Trustworthy Execution on Mobile Devices: What security properties can my mobile platform give me?

Amit Vasudevan, Emmanuel Owusu, Zongwei Zhou

James Newsome, and Jonathan McCune

November 16, 2011

CMU-CyLab-11-023

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Trustworthy Execution on Mobile Devices:
What security properties can my mobile platform give *me*?

Amit Vasudevan, Emmanuel Owusu, Zongwei Zhou, James Newsome, and Jonathan McCune

Abstract

We are now in the post-PC era, yet our mobile devices are insecure. We consider the different stake-holders in today's mobile device ecosystem, and analyze why widely-deployed hardware security primitives on mobile device platforms are inaccessible to application developers and end-users. We systematize existing proposals for leveraging such primitives, and show that they can indeed strengthen the security properties available to applications and users, all without reducing the properties currently enjoyed by OEMs and network carriers. We also highlight shortcomings of existing proposals and make recommendations for future research that may yield practical, deployable results.

1 Introduction

We are putting ever more trust in mobile devices. We use them for e-commerce and banking, whether through a web browser or specialized *apps*. Such apps hold high-value credentials and process sensitive data that need to be protected.

Meanwhile, mobile phone OSes are untrustworthy. While in principle they attempt to be more secure than desktop OSes (e.g., by preventing modified OSes from booting, by using safer languages, or by sandboxing mechanisms for third-party apps such as capabilities), in practice they are still fraught with vulnerabilities.

Mobile OSes are as complex as desktop OSes. Isolation and sandboxing provided by the OS is routinely broken, c.f. Apple iOS jail-breaking by *clicking a button on a web page* [15, 51]. Mobile OSes often share code with open-source OSes such as GNU/Linux, but often lag behind in applying security fixes, meaning that attackers need only look at recent patches to the open-source code to find vulnerabilities in the mobile device's code. Therefore, there is a need for isolation and security primitives exposed to application developers in such a way that they need not trust the host OS.

We argue that this problem is severe enough to have garnered significant attention outside of the security community. Demand for mobile applications with stronger security requirements has given rise to add-on hardware with stronger security properties (§2). This situation is unfortunate, given that many current mobile devices already have hardware support for isolated execution environments and other security features. However, these features are not made available to all parties who may benefit from their presence.

Today's mobile device hardware and software ecosystem consists of multiple *stake-holders*, primarily comprising the OEM (handset manufacturer), telecommunications provider or carrier, application developers, and the device's owner (the human user). Carriers typically also serve in the role of platform integrator, customizing an OEM's handset with additional features and branding (typically via firmware or custom apps). To date, security properties desirable from the perspectives of application developers and users have been secondary concerns to the OEMs and carriers [13, 38, 54].

The historically closed partnerships between OEMs and carriers have lead to a monolithic trust model within today's fielded hardware security primitives. Everything "inside" is assumed to be trustworthy, i.e., the software modules executing in the isolated environment often reside in each other's trusted computing base (TCB). As long as this situation persists, OEMs and carriers will not allow third-party code to leverage these features. Only in a few cases, where the OEM has partnered with a third party, are these features used to protect the *user's* data (c.f. §2, Google Wallet).

We approach this scenario optimistically, and argue that there is room to meet the needs of application developers and users while adding negligible cost. We thus define the principal challenge for the technical community: **to present sound technical evidence that application developers and users can simultaneously benefit from hardware security features without detracting from the security properties required for the OEMs and carriers.**[1] Our goal in this paper is to systematize deployed (or readily available) hardware security features, and to provide an extensive and realistic evaluation of existing (largely academic) proposals for multiplexing these features amongst *all* stake-holders.

We proceed in §3 by defining a set of security features that may be useful for application developers that need to process sensitive data. Our focus is on protecting secrets belonging to the *user*, such as credentials used to authenticate to online services and locally cached data.

We next provide an overview of hardware security features available on today's mobile platforms (§4). We show that hardware security features that can provide the desired properties to application developers are prevalent, but they are typically not accessible in COTS devices' default configurations.

[1] We wish to distinguish this challenge from proposals that OEMs increase their hardware costs by including additional hardware security features that are exclusively of interest to application developers and users. Our intention in this paper is to emphasize practicality, and thus define such proposals to be out of scope.

We then move on to evaluate existing proposals (given the hardware security features available on mobile devices) for creating a trustworthy execution environment that is able to safely run sensitive applications that are potentially considered untrustworthy by other stake-holders (§5). We show that multiplexing these secure execution environments for mutually-distrusting sensitive applications is quite possible if the threat model for application developers and users is primarily software-based attacks (§6).

Finally (§7), we provide an end-to-end analysis and recommendations for the current best practices for making the most of mobile hardware-based security features, from the points of view of each stake-holder. Unfortunately, without firmware or software changes by OEMs and carriers, individual application developers today have little opportunity to leverage the hardware security primitives in today's mobile platforms. The only real options are either to partner with a mobile platform integrator, to distribute a customized peripheral (e.g., a smart-card-like device that can integrate with a phone, such as a storage card with additional functionality), or to purchase unlocked development hardware. We provide recommendations for OEMs and carriers for how they can make hardware-based security capabilities more readily accessible to application developers without compromising the security of their existing uses.

2 Demand for Applications Requiring Hardware Security

Does providing third-party developers with access to hardware-supported security features make sense for the OEMs or carriers? This is an important consideration for an industry where a few cents in cost savings can be the deciding factor for features. We show that there are many applications on mobile devices that require strong security features, and that must currently work-around the lack of those features. Being forced to deal with these work-arounds stifles the market for security-sensitive mobile applications, and endangers the security of the applications that are deployed anyways.

Google Wallet[2] allows consumers to use their mobile phones as a virtual wallet. The application stores users' payment credentials locally, which are then used to make transactions via near field communication (NFC) with point-of-sale (POS) devices. To store the users' credentials securely, Wallet relies on a co-processor called a Secure Element (SE) which provides isolated execution (§3.1), secure storage (§3.2), and a trusted path (§3.5) to the on-board NFC radio. Unfortunately, the SE only runs code that is signed by the device manufacturer. This may be because the SE lacks the ability to isolate authorized modules from each-other, or it may simply be considered a waste of time. As a result, developers without Google's clout will not be able to leverage these capabilities for their own applications.

There is evidence that Apple has similar plans for its products; they recently published a patent for an embedded SE with space allocated for both a Universal Subscriber Identity Module (USIM) application and "other" applications [50].

Services such as Square and GoPay allows merchants to complete credit card transactions with their mobile device using an application and a magnetic stripe reader [39]. While Square's security policies[3] indicate that they do not store credit card data on the mobile device, the data does not appear to be adequately protected when it passes through the mobile device. Researchers have verified that the stripe reader does not protect the secrecy or integrity of the read-data [42]. This implies that malware on the mobile device could likely eavesdrop on credit-card data for swiped cards or inject stolen credit-card information to make a purchase [42].

These applications could benefit greatly from the hardware-backed security features we describe in §3. A trusted path (§3.5) could enforce that the intended client application has exclusive access to the audio port (with which the card readers interface), thus protecting the secrecy and integrity of that data from malware. They could also benefit greatly from a remote attestation mechanism (§3.3), which the servers could use to ensure that received-data is actually from the authorized client-application, and that it used a trusted-path to the reader, thus helping to ensure that the physical credit card was actually present.

Companies have attempted to fill the gap left behind by the lack of developer-accessible hardware security features on mobile devices, by implementing removable Secure Elements (SEs). Removable Secure Elements (also known as Independent Secure Elements or Secure Memory Cards) are SEs interfaced to removable memory such as a Universal Integrated Circuit Card (UICC) or Secure Digital (SD) Card [46].

Secure memory cards allow third-party developers to develop applications for one Secure Element interface instead of having to account for the specific interface requirements of various handset manufacturers. Since secure memory cards are removable, consumers can easily move credentials to other handsets or devices. On the other hand, removable SEs may be more vulnerable to physical attack (e.g., the SE may be more easily lost, stolen, or corrupted).

UICC is a general-purpose platform for smart-card applications. UICC is capable of hosting applications for the card issuer—such as USIM for voice and data access—in addition to hosting non-telecommunications applications including mobile payments and ticketing. As a multi-tenant SE, the UICC's operating system manages memory access for multiple mutually-distrusting applications [30].

Giesecke & Devrient offer a 2 GB microSD card coupled with a Secure Element—the Mobile Security Card. The SE supports cryptographic functions including SHA-

[2] http://www.google.com/wallet/how-it-works-security.html

[3] https://squareup.com/security

256, DES/3-DES, AES, and RSA.[4] This functionality enables security-sensitive applications such as disk encryption, single sign-on, building access control, and PKI key management. Tyfone offers a similar product called SideSafe.[5]

The number of new applications requiring hardware security features is evidence that there is demand for hardware-backed security primitives among third-party businesses and application developers. Unfortunately, some of the workarounds may actually increase the attack surface for fraud [42]. Several third parties have stepped in to provide hardware-backed security features in the form of removable Secure Elements. OEMs could provide a more tightly integrated experience for developers, and avoid potential security vulnerabilities by opening up pre-existing hardware security primitives to application developers.

3 Desired Security Features

Here we describe a set of features intended to enable secure execution on mobile devices. This can be interpreted as the wish-list for a security-conscious application developer. The strength of these features can typically be measured by the size, complexity, and attack surface of the components that must be relied upon for a given security property to hold. This is often referred to as the *trusted computing base* (TCB).

On many systems, the OS provides security-relevant APIs for application developers. However, this places the OS in the TCB, meaning that a compromised OS voids the relevant security properties. We briefly discuss whether and how the security features below are provided on today's mobile platforms, and some strategies for providing these properties to applications without including the OS in the TCB.

3.1 Isolated Execution

Isolated execution gives the application developer the ability to run a software module in complete isolation from other code. It provides secrecy and integrity of that module's code and data at *run-time*. Today's mobile OSes provide process-based isolation to protect applications' address spaces and other system resources. However, these mechanisms are circumventable when the OS itself is compromised.

To provide isolated execution that does not depend on the operating system, some alternative execution environment not under control of the OS is required. Such an environment could be provided by a layer running under the OS on the same hardware (i.e., a hypervisor), or in a parallel environment (such as a separate coprocessor). We examine some candidate isolated execution environments and their suitability for mobile platforms in §5.

Regarding today's mobile platforms, the Meego Linux distribution for mobile devices does include provisions for isolated execution. Meego's Mobile Simplified Security Framework (MSSF) implements a trusted execution environment (TrEE) that is protected from the OS [34]. However, this environment is not open to third party developers.

3.2 Secure Storage

Secure storage provides secrecy, integrity, and/or freshness for a software module's data *at rest* (primarily when the device is powered off, but also under certain conditions based upon which software has loaded). The most common example demonstrating the need for secure storage is access credentials, such as a cached password or a private asymmetric key. Other examples include sensitive information cached for offline consumption, such as bills or medical information.

Most mobile OSes provide this property at least using file system permissions, which are enforced by the operating system. File system permissions alone can be circumvented by compromising the OS itself. They can also be circumvented by offline attacks such as by loading an alternative OS that does not respect those permissions, or by removing the storage media and accessing it directly.

A stronger form of secure storage can be built using a storage location that is physically protected, and with access control implemented independently of the OS. E.g., on the PC, the TPM has a small amount of on-board NVRAM for this purpose. A physically protected piece of secure storage used in this way is called a *root of trust for storage*, or RTS.

A root of trust for storage can be used to bootstrap a larger secure storage mechanism, using *sealed storage*. The sealed storage primitive uses a key protected by the RTS to encrypt the given data, and to protect the authenticity of that data and of attached meta-data. The metadata includes an access-control-policy for which code is authorized to request decryption (e.g., represented as a hash over the code), and potentially other data such as which software module sealed the data in the first place. Sealed data (ciphertext) can then be stored on an unprotected storage device.

Extra steps are required to provide *state continuity* for sealed data on untrusted storage; otherwise the sealed data may be undetectably rolled back to an older version. This could be a problem, e.g., if the sealed data is an access-control list or a revocation list. Freshness protection can be implemented using trusted counters, a small piece of protected storage for a counter or hash, or a trusted time source [44].

Symbian and Meego make use of protected memory and sealed storage [34]. Symbian's installer system distinguishes between removable and permanently-installed storage, and it calculates a hash over any applications installed to removable media, and stores that hash in permanently-installed storage. Applications executed from removable media are subsequently integrity-checked using the relevant hash.

MSSF uses keys kept in its Trusted Execution Environment (TrEE) (§3.1) to protect the integrity of application binaries, and to provide a sealed storage facility, which *is* available to third party developers [34]. While this offers protection

[4] http://www.gd-sfs.com/the-mobile-security-card/mobile-security-card-se-1-0/

[5] http://tyfone.com

against offline attacks, since third party applications are not allowed to execute in the TrEE, data protected by this mechanism is vulnerable to online attacks via a compromised OS.

Recent versions of iOS combine a user-secret with a protected device-key to implement secure storage [5]. However, the device-key does not appear to be access-controlled by code identity, meaning that an attacker can defeat this mechanism if he is able to obtain the user secret, e.g., via malware, or via performing an online brute-force attack [22, 32].

Android offers an AccountManager API [2]. The model used by this API supports code modules that perform operations on the stored credential rather than releasing them directly, which would make it amenable to a model with sealed storage and isolated execution. Unfortunately, it appears that the data is currently stored in plaintext, and can be retrieved via direct access to the storage device or by compromising the operating system [1, 59].

Android began offering file-system encryption in version 3.0 [3]. However, this feature is protected only by a user-secret entered at boot time, meaning that it can be circumvented by compromising the operating system at runtime, or by brute-forcing the user-secret in an offline attack. Android 4.0 will also support a new keychain API [4]. The details of this API and of how the data is protected are not yet available.

3.3 Remote Attestation

Remote attestation allows remote parties to verify that a particular message originated from a particular software module. Remote attestation is useful in cases where a remote service wishes to ensure that it is communicating with a known client, and not with malware. For attestation to be meaningful, it must attest to the entire TCB of the given application. For an application running on a normal OS, the attestation would necessarily include a measurement of the OS kernel, which is part of that TCB, and of the application itself. A remote party, such as an online banking service, could use this information, if it knew a list of valid OS kernel identities and a list of valid client banking-app identities, to ensure that the system had booted a known-good kernel, and that the OS had launched a known-good version of the client banking app.

Remote attestations are more meaningful when the TCB is relatively small and stable. In the example of a banking application, if a critical component of the app ran as a module in an isolated execution environment with a remote-attestation capability, then the attestation would only need to include a measurement of the smaller isolated execution environment code, and of the given module. Not only would it be easier to keep track of a list of known-good images (assuming that the isolated execution environment's code is relatively stable), but the attestation would be more meaningful because the isolated execution environment is presumed to be less susceptible to run-time compromise. This is important because the attestation only tells the verifier what code was *loaded*; it would not detect if a run-time exploit overwrote that code with unauthorized code.

Attestation mechanisms are typically built using a private key that is only accessible by a small TCB (§3.1) and kept in secure storage (§3.2). A certificate issued by a trusted party, such as the device manufacturer, certifies that the corresponding public key belongs to the device. One or more platform configuration registers store measurements of loaded code. The private key can then be used to generate signed attestations about its state or the state of the rest of the system.

It can be useful to contrast an attestation scheme to a *secure boot* scheme. Secure boot is the process of performing integrity checks (e.g., verifying a cryptographic hash or digital signature) on each stage of the boot process, and halting if any stage fails its check. A fully booted device is thus implicitly believed to be in an approved configuration. Attestation separates the process of measuring (performing a cryptographic hash) each stage of execution from the process of evaluating whether a set of measurements represents a valid configuration. Especially when there are multiple stake-holders, secure boot does not scale all the way to individual third-party applications. Remote attestation can convey meaningful information under such conditions, because individual attestations can be sent to the relevant stake-holders for evaluation.

Some forms of remote attestation are implemented and used on today's mobile platforms [34]. However, as far as we know, no such mechanisms are made available to arbitrary third-party developers.

3.4 Secure Provisioning

Secure provisioning is a mechanism to send data to a *specific software module*, running on a *specific device*, while protecting that data's secrecy and integrity. This is useful for migrating data between a user's devices. For example, a user may have a credential database that he wishes to migrate or synchronize across devices while ensuring that only the corresponding credential-application running on the intended destination device will be able to access that data.

One way to build a secure provisioning mechanism is to use remote attestation (§3.3) to attest that a public encryption key belongs to a particular software module running on a particular device. The sender can then use that key to protect data to be sent to the target software module on the target device.

Some of today's mobile platforms implement mechanisms to authenticate external information from the hardware stakeholders (e.g., software updates), with the hash of the public portion of the signing key stored immutably on the device [34]. Other secure provisioning mechanisms are likely implemented and used by device manufacturers to implement features such as digital rights management. As far as we know, however, secure provisioning mechanisms are not available for direct use by arbitrary third-party developers on mobile platforms.

3.5 Trusted Path

Trusted path protects authenticity, and optionally secrecy and availability, of communication between a software module and a peripheral (e.g., keyboard or touchscreen) [24, 31, 37, 55]. When used with human-interface devices, this property allows a human user to ascertain precisely the application with which she is currently interacting. With full trusted path support, malicious applications that attempt to spoof legitimate applications by creating identical-looking user interfaces will conceivably become ineffective. While human factors abound in designing the precise UI elements [17, 49], the technical underpinnings that enable any such architecture remain a significant challenge.

Trusted path to sensors and actuators can be another useful feature. For example, trusted paths to sensors can be used to facilitate "citizen-journalism" applications, where a software module uses trusted path to ensure that it is receiving unaltered sensor input, and then uses remote attestation mechanisms to attest to the accuracy of the sensed data [23, 48].

Building secure trusted paths is a challenging problem. In principle, many mobile platforms support a form of trusted path, but the TCB is relatively large and untrustworthy. For example, the *Home* button on iOS and Android devices constitutes a *secure attention sequence* that by design uncircumventably transfers control of the user interface to the OS's "Home" screen. Once there, the user can transfer control to the desired application.

However, the TCB for such mechanisms includes the entire OS. For the Android Home button, the TCB also includes third-party apps that the user installed with the "launcher" capability. An app that the user installs with the launcher capability can replace the Home screen with its own arbitrary interface (potentially impersonating the previous Home screen if it didn't advertise itself as a custom Home screen). It is then free to impersonate other apps by spoofing the UI of the requested app instead of launching the requested app.

The OS can be removed from the TCB of such trusted paths by preventing the OS from communicating directly with the device and running the device driver in an isolated environment. This requires the platform to support a low-level access-control policy for access to peripherals. ARM's TrustZone extensions facilitate this type of isolation (§4.2.1).

4 Available Hardware Primitives

In this section we discuss currently-available hardware security primitives with a focus on existing smartphone and tablet platforms. As the vast majority of these platforms are built for the ARM architecture, we first present a generic ARM platform hardware and security architecture, focusing our discussion on platform hardware components that help realize the features discussed in §3. We then identify design gaps and implementation challenges in off-the-shelf mobile devices that prevent third-party application developers from fully realizing the desired security properties. Finally, we provide two case studies of inexpensive mobile *development* platforms with myriad security features, to serve as references against which to compare mass-market devices.

4.1 ARM Platform: Hardware and Security Architecture

ARM's platform architecture comprises the Advanced Microcontroller Bus Architecture (AMBA) and different types of interconnects, controllers and peripherals. ARM calls these the "CoreLink", which has four major components (Figure 1).

- *Network interconnects* are the low-level physical on-chip interconnection primitives that bind various system components together. These include switches, bridges, and routing fabric. AMBA defines two basic types of interconnects: (i) the Advanced eXtensible Interface (AXI) – a high performance master and slave interconnect interface, and (ii) the Advanced Peripheral Bus (APB)—a low-bandwidth interface to peripherals.
- *Memory controllers* correspond to the predominant memory types: (i) static memory controllers (SMC) interfaced with SRAM, and (ii) dynamic memory controllers (DMC) interfaced with DRAM.
- *System controllers* include the: (i) Generic interrupt controller (GIC)—for managing device interrupts, (ii) DMA controllers (DMAC)—for direct memory access by peripheral devices, and (iii) TrustZone Address Space Controller (TZASC) and TrustZone Memory Adapter (TZMA)—for partitioning memory between multiple "worlds" in a split-world architecture (§4.2.1).
- *System peripherals* include LCDs, timers, UARTs, GPIO pins, etc. These peripherals can be further assigned to specific "worlds").

We now proceed to discuss the above components in the context of each of the security features described in §3.

4.2 Isolated Execution

Multiple hardware primitives exist for isolated execution on ARM architecture devices today. ARM first introduced their TrustZone Security Extensions in 2003 [6], enabling a "two-world" model, whereby both secure and non-secure software can coexist on the same processor. Today, TrustZone features are available for many system components beyond just the CPU(s), as we discuss below.

ARM recently announced hardware support for virtualization for their Cortex A15 CPU family [11]. These extensions enable more traditional virtualization solutions in the form of hypervisors or virtual machine monitors [45].

4.2.1 Split-World-based Isolated Execution

ARM's TrustZone Security Extensions [7] enable a single physical processor core to safely and efficiently execute code in two "worlds"—the *secure world* for security sensitive application code and the *normal world* for non-secure applica-

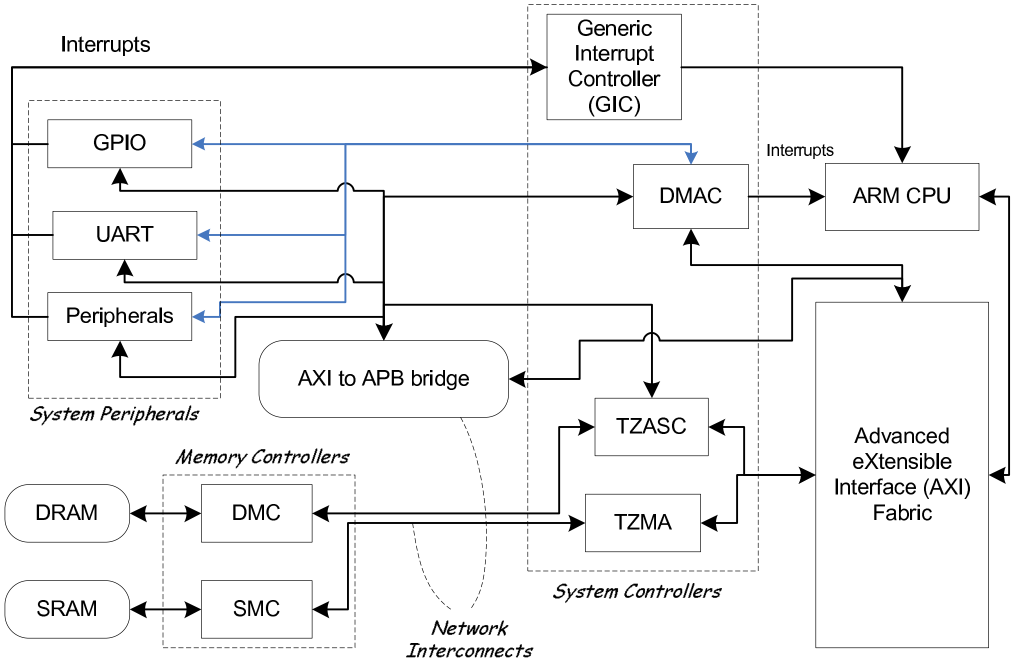

Figure 1: Generic ARM platform hardware and security architecture.

tions (Figure 2). CPU state is banked between both worlds; the secure-world can access all normal-world state, but not vice-versa. A new processor mode, called the *monitor mode*, supports context switching between the secure-world and the normal-world and can be entered either asynchronously (e.g., as a result of hardware interrupts or exceptions) or synchronously by the execution of the Secure Monitor Call (SMC) instruction. Note that the SMC instruction can only be executed from the supervisor mode (SVC) in the normal-world. The monitor mode software is responsible for context-switching CPU state that is not automatically banked.

Memory Isolation. ARM's TrustZone Security Extensions split CPU state into two distinct worlds, but they alone cannot partition memory between the two worlds. Memory isolation is achieved using a combination of TrustZone-aware Memory Management Units (MMU), TrustZone Address Space Controllers (TZASC), TrustZone Memory Adapters (TZMA), and Tightly Coupled Memory (TCM).

A TrustZone-aware MMU provides a distinct MMU interface for each processor world, enabling each world to have a local set of virtual-to-physical memory address translation tables. The translation tables have protection mechanisms which prevent the normal-world from accessing secure-world memory. Such MMUs employ tagged Translation Look-aside Buffers (TLB), where entries are tagged with the identity of the world. This enables secure- and normal-world entries to co-exist so as to improve performance [7].

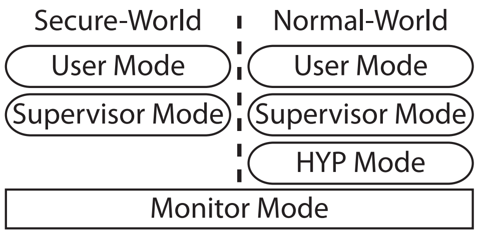

Figure 2: ARM Isolated Execution Hardware Primitives. Split-world-based isolation enables both secure and normal processor worlds. Virtualization-based isolation adds a higher-privileged layer for a hypervisor in the normal world.

The TZASC interfaces devices such as Dynamic Memory Controllers (DMC) to partition DRAM into distinct memory regions. The TZASC has a secure-world-only programming interface that can be used to designate a given memory region as secure or normal. The TZASC rejects memory transactions from the normal-world that are directed towards secure memory regions. The TZMA provides similar functionality for off-chip ROM or SRAM. With a TZMA, ROM or SRAM can be partitioned between the two worlds.

Tightly Coupled Memory (TCM) is memory that is in the same physical package as the CPU, so that physical tampering with the external pins of an integrated circuit will be ineffective in trying to learn the information stored in TCM. TCMs are typically blocks of fast on-chip SRAM that exist at the same level as the CPU's L1 cache subsystem. Secure-world software is responsible for configuring access permissions (secure vs. normal) for a given TCM block.

Peripheral isolation. Peripherals in the ARM platform architecture can be designated as *secure* or *normal*. Secure peripherals are intended to be accessible by the secure world while normal peripherals can be accessed from both worlds. Thus, there is a need to isolate secure and normal peripherals so that software running in the normal world cannot maliciously or inadvertently address secure-world peripherals.

ARM's "CoreLink" architecture connects high-speed system devices such as the CPU and memory controllers using the Advanced eXtensible Interface (AXI) bus [9]. The rest of the system peripherals are typically connected using the Advanced Peripheral Bus (APB). The AXI-to-APB bridge device is responsible for interfacing the APB interconnects with the AXI fabric. The AXI bus transaction packets include an identification field that designates the transaction as secure or normal. However, the APB transactions do not have such a provision [10]. This places the responsibility for managing security-relevant state with the AXI-to-APB bridge.

A TrustZone-aware AXI-to-APB bridge contains address decode logic that selects the desired peripheral based on the security state of the incoming AXI transaction; the bridge rejects normal-world transactions to peripherals designated to be used by the secure-world. A TrustZone AXI-to-APB bridge can include an optional software programming interface that allows dynamic switching of the security state of a given peripheral. This can be used for sharing a peripheral between both the secure and normal worlds.

DMA Protection. Certain peripherals (e.g., LCD controllers and storage controllers) can transfer data to and from memory using Direct Memory Access (DMA), which is not access-controlled by the AXI-to-APB bridge. A TrustZone-aware DMA controller (DMAC) supports concurrent secure and normal peripheral DMA accesses, each with independent interrupt events. Together with the TZASC, TZMA, GIC, and the AXI-to-APB bridge, the DMAC can prevent a peripheral assigned to the normal-world from performing a DMA transfer to or from secure-world memory regions.

Hardware Interrupt Isolation. As peripherals can be assigned to either the secure or normal world, there is a need to provide basic interrupt isolation so that interrupts from secure peripherals are always handled in secure world.

Hardware interrupts on the current ARM platforms can be categorized into: IRQ (normal interrupt request) and FIQ (fast interrupt request). The Generic Interrupt Controller (GIC) can configure interrupt lines as secure or normal and enables secure-world software (in monitor mode) to selectively trap such system hardware interrupts. This enables flexible interrupt partitioning models. For example, IRQs can be assigned for normal-world operations and FIQs for secure-world operations. The CPU core provides support for interrupt identification and redirection. For example, if an IRQ occurs during normal-world execution, it is handed over to the normal-world interrupt handler immediately. However, if an IRQ occurs during secure-world execution, the monitor-mode handler is invoked which can choose to handle the IRQ or inject it back to the normal-world. The GIC hardware also includes logic to prevent normal-world software from modifying secure interrupt line configurations. Thus, secure world code and data can be protected from potentially malicious normal-world interrupt handlers, but TrustZone by itself is not sufficient to implement device virtualization.

4.2.2 Virtualization-based Isolated Execution

ARM's Virtualization Extensions provide hardware virtualization support to normal-world software starting with the Cortex A15 CPU family [11]. The basic model for a virtualized system involves a hypervisor, that runs in a new normal-world mode called Hyp mode (Figure 2). The hypervisor is responsible for multiplexing guest OSes, which run in the normal world's traditional OS and user modes. Note that software using the secure world is unchanged by this model, as the hypervisor has no access to secure world state. The hypervisor can optionally trap any calls from a guest OS to the secure world. As hardware-supported virtualization architectures have been studied for over four decades [45], we elide further detail on the ARM specifics.

4.3 Secure Storage

Current ARM platform specifications do not include a root of trust for long-term secure storage. Platform hardware vendors are free to choose and implement a proprietary mechanism if desired. In this section we discuss hardware roots of trust for secure storage that are available on devices today.

4.3.1 Secure Elements

The Secure Element (SE) provides a solution for establishing a root of trust for mobile devices. SEs provide storage and processing of digital credentials and sensitive data in a physically separate protected module such as a smart-card, thereby reducing the physical attack surface. Ideally, the SE provides a flexible secure platform that supports many applications, each of which can be customized and managed independently [14]. Secure elements fall into three broad categories: software SEs, embedded hardware SEs, and removable hardware SEs [46]. For the purposes of this discussion we only consider hardware-based SE solutions.

An embedded SE is an IC fixed to a mobile device to provide a high degree of security for applications handling sensitive data. Embedded SEs are commonly used to provide security for near field communication (NFC) applications such as automated access control, ticketing, and mobile payment systems. For example, Google Wallet uses embedded secure elements to store and manage encrypted payment card credentials,[6] so that they are never available to a compromised mobile device OS. Development platforms such as the FreeScale i.MX53 (§4.8.1) and Texas Instruments M-Shield (§4.8.2), employ an embedded SE to provide a tamper-resistant secure execution and storage environment.

Removable SEs are interfaced to removable memory such as a Secure Digital (SD) Card or Universal Integrated Cir-

[6] http://www.google.com/wallet/faq.html

cuit Card (UICC). With removable SEs, third-party developers can develop applications against a single platform-independent interface. However, removable SEs are readily physically separated from the mobile device (e.g., the SE may be independently lost or stolen). Giesecke & Devrient and Tyfone are notable vendors currently selling removable SEs.

4.4 Remote Attestation

A remote attestation primitive relies on a private key that is exclusively accessible by a small TCB, and the presence of one or more registers to store measurements (cryptographic hashes) of the loaded code (§3.3). A vast majority of off-the-shelf mobile devices include support for secure or authenticated boot. The boot-ROM is a small immutable piece of code which has access to a public key (or its hash) and authenticates boot components that are signed by the device authority's private key. Platforms such as the FreeScale i.MX53 (§4.8.1) and Texas Instruments' M-Shield (§4.8.2) contain secure on-chip keys (implemented using e-fuses) that are one-time-programmable keys accessible only from inside a designated secure environment for such authentication purposes. However, none of the hardware platforms, to the best of our knowledge, support platform registers to accumulate measurements of the loaded code. In principle, this support could be added in software by leveraging the hardware isolation primitives and secure storage described previously.

4.5 Secure Provisioning

Current mobile platforms implement mechanisms to authenticate external information, with the hash of the public portion of the signing key stored immutably on the device [34]. However, such capabilities are currently restricted to OEMs or carriers (e.g., software updates, assigning different identities to the device) and remain unavailable for use by arbitrary third-party developers.

4.6 Trusted Path

Platforms such as M-Shield (§4.8.2) provide basic hardware primitives to realize a trusted path. A special chip interconnect allows peripheral and memory accesses only by the designated secure environment, and secure DMA channels to guarantee data confidentiality from origin to destination. Such capabilities are being used for DRM (video streaming) on certain off-the-shelf mobile devices [28], but it remains unclear if they are available to third-party developers.

4.7 Design gaps and Challenges

Having described the ARM hardware platform and security architecture and how the different components interplay to provide various hardware security features, we now identify design gaps and implementation challenges in off-the-shelf mobile devices that prevent third-party application developers from fully realizing the desired security features.

ARM's hardware platform architecture is only a specification, leaving the OEMs free to customize a specific implementation to suit their business needs. This means that OEMs could leave out components whose absence can severely constrain some security features and in some cases even break feature correctness. For example, the absence of a TZASC (and/or TZMA) leaves main memory (DRAM/SRAM) accessible to both the secure and normal worlds. The only way to enforce memory isolation between the worlds is to use TCM (§ 4.2.1), which has a very limited size (typically 16-32 KB). Similarly, DMA protection requires a TrustZone-aware DMA controller, GIC, TZASC (and/or TZMA), and a TrustZone-aware AXI-to-APB bridge. The absence of one of these components will result in the DMA protection being ineffective.

Unfortunately, most of today's off-the-shelf mobile devices include a single set of devices shared between the secure and normal worlds and do not include all the required components to fully realize the hardware security primitives described previously. This results in a huge gap between functional specification and device implementation. OEMs and carriers are generally not concerned with DMA-style attacks or including a TZASC (and/or TZMA) because their physical security requirements already force them to process sensitive data in TCM or other device-specific isolated environments unreachable via DMA.

Many OEMs explicitly lock-out platform security features. For example, TrustZone secure-world is enabled or disabled by a single bit in the system configuration register [7]. Once this bit is set to 1 (disabling secure-world), it can no longer be cleared until a device reset. In many off-the-shelf mobile devices such as the Droid, Droid-X, BeagleBoard, and some Gumstix platforms, this bit is set to 1 by the boot-ROM code, in essence allowing only normal-world operations.

From a developer's perspective, an abundance of documentation and open-source (or low-cost) development tools are two key factors that facilitate device and platform adoption. While ARM offers decent documentation and development tools (FastModel/RVDS/RTSM) to leverage the hardware security primitives, the cost of the tools (outside of academia) is greater than cost of a typical device. We believe this to be a significant reason why the open-source and hobbyist community has not rallied around ARM's tools.

4.8 Platform Case Studies

We now describe several readily available, inexpensive development platforms that come with a host of interesting security features. These examples serve to show that there is no shortage of *security potential* in mobile device platforms.

4.8.1 FreeScale i.MX53

The FreeScale i.MX53 is a $149 MSRP development board with an ARM Cortex A8 CPU and many security features.

Secure-Boot Process. The i.MX53 supports a High Assurance Boot (HAB) process where the system boot-ROM prevents the platform from executing unauthorized software dur-

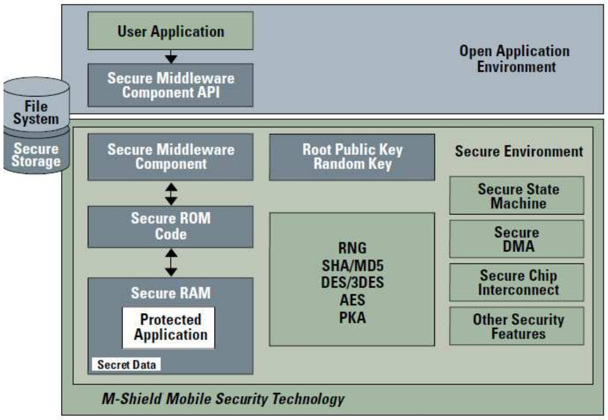

Figure 3: TI's M-Shield Mobile Security Architecture [12].

ing the boot sequence. Using digital signatures to recognize authentic software, HAB supports booting the device to a known initial state, running software signed by the (lifetime-write-once) designated authority.

Secure Cryptographic Key Storage. The i.MX53 Security Controller provides a small Secure RAM area that is self-clearing on tamper detection or software deallocation. The controller is TrustZone-aware and provides configurable access controls. The Code can execute out of the Secure RAM.

Secure Cryptographic Computing Engine. The i.MX53 Security Accelerator (SAHARA) provides a dedicated cryptographic engine for importing data to or exporting data from Secure RAM. It has a 256-bit dedicated secret master key that is protected from other software or hardware accesses. The SAHARA has a dedicated TrustZone-aware DMA controller and accelerates the following cryptographic functions: AES, DES/3DES, ARC4, MD5, HMAC, SHA-1, SHA-224 and SHA-256. It also features entropy generation.

DMA Controller. The i.MX53 Smart Direct Memory Access (SDMA) controller is a software programmable DMA controller that supports two security levels: (a) open mode—where the CPU has full control to load scripts and execution context into SDMA RAM and modify SDMA registers; and (b) locked mode—where selected SDMA registers become read-only to prevent modification of software reset, exception, and debug handling.

4.8.2 Texas Instruments' M-Shield

Texas Instruments M-Shield mobile security technology [12] is a system-level security solution with hardware and software components (Figure 3). M-Shield provides one-time programmable on-chip keys (using e-fuses) that are accessible only from inside the secure environment, and are typically used for authentication and encryption. M-Shield also provides a hardware AES and public-key accelerator, as well as DES/3DES, SHA and MD5 hardware accelerators.

The M-Shield secure environment has a secure state machine (SSM) as well as secure ROM and RAM. The SSM enforces *isolation* by enforcing the system's security policy rules during secure environment entry, execution, and exit.

M-Shield also provides hardware primitives for *trusted path*. A Secure DMA controller tags DMA transfers to protect the confidentiality of sensitive high-value data during their processing and transfer throughout the platform. To further ensure protection against attacks, a secure chip-interconnect allows accessing peripherals and memories only by the secure environment and/or by secure DMA channels so that the confidentiality of sensitive information is guaranteed through the entire data path, from origin to destination.

M-Shield includes a public-key infrastructure that provides a secure means to validate the authenticity and integrity of software on the platform before execution, thereby supporting *secure-boot* and enabling *authenticated-boot*. The platform exposes the TrustZone API (§6.3.3) for managing secure services. According to the white-paper [12], there are associated middleware and developer APIs for developing such secure services. However, documentation detailing those APIs does not seem to be readily available.

5 Isolated Execution Environments

An execution environment that is isolated from the device operating system (§3.1) is perhaps the most critical security feature described in §3. Such an environment can be used to run secure services that multiplex hardware-backed security features, such as secure storage (§3.2), amongst the various stake-holders, including third party application developers.

Greater flexibility can be offered to third-party developers by allowing them to run modules inside that environment. This mechanism provides the strongest security for those modules, since data can be prevented from leaving the secure environment. However, it also requires ensuring that software modules in that environment cannot compromise each other, the environment itself, or the main OS. While this increases the size and complexity of the isolated environment's trusted-computing-base, such an environment remains smaller and more trustworthy than a full-featured OS.

The available isolated-execution hardware primitives (§4.2) offer several options for implementing isolated execution environments. We consider two high-level approaches: either using a parallel execution environment, or multiplexing a single execution environment using a hypervisor.

5.1 Parallel Isolated Execution

One strategy for isolated execution is to put sensitive code in a distinct, parallel environment. As described in §4.2.1, current ARM platforms that support TrustZone offer a mechanism by which secure software can execute in isolation within a special processor world. Several research proposals [19–21, 35, 57, 60] employ TrustZone to achieve isolation and provide a subset of the security properties discussed in §3. Other approaches make use of a physically separate protected module such as a smart-card to achieve isolation. One notable example is the Trusted Execution Module (TEM) [16], which is capable of securely executing procedures (called *closures*) expressing arbitrary computation. The TEM itself is a byte-code interpreter for a small special-purpose programming language. This interpreter is realized as a JavaCard applet, hosted inside a JavaCard-enabled [52] smart-card. Another example is a smart-card-based Mobile Trusted Module (MTM) [18] that implements the MTM functionality in Java applets that can be downloaded into the smart-card. We provide a detailed discussion of the above frameworks in §6.

5.2 Hypervisors

A *microkernel* is a minimal OS, with many components that would be part of a monolithic OS, particularly device drivers, either removed or running as deprivileged processes. A *hypervisor* is a microkernel that can run other OSes as deprivileged processes. OSes can run unmodified if the environment provided by the hypervisor (optionally with help from some of its deprivileged services) matches the physical hardware expected by that OS. Otherwise we say that the OS must be *para-virtualized*—modified to run in the environment that is provided by the hypervisor.

A hypervisor can be used to implement an execution environment that is isolated from the main OS by running the operating system as one process (a virtual machine), and by running the modules to-be-isolated as separate processes.

We now briefly summarize some noteworthy existing ARM hypervisor projects shown in Figure 4. Current closed-source hypervisors include Winter [57], seL4 [33], OKL4 [40], and INTEGRITY [29]. Winter outlines an approach to merge TCG-style Trusted Computing concepts with ARM TrustZone technology in order to build an open Linux-based embedded trusted computing platform. The seL4 project gained notoriety in 2009 when they announced a formally verified microkernel for the ARM architecture. OKL4 is a microkernel-based embedded hypervisor with a small footprint and CPU support to target mobile telephony. The INTEGRITY multivisor uses a security kernel to provide domain isolation and is targeted at in-vehicle infotainment and next-generation mobile devices.

Codezero[7], XenARM [58], and KVMARM[8] are some noteworthy open-source hypervisor initiatives. The CodeZero project proposes a hypervisor based on the L4 microkernel, written in C/C++ in under 10K SLOC. Samsung has supported the Xen hypervisor project to produce an open-source variant of the Xen hypervisor for the ARM architecture. A port is underway of the popular Linux KVM (Kernel Virtual Machine) to the ARM architecture.

Hypervisor frameworks potentially hold value for all stakeholders (OEMs, carriers developers, and users). From an OEM perspective, secure hypervisor frameworks allow multiplexing security-critical baseband functionality on the same processor as popular unmodified OSes and user-facing applications, thereby reducing the cost of materials in a smartphone [40, 43]. Indeed, this appears to be OK Labs' primary business model. From a developer stand-point, hypervisor frameworks allow creation of custom security applications that can benefit from improved isolation (e.g., mobile banking and payments or anti-malware). From a user's perspective, a hypervisor framework may enable simultaneous execution of different OSes, offering a rich set of security features and execution environments on a single mobile device.

Hypervisors are deployed in custom (OEM- and carrier-specific) environments on roughly 1 billion off-the-shelf mobile devices [40, 43]. These can be, and likely already are, used to run security-critical services in isolation from a fully-featured OS running on the same CPU. Unfortunately, we observe that this is done transparently to the user and to third-party developers. These devices do not provide an open API to third-party developers to run *their own modules* in an isolated execution environment provided by the hypervisor.

5.2.1 Limitations of Paravirtualization

All known ARM hypervisors except for KVMARM use paravirtualization to support guest OSes (Figure 4). While KVMARM is targeted at full virtualization, it is a work-in-progress that presently supports a customized v2.6.27 of the Linux kernel, and faults due to a bug while booting the kernel.

While paravirtualization in general has proven successful, and many individual drivers are paravirtualized on many commodity platforms such as x86, there is an unavoidable additional maintenance cost for paravirtualization. Unless the paravirtualized hardware architecture and corresponding OS and driver changes are accepted as a first-class architecture by the OS kernel, the maintainers of the paravirtualization-related changes will perpetually be playing catch-up.

Thus, the price of paravirtualization is increased maintenance cost and more limited availability in terms of supported guest operating system versions. For example, Samsung's Xen for ARM requires modifying approximately 5000 lines of code in the Linux kernel [58]. The most recent kernel version it can support is a modified Linux 2.6.11 kernel, a relatively old version of Linux released in 2005.

6 API Architectures

Having discussed the hardware primitives available on today's mobile platforms in §4, and how those can be used to implement reduced-TCB isolated execution environments in §5, we now discuss potential application programmer interfaces (APIs) that those isolated execution environments may expose to developers.

6.1 API Types

We distinguish between two types of APIs: *App-IEE* APIs and *Module-IEE* APIs.

[7] `http://www.l4dev.org`

[8] `http://wiki.ncl.cs.columbia.edu/wiki/KVMARM:MainPage`

HyperVisor/Microkernel	Virtualization Type	Code Availability	Maturity Level
Winter	Split-world	Closed-source	Unknown
SeL4	Para-virtualization	Closed-source	Unknown
OKL4	Para-virtualization	Closed-source	Mature
INTEGRITY	Para-virtualization	Closed-source	Mature
CodeZero	Para-virtualization	Open-source	Work-in-Progress
XenARM	Para-virtualization	Open-source	Work-in-Progress
KVMARM	Full-virtualization	Open-source	Work-in-Progress

Figure 4: Noteworthy ARM hypervisors and microkernels.

App-IEE APIs specify how normal applications running on the main OS interact with the isolated execution environment. Such APIs could include mechanisms for communicating with modules running inside the isolated execution environment, for discovering service-modules running inside the isolated execution environment, or for loading third-party software modules into the isolated execution environment.

Module-IEE APIs specify how to develop modules running inside the isolated execution environment. As discussed in §3 and §5, such environments will typically minimize their size and complexity by not offering the functionality of a full-featured OS. Instead, these APIs will typically offer some or all of the security services discussed in §3, some APIs for communicating with software running on the full OS, and possibly some APIs for communicating directly with peripherals (i.e., a trusted path to those peripherals).

6.2 App-IEE-only model vs App-IEE + Module-IEE model

A minimal way to make hardware security features available to application developers is for OEMs or network carriers to provide security-relevant services running inside the isolated execution environment, and expose them via App-IEE APIs. This approach may be attractive to OEMs and carriers, who may not want to bear the risk of allowing third-party code to run in the device's isolated environment, or the cost of implementing strong isolation between modules in that environment. Unfortunately, without providing application developers with an isolated execution environment (§3.1) in which to run their modules, the security properties gained have a large TCB that includes the entire OS at runtime. Still, even this strategy improves the set of security features available to third-party developers today, as we detail below.

We first summarize the desirable properties that arise when a Module-IEE API for running custom code in the isolated execution environment *is* available to application developers. Module-IEE APIs for secure storage enable developers to ensure that only their module can access sealed data, even if the OS is compromised. Module-IEE APIs for remote attestation can run code isolated from the OS, and need not include the OS's measurements in their remote attestations. Module-IEE APIs for secure provisioning can ensure that only the intended module running in the isolated execution environment will be able to access provisioned data. A trusted path implemented via Module-IEE APIs can provide assurance to the user that he is communicating with the intended module running in the isolated execution environment.

We now summarize the benefits to application developers that arise from OEM- or carrier-provided security services exposed through an App-IEE interface. Secure storage (§3.2) can be implemented by allowing direct access to a secure storage location, or by implementing a sealed-data API. Data sealed in this way would be protected from offline attacks, and attacks where a different OS is booted (since the sealed-data-service would refuse to unseal for the modified OS). Remote attestation (§3.3) implemented in the App-IEE-only model can attest that a known OS image booted. This can provide some assurance to remote parties that they are communicating with a client that started in a known configuration. However, such mechanisms cannot detect if the OS has been compromised after it was booted. Similarly, a secure provisioning (§3.4) service built in the App-IEE-only model can ensure that exported data can only be accessed by a known device that booted a known OS. However, it would have to trust that OS to not compromise the data itself or to allow unauthorized applications to access that data. A trusted-path service (§3.5) implemented in the App-IEE-only model can ensure to the user that an authorized OS booted, but not that the OS remains uncompromised after it has booted.

6.3 Candidate APIs

We next discuss several published APIs. All of these specify App-IEE APIs; some of them additionally specify Module-IEE APIs.

6.3.1 Mobile Trusted Module

The Mobile Trusted Module (MTM) is a specification by the Trusted Computing Group (TCG) for a set of trusted computing primitives [53]. Like the Trusted Platform Module on PCs, the MTM provides APIs for secure storage and for attestation, but does not by itself provide an isolated execution environment for third-party code or facilities for trusted path.

Unlike the TPM, the MTM is explicitly designed to be implemented in *software*. In particular, it is amenable to being implemented as a module running inside an isolated execution environment on a mobile platform. Also unlike the TPM, the MTM explicitly supports the instantiation of several parallel instances. This feature is intended to support an instance for each of a few stake-holders on a mobile platform. In principle, it could be used to support a private MTM instance to each individual software module that is loaded into an isolated execution environment.

Adding an MTM alone to a mobile platform and allowing third-party developers to access it via App-IEE APIs would serve to expose the underlying hardware security features in a uniform way across hardware platforms.

The MTM could also be used in architectures where third-party code is allowed to execute in an isolated execution environment. However, simply giving secure modules direct access to a single shared MTM instance would put all running modules into each-other's TCB; e.g., modules would be able to unseal data belonging to other modules. This limitation could be addressed by instantiating a fresh, private, MTM instance for each module that runs. Optionally, to minimize complexity and resource-usage, these on-demand MTMs could implement only a small subset of the MTM specification. This is similar to the approach taken by previous research on x86 platforms, with the MTM taking the place of the TPM [41, 47].

Another, orthogonal, way to use an MTM is for the isolated execution environment itself to use the MTM as a backend. This strategy could provide a uniform interface for implementing the isolated execution environment itself across multiple hardware platforms.

While several researchers have implemented the MTM [18, 21, 36, 57, 60], it is not to our knowledge implemented on any off-the-shelf mobile platforms.

6.3.2 OnBoard Credentials

OnBoard Credentials (ObC) [19, 35] is an architecture to provide an isolated execution environment to third-party software modules written in the Lua[9] scripting language. It includes both App-IEE and Module-IEE APIs.

ObC provides most of the features described in §3: an isolated execution environment, secure (sealed) storage, and secure provisioning. It also provides a form of trusted path, implemented using a management application with a customizable interface. Unfortunately it does not provide a remote attestation API, though adding one would be straightforward.

While ObC supports only Lua modules and not native-code modules, this design decision was made so that its isolated execution environment would have very small run-time memory requirements (6 KB for the Lua interpreter). This allows ObC to fit into on-chip memory, thus mitigating physical attacks such as bus-sniffing. This feature is beneficial for some use-cases, e.g., to protect the owner's secrets from being compromised if the device is physically lost or stolen, or DRM applications where the legitimate owner of the physical device may be the attacker.

ObC's key provisioning design seems to be optimized for DRM use-cases, where it is undesirable to have to re-encrypt media for each individual device, As a result, it relies heavily on the physical security of all participating devices. Secured data is provisioned or migrated between devices by encrypting it under a global program-family symmetric key. That key is, in turn, provisioned to devices trusted to protect it and to use it in accordance with ObC policy (i.e., only use it to encrypt or decrypt data for ObC programs that are part of the program-family). In this model, compromising the program-family key from any participating device is sufficient to compromise the confidentiality and integrity of data migrated by that program-family on any device—a break-once, run-anywhere attack. Hence, for applications protecting data that is confidential to the device owner, such as web site or banking credentials, it would be preferable to reduce that attack surface to the set of devices trusted by *that user*.

It may be possible to extend ObC to support a user-centric trust model, by replacing program-family-keys with user-keys, and putting the user in charge of provisioning those keys to the devices that the user owns or otherwise trusts. Such a provisioning mechanism could be built using a remote-attestation mechanism; while ObC assumes the existence of such a mechanism (using device-keys), its API does not expose a remote attestation feature to secure software modules. However, adding such an API would be straightforward.

The primary implementation of ObC uses Texas Instruments' M-Shield [12] to provide an isolated execution environment, secure storage, and integrity of the isolated execution environment (via secure boot). While multiple commodity smartphones are equipped with the necessary hardware support for ObC, enabling it requires a specially signed device firmware image from the OEM or carrier, and is outside the reach of third-party developers and device owners.

6.3.3 TrustZone API

The TrustZone API (not to be confused with the TrustZone hardware features) is an App-IEE API for managing and invoking modules in an isolated execution environment [8].

The TrustZone API appears to have been designed to work with services running in the TrustZone secure world (§4.2.1) in particular; however, the model is fairly abstract and provides interfaces for selecting *which* secure "device" to communicate with. Hence, the TrustZone API could conceivably be implemented to communicate with secure services backed with other protection mechanisms, or even services running on a remote device.

The (publicly available) TrustZone API does *not* include Module-IEE APIs. Hence, while it could be a useful set of APIs to expose to app developers, allowing them to communicate with services running in an isolated execution environment, by itself it does not fully specify the APIs needed for *developing* such service modules.

We are not aware of any mobile platforms where the Trust-

[9] `www.lua.org`

Zone API is open to third-party developers.

6.3.4 GP Trusted Execution Environment (TEE)

The GlobalPlatform consortium is developing a set of standards for a Trusted Execution Environment (TEE) [27]. It includes both App-IEE APIs for applications to interact with isolated modules [25], and Module-IEE APIs for developing such modules [26].

While the system architecture specifically suggests options where the environment is created by multiplexing resources with an untrusted OS, to our knowledge the only implementations of the TEE use a dedicated device such as a Secure Element (§4.3.1) or smartcard, and only run applications in the secure environment that are pre-approved by the entity deploying that device.

The TEE client specification [25] includes APIs for connecting to and invoking a secure application. The TEE internal specification [26] defines the runtime support available to secure applications running inside the TEE. These include communication with calling code outside of the TEE, secure storage (though it is unclear if state continuity is provided [44]), cryptographic primitives, and trusted time.

Of the security features from §3, those missing are remote attestation, secure provisioning, and trusted path. In principle remote attestation can be added, which, as discussed in (§3.3), can be used to build secure provisioning.

7 Analysis and Recommendations

We now give our analysis of the security properties that today's mobile devices can provide, and offer recommendations to the research community, to app developers, to platform integrators, and to hardware vendors.

The set of primary stake-holders today includes only the OEMs and telecommunications carriers (and their immediate business partners). Thus, the hardware security primitives that are actually included in mass-market mobile devices are only those of interest to the OEMs and telecommunications providers. It is our primary recommendation that application developers and device owners be considered first-class *stake-holders* by OEMs and telecommunications service providers. While economics may prevent the inclusion of additional hardware security primitives in mass-market devices without a compelling business reason, those primitives which are present should be leveraged to offer additional security features to application developers and devices owners.

We have shown that—while helpful—the security APIs provided to application developers by today's mobile OSes are inadequate because of the continuing ease with which mobile device OSes are compromised. We have also shown (§2) that the market has responded to the need for security features with add-on hardware that provides Secure Element functionality, either exclusively or in conjunction with new I/O interfaces. Some newer devices (e.g., the Nexus S) are beginning to include embedded Secure Elements; unfortunately, this hardware is being monopolized by a single application.

Given the rise in add-on security devices, it is reasonable to question why the OEMs and carriers have not responded more aggressively by opening up or including additional security features. On this issue we can only speculate, but we list here a few plausible explanations: (1) an existing culture of security-through-obscurity is reluctant to embrace change; (2) business interests are attempting to corner the market for their exclusive use; (3) fragmentation in existing proposals hampers their adoption for fear of future incompatibilities; or (4) there is little awareness of the level of maturity for proposals for multiplexing hardware security features between multiple stake-holders.

We reject the position that OEMs or carriers are unable to monetize such additions. Features sell phones, and security primitives enable application developers to produce new kinds of applications. We reject the position that DRM is the only viable use for such features. Digital media is inherently break-once, run-anywhere, and it *will* eventually be broken. We argue that hardware-backed security features can immediately add significant value in such areas as protecting users' cached login credentials and encrypting users' data while at rest on the phone. Mobile payments and myriad other applications can follow in quick succession.

7.1 Research Community Recommendations

It is our recommendation to the research community to continue to investigate viable architectures for multiplexing mutually-distrusting stake-holders on resource-constrained hardware security primitives (§6). This is especially important as virtualization extensions make their way to the ARM architecture (§4.2.2), opening up the possibility for two divergent approaches (split-world vs. virtualization). Special attention should be paid to the possibility for a heterogeneous threat model: OEMs and carriers are concerned about defenses against physical attacks, whereas many use-cases for protecting the end-user's data are primarily concerned with software-based attacks that arrive via a network connection.

Development hardware with a multitude of unlocked security features is now readily available and inexpensive (§4.8). Though hardware with virtualization extensions remains unavailable at the time of this writing, ARM's toolkit enables emulation of Cortex A15 platforms today. Open-source contributions of viable multiplexing architectures can serve to raise awareness with OEMs, carriers, and application developers. We are optimistic that a robust reference implementation could even enjoy widespread deployment.

The fear of fragmentation of security APIs can be addressed by developing consistent interfaces. We recommend the adoption of consistent Module-IEE and App-IEE APIs, so that application developers that endeavor to privilege-separate their programs today can continue to reap the security benefits into the future without significant risk of incompatibility or maintenance / support nightmares. Support for multiple feature sets may also be reasonable. For example, credential programs such as those enabled by On-Board Credentials (§6.3.2) may reasonably coexist with more feature-rich isolated execution environments that allow arbitrary computation (within resource limits).

7.2 Application Developer Recommendations

It is our recommendation to application developers to continue to demand improved security APIs and primitives in the development environment for popular mobile device platforms. The incredible volume of misinformation bandied about on Internet forums about how to protect cached credentials, encrypt data at rest, or give users a false sense of security is deeply disturbing.

We encourage application developers to learn about existing proposals for Module-IEE and App-IEE APIs, and to consider their implications for the architecture of their applications. Especially those developers with an interest in open-source can produce reference implementations that we expect may be rapidly adopted by other developers.

7.3 Platform Integrator Recommendations

We recommend that platform integrators (typically network carriers) take an interest in the security of applications on their devices. We argue that they should adopt a realistic perspective regarding the robustness of the OS APIs for security.

Hardware-backed or otherwise isolated security features are in demand by application developers. Existing Module-IEE and App-IEE proposals should be adopted, to avoid fragmentation and a lack of developer buy-in. These security features will enable application developers to add new value to the mobile device platforms as a whole, resulting in an overall increase in the utility of mobile devices.

We strongly urge platform integrators to make hardware security features available that are otherwise included in the silicon but disabled immediately during every boot. Other developers and the open-source community are likely to energize and do much of the engineering. As a viable first step, we recommend an implementation of the TCG's Mobile Trusted Module (MTM) in devices with TrustZone capabilities that are otherwise unused (§6.3.1). This suggestion is consistent with the App-IEE-only approach discussed in §6.1, and offers new security features to application developers. Note that it does not give application developers the ability to directly execute their own code inside of an isolated execution environment (§3.1 and §6.2). Thus, it is a reasonable compromise between conservative, risk-averse OEMs and carriers, and a useful set of APIs for application developers.

We emphasize that platform openness and security are *not* fundamentally opposing trade-offs, and that additional access to hardware security primitives will not somehow weaken them. The key to reconciling this common misconception is isolation. A single vulnerable application—even a security-critical subcomponent of an application—should never be in a position where it is able to compromise the entire device.

7.4 Hardware Vendor Recommendations

Unconstrained memory isolation and improved protection against DMA-based attacks (§4.7) are significant needs in current device hardware. It is more difficult for us to justify the added expense in device hardware at the present time. If the market does indeed parallel our recommendations in the preceding sections, and existing hardware security features begin to enable new applications, then the logical next step is to offer additional hardware security features.

To this end, our recommendation is to address the DMA insecurity problem (§4.7). This will not only add protection against currently prevalent attacks from malicious peripherals [56], but will also result in the automatic inclusion of memory address-space controllers such as a TZASC and/or TZMA (§4.2.1), so that security-sensitive modules that execute in isolation need not grapple with today's dearth of Tightly Coupled Memory.

References

[1] Android – An Open Handset Alliance Project. Issue 10809: Password is stored on disk in plain text. `code.google.com`, Aug. 2010.

[2] Android Developers. Android API: AccountManager. `developer.android.com`. Accessed, Nov. 2011.

[3] Android Open Source. Notes on the implementation of encryption in Android 3.0. `source.android.com`. Accessed, Nov. 2011.

[4] android.com. Android 4.0 platform highlights. `http://developer.android.com/sdk/android-4.0-highlights.html#security-dev`.

[5] Apple. iOS: Understanding data protection. Article HT4175, Oct. 2011.

[6] ARM Limited. ARM builds security foundation for future wireless and consumer devices. ARM Press Release, May 2003.

[7] ARM Limited. ARM security technology: Building a secure system using TrustZone technology. WhitePaper PRD29-GENC-009492C, Apr. 2009.

[8] ARM Limited. TrustZone API specification 3.0. Technical Report PRD29-USGC-000089 3.1, ARM, Feb. 2009.

[9] ARM Limited. AMBA 4 AXI4-Stream protocol version 1.0 specification, Mar. 2010.

[10] ARM Limited. AMBA APB protocol version 2.0 specification, Apr. 2010.

[11] ARM Limited. Virtualization extensions architecture specification. `http://infocenter.arm.com`, Oct. 2010.

[12] J. Azema and G. Fayad. M-Shield mobile security: Making wireless secure. Texas Instruments WhitePaper, June 2008.

[13] M. Becher, F. C. Freiling, J. Hoffman, T. Holz, S. Uellenbeck, and C. Wolf. Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[14] B. Choudhary and J. Risikko. Mobile device security element: Key findings from technical analysis v1.0. Mobey Forum: Mobile Financial Services, 2005.

[15] comex. JailbreakMe. `jailbreakme.com`. Accessed, Nov. 2011.

[16] V. Costan, L. F. Sarmenta, M. van Dijk, and S. Devadas. The trusted execution module: Commodity general-purpose trusted computing. In *Proceedings of CARDIS*, 2008.

[17] L. F. Cranor. What do they "indicate"?: evaluating security and privacy indicators. *Interactions*, 13(3):45–47, 2006.

[18] K. Dietrich and J. Winter. Towards customizable, application specific mobile trusted modules. In *Proceedings of the ACM workshop on Scalable Trusted Computing*, 2010.

[19] J. E. Ekberg, N. Asokan, K. Kostiainen, and A. Rantala. Scheduling execution of credentials in constrained secure environments. In *Proceedings of the ACM workshop on Scalable Trusted Computing*, 2008.

[20] J.-E. Ekberg and M. Kylänpää. Mobile trusted module (mtm) – an introduction. Technical Report NRC-TR-2007-015, Nokia Research Center, Nov. 2007.

[21] J.-E. Ekberg and M. Kylänpää. MTM implementation on the TPM emulator. Source code: `mtm.nrsec.com`, Feb. 2008.

[22] ElcomSoft: Proactive Software. iOS forensic toolkit, Nov. 2011.

[23] P. Gilbert, L. P. Cox, J. Jung, and D. Wetherall. Toward trustworthy mobile sensing. In *Proceedings of the Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2010.

[24] V. D. Gligor, C. S. Chandersekaran, R. S. Chapman, L. J. Dotterer, M. S. Hecht, W.-D. Jiang, A. Johri, G. L. Luckenbaugh, and N. Vasudevan. Design and implementation of Secure Xenix. *IEEE Transactions on Software Engineering*, 13:208–221, February 1986.

[25] Global Platform Device Technology. TEE client API specification version 1.0. Technical Report GPD_SPE_007, `globalplatform.org`, July 2010.

[26] GlobalPlatform Device Technology. TEE internal API specification version 0.27. Technical Report GPD_SPE_010, `globalplatform.org`, Sept. 2011.

[27] GlobalPlatform Device Technology. TEE system architecture version 0.4. Technical Report GPD_SPE_009, `globalplatform.org`, Oct. 2011.

[28] GottaBeMobile. Texas Instruments ARM OMAP4 becomes first mobile CPU to get Netflix certification for Android HD streaming. `gottabemobile.com`, 2011.

[29] Green Hills Software. Emergence of the mobile multivisor. `ghs.com`, 2011.

[30] J. Guaus, L. Kanniainen, P. Koistinen, P. Laaksonen, K. Murphy, J. Remes, N. Taylor, and O. Welin. Best practice for mobile financial services v1.0. Technical report, Mobey Forum, 2008.

[31] M. S. Hecht, M. E. Carson, C. S. Chandersekaran, R. S. Chapman, L. J. Dotterrer, V. D. Gligor, W. D. Jiang, A. Johri, G. L. Luckenbaugh, and N. Vasudevan. UNIX without the superuser. In *Proceedings of USENIX Technical Conference*, pages 243–256, 1987.

[32] J. Heider and M. Boll. Lost iPhone? Lost passwords! Practical consideration of iOS device encryption security. Technical report, Fraunhofer SIT, Feb. 2011.

[33] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[34] K. Koistiainen, E. Reshetova, J.-E. Ekberg, and N. Asokan. Old, new, borrowed, blue—a perspective on the evolution of mobile platform security architectures. In *Proceedings of the first ACM conference on data and application security and privacy (CODASPY)*, 2011.

[35] K. Kostiainen, J. E. Ekberg, N. Asokan, and A. Rantala. On-board credentials with open provisioning. In *Proceedings of ASIACCS*, 2009.

[36] K. Kursawe and D. Schellekens. Flexible MicroTPMs through disembedding. In *Proceedings of ASIACCS*, 2009.

[37] B. Lampson. Usable security: How to get it. *Communications of the ACM*, 52(11), 2009.

[38] A. Lineberry, T. Strazzere, and T. Wyatt. Inside the Android security patch lifecycle. Presented at BlackHat, Aug. 2011.

[39] M. Mastin. Square vs. intuit gopayment: Mobile credit card systems compared. PCWorld. `http://www.pcworld.com/businesscenter/article/239250/`, Sept. 2011.

[40] R. McCammon. How to build a more secure smartphone with mobile virtualization and other commercial off-the-shelf technology. Open Kernel Labs Technology White Paper, Sept. 2010.

[41] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[42] E. Mills. Researchers find avenues for fraud in square. CNET. `http://news.cnet.com/8301-27080_3-20088441-245/`, Aug. 2011.

[43] Open Kernel Labs. OK Labs company datasheet. `www.ok-labs.com`, 2010.

[44] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2011.

[45] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17, July 1974.

[46] M. Reveilhac and M. Pasquet. Promising secure element alternatives for NFC technology. In *Proceedings of the International Workshop on Near Field Communication (NFC)*, Feb. 2009.

[47] R. Sailer, T. Jaeger, E. Valdez, R. Cáceres, R. Perez, S. Berger, J. Griffin, and L. van Doorn. Building a MAC-based security architecture for the Xen opensource hypervisor. In *Proceedings of the Annual Computer Security Applications Conference*, Dec. 2005.

[48] S. Saroiu and A. Wolman. I am a sensor, and I approve this message. In *Proceedings of the Workshop on Mobile Computing Systems & Applications (HotMobile)*, 2010.

[49] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The emperor's new security indicators. In *IEEE Symposium on Security and Privacy*, 2007.

[50] S. V. Schell, M. Narang, and R. Caballero. US Patent 2011/0269423 Al: Wireless Network Authentication Apparatus and Methods, Nov. 2011.

[51] M. J. Schwartz. Apple iOS zero-day PDF vulnerability exposed. InformationWeek. `http://www.informationweek.com/news/231001147`, July 2011.

[52] Sun Microsystems, Inc. Java card specifications v3.0.1: Classic edition, Connected edition, May 2009.

[53] TCG Mobile Phone Working Group. TCG mobile trusted module specification. Version 1.0, Revision 7.02, Apr. 2010.

[54] Texas Instruments E2E Community. Setup of secure world environment using TrustZone. OMAP35X Processors Forum, `e2e.ti.com`, Aug. 2010.

[55] US Department of Defense. Trusted computer system evaluation criteria (orange book). DoD 5200.28-STD, Dec. 1985.

[56] Z. Wang and A. Stavrou. Exploiting smart-phone usb connectivity for fun and profit. In *Proceedings of the Annual Computer Security and Applications Conference (ACSAC)*, 2010.

[57] J. Winter. Trusted computing building blocks for embedded linux-based ARM TrustZone platforms. In *Proceedings of the ACM workshop on Scalable Trusted Computing*, 2008.

[58] Xen.org. Xen ARM project. `wiki.xen.org/wiki/XenARM`. Accessed, Nov. 2011.

[59] Y. Yao. Security issue exposed by android accountmanager. `http://security-n-tech.blogspot.com/2011/01/security-issue-exposed-by-android.html`, Jan. 2011.

[60] X. Zhang, O. Aciicmez, and J. P. Seifert. A trusted mobile phone reference architecture via secure kernel. In *Proceedings of the ACM workshop on Scalable Trusted Computing*, 2007.