15] is a shielded execution framework that enables developers to compile their C applications into Docker containers. SGX-Mon [24] is a host-based kernel integrity monitor that resides in SGX enclaves to prevent attackers from tampering its execution and operation-critical data. In contrast with these works, Andromeda is the first approach, to the best of our knowledge, that enables SGX enclaves for the Android OS. Moreover, there are recently proposed approaches that implement user-level enclaves, similar to SGX, either independent of the underlying CPU [27] or on top of ARM TrustZone [20]; Andromeda is not fundamentally tight to Intel SGX and, as such, could be implemented on top of such approaches instead.

Finally, several improvements for SGX have been recently developed in order to protect against memory bugs [28,35,30] or controlled-channel attacks [36]. SGXBOUNDS [28] enables bounds-checking with low memory overheads, in order to fit within the limited EPC size. SGX-Shield [35] implements Address Space Layout Randomization (AS-LR) in enclaves, with a scheme to maximize the entropy, and the ability to hide and enforce ASLR decisions. Eleos [30] proposes to reduce the number of enclave exits by asynchronously servicing system calls outside of the enclaves, and enabling user-space memory paging. T-SGX [36] is an approach that combines SGX with Transactional Synchronization Extensions, in order to mitigate controlled-channel attacks. All these works are orthogonal to our approach and can be integrated to Andromeda.

10 Conclusion

In this work, we present the design, implementation, and evaluation of Andromeda, a framework that provides the first SGX interface for Android OS. Using Andromeda, developers can explicitly use SGX for their applications via the native API in C/C++ or the Java interface that provides access to the secure enclaves through JNI bindings. Also, Andromeda offers services that enhance Android's security and provides protection schemes for applications that deal with sensitive data.

As part of our future work, we plan to port Andromeda to SGX-compliant approaches that do not depend on specific CPU models though, either using software-only techniques [27], either on top of ARM TrustZone [20]. Also, we plan to enhance our secure pairing mechanism by utilizing protocols that offer

mutually trusted secure communication channels between enclaves that reside in different physical devices, similar to [22].

Acknowledgments

The research work was supported by the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Technology (GSRT), under the HFRI PhD Fellowship grant (GA. No. 2767). This work was also supported by the projects CONCORDIA, C4IIoT and COLLABS, funded by the European Commission under Grant Agreements No. 830927, No. 833828, and No. 871518. This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

References

1. Amazon's AWS permission managements. https://aws.amazon.com/iam/details/manage-permissions/
2. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/amd-secure-memory-encryption-sme-amd-secure-encrypted-virtualization-sev/
3. Android Keystore. https://developer.android.com/training/articles/keystore.html
4. Android Sensor API. https://developer.android.com/guide/topics/sensors/sensors_overview
5. Android Services. https://developer.android.com/guide/components/services.html
6. Bosch IOT. https://www.bosch-iot-suite.com/permissions/
7. Crystax NDK. https://www.crystax.net/android/ndk/
8. Intel Software Guard Extensions (SGX). https://software.intel.com/en-us/sgx
9. Intel's Skylake Processors. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-introduction-basics-paper.pdf
10. International Data Corporation. https://www.idc.com/promo/smartphone-market-share/os
11. Mobile Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide
12. Samsung SmartThings. https://www.samsung.com/us/smart-home/smartthings/
13. Statista. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/
14. ARM LIMITED: ARM Security Technology - Building a Secure System using TrustZone Technology (2009)
15. Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M.L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., Fetzer, C.: SCONE: Secure Linux Containers with Intel SGX. OSDI (2016)
16. Azab, A.M., Ning, P., Shah, J., Chen, Q., Bhutkar, R., Ganesh, G., Ma, J., Shen, W.: Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. CCS (2014)
17. Azab, A.M., Ning, P., Wang, Z., Jiang, X., Zhang, X., Skalsky, N.C.: Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. CCS (2010)

18. Baumann, A., Peinado, M., Hunt, G.: Shielding Applications from an Untrusted Cloud with Haven. ACM Trans. Comput. Syst. **33**(3), 8:1–8:26 (Aug 2015)
19. Boivie, R., Williams, P.: Secureblue++: Cpu support for secure execution. Technical report (2012)
20. Brasser, F., Gens, D., Jauernig, P., Sadeghi, A.R., Stapf, E.: Sanctuary: Arming trustzone with user-space enclaves (2019)
21. Caddy Tom: Side-channel attacks. https://link.springer.com/referencework/10.1007%2F0-387-23483-7 (2011)
22. Chalkiadakis, N., Deyannis, D., Karnikis, D., Vasiliadis, G., Ioannidis, S.: The Million Dollar Handshake: Secure and Attested Communications in the Cloud. CLOUD (2020)
23. Colp, P., Zhang, J., Gleeson, J., Suneja, S., de Lara, E., Raj, H., Saroiu, S., Wolman, A.: Protecting Data on Smartphones and Tablets from Memory Attacks. ASPLOS (2015)
24. Deyannis, D., Karnikis, D., Vasiliadis, G., Ioannidis, S.: An Enclave Assisted Snapshot-Based Kernel Integrity Monitor. EdgeSys (2020)
25. Deyannis, D., Papadogiannaki, E., Kalivianakis, G., Vasiliadis, G., Ioannidis, S.: TrustAV: Practical and Privacy Preserving Malware Analysis in the Cloud. CODASPY (2020)
26. Fernandes, E., Paupore, J., Rahmati, A., Simionato, D., Conti, M., Prakash, A.: FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In: Proceedings of the 25th USENIX Security Symposium. USENIX Security (2016)
27. Ferraiuolo, A., Baumann, A., Hawblitzel, C., Parno, B.: Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. SOSP (2017)
28. Kuvaiskii, D., Oleksenko, O., Arnautov, S., Trach, B., Bhatotia, P., Felber, P., Fetzer, C.: SGXBOUNDS: Memory Safety for Shielded Execution. In: Proceedings of the Twelfth European Conference on Computer Systems. EuroSys (2017)
29. Li, X., Hu, H., Bai, G., Jia, Y., Liang, Z., Saxena, P.: DroidVault: A Trusted Data Vault for Android Devices. ICECCS (2014)
30. Orenbach, M., Lifshits, P., Minkin, M., Silberstein, M.: Eleos: ExitLess OS Services for SGX Enclaves. EuroSys (2017)
31. Pirker, M., Slamanig, D.: A framework for privacy-preserving mobile payment on security enhanced arm trustzone platforms. TrustCom (2012)
32. Samsung: White Paper : An Overview of Samsung KNOX. http://www.samsung.com/my/business-images/resource/white-paper/2013/11/Samsung_KNOX_whitepaper_An_Overview_of_Samsung_KNOX-0.pdf (2013)
33. Santos, N., Raj, H., Saroiu, S., Wolman, A.: Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. ASPLOS (2014)
34. Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., Russinovich, M.: VC3: Trustworthy Data Analytics in the Cloud Using SGX. In: Proceedings of the 2015 IEEE Symposium on Security and Privacy. S&P (2015)
35. Seo, J., Lee, B., Kim, S., Shih, M.W., Shin, I., Han, D., Kim, T.: SGX-Shield: Enabling address space layout randomization for SGX programs. NDSS (2017)
36. Shih, M.W., Lee, S., Kim, T., Peinado, M.: T-SGX: Eradicating controlled-channel attacks against enclave programs. NDSS (2017)
37. Tsai, C.C., Porter, D.E., Vij, M.: Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. USENIX ATC (2017)
38. Wang, J., Stavrou, A., Ghosh, A.: Hypercheck: A hardware-assisted integrity monitor. In: Recent Advances in Intrusion Detection. pp. 158–177. Springer (2010)
39. Xianyi Zheng, Lulu Yang, Jiangang Ma, Gang Shi, Dan Meng: TrustPAY: Trusted mobile payment on security enhanced ARM TrustZone platforms. ISCC (2016)