Using ARM TrustZone to Build a Trusted Language Runtime for Mobile Applications

Nuno Santos[†1], Himanshu Raj[‡2], Stefan Saroiu[‡3], Alec Wolman[‡4]

[†]INESC-ID / Instituto Superior Técnico, University of Lisbon
[‡]Microsoft Research
[1]nuno.santos@inesc-id.pt, [2]rhim@microsoft.com, [3]ssaroiu@microsoft.com, [4]alecw@microsoft.com

Abstract

This paper presents the design, implementation, and evaluation of the Trusted Language Runtime (TLR), a system that protects the confidentiality and integrity of .NET mobile applications from OS security breaches. TLR enables separating an application's security-sensitive logic from the rest of the application, and isolates it from the OS and other apps. TLR provides runtime support for the secure component based on a .NET implementation for embedded devices. TLR reduces the TCB of an open source .NET implementation by a factor of 78 with a tolerable performance cost. The main benefit of the TLR is to bring the developer benefits of managed code to trusted computing. With the TLR, developers can build their trusted components with the productivity benefits of modern high-level languages, such as strong typing and garbage collection.

Categories and Subject Descriptors D.4.6 [*Security and Protection*]: Security Kernels

Keywords Mobile Computing; Trusted Computing; ARM TrustZone; Language Runtime

1. Introduction

E-wallet and *e-health* mobile apps have already started to revolutionize the way people make purchases, and how they handle their health records. As mobile apps start to handle security-sensitive data, smartphones become an attractive target for attacks. In particular, data such as personal photos, location trails, and online banking information have a high value to spammers and identity thieves. As a result, mobile applications have emerged recently with questionable practices [20] as well as outright malware [24].

Unfortunately, protecting data on mobile devices is far from trivial. Typically, mobile apps rely on ad-hoc OS and application-level mechanisms to protect sensitive data and prevent leaks. However, the Trusted Computing Base (TCB) code that mobile apps depend upon is very complex: popular mobile platforms based on iOS [3], Android [1], or Windows 8 [11] comprise a full blown OS, local services, and system libraries, consisting of millions of lines of code (LOC). Therefore it is difficult to ensure the absence of exploitable code vulnerabilities that could be used to disable security checks and retrieve sensitive data.

To improve this situation, the research community has studied the design of trusted computing systems with small TCBs. Flicker [28] and TrustVisor [29] provide confidentiality and integrity protection of application code and data while depending on a small TCB. These systems allow application developers to execute parts of the application logic in a trusted environment isolated from the OS. Because only a few basic services are offered in the trusted environment, in their minimal setup these systems can be built with a TCB on the order of tens of thousands of lines of code.

However, while this prior research has managed to explore the limits in shrinking the TCB of trusted computing systems, the functionality of these systems may be too restrictive for mobile applications. Mobile apps are typically written in high-level languages and compiled to intermediate code (e.g., Dalvik *bytecode* [2] or .NET *managed code* [5]). Flicker and TrustVisor can only execute small pieces of application logic written in native code; they have no built-in runtime engine and are unable to interpret intermediate code.

This paper presents the Trusted Language Runtime (TLR), a small runtime engine capable of interpreting .NET managed code inside a trusted environment. While adding this runtime increases the TCB size, TLR is carefully crafted to avoid a bloated TCB and keep it significantly smaller than a full-blown .NET framework and a full-featured OS. TLR uses three techniques to keep the TCB small: (i) allow application developers to factor out the security-sensitive app

ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.

ACM 978-1-4503-2305-5/14/03. . . $15.00.
http://dx.doi.org/10.1145/2541940.2541949

logic into classes that transparently run in a trusted environment, (ii) isolate the TLR and the trusted app code from the bulk of the system software by using ARM TrustZone technology [14], and (iii) borrow parts of the runtime engine design from the .NET Micro Framework (NETMF) [6], a small .NET implementation designed for embedded devices.

Although we presented the basic TLR design in an earlier workshop paper [34], this paper goes beyond the original TLR proposal and makes the following contributions. First, we provide a complete design of the TLR architecture describing the components running in the trusted environment, in the OS, and in the untrusted application. We handle issues such as the arbitration of resource management between both the TLR and the native OS (i.e., how to securely share memory, CPU, and handle interrupts across domains) and include low-level mechanisms that coordinate the execution flow of the applications between the trusted and the untrusted domains (e.g., transparently bind the native-code processes of the OS with managed-code threads in the TLR).

Second, we describe our TLR prototype implementation on real TrustZone hardware, rather than in an ARM simulator. We present the challenges related to the TrustZone compatibility and portability issues of a large OS, namely Linux.

Third, we present a thorough evaluation of the TLR. We describe the performance overhead due to running interpreted code rather than native code. However, this overhead does not hurt the user experience for the use cases and benchmarks we tested. To illustrate that the TLR can be useful for realistic applications, we implemented several financial and e-health mobile apps. This experience shows that the TLR is easy to program because of its tight integration with .NET that offers the productivity benefits of modern high-level languages and rich software development tools. Our results show that, although the TCB size of the TLR is larger than existing trusted computing systems (namely, it is $6\times$ larger than TrustVisor), the TLR's TCB reduction when targeting an open source .NET setup based on Mono [4] and Linux is substantial, namely by a factor of 78.

2. Background

This section provides background on ARM TrustZone and the .NET Micro Framework (NETMF), which are the key technologies used in the design of the TLR.

2.1 ARM TrustZone

TrustZone [14] is a hardware security technology incorporated into recent ARM processors. It consists of security extensions to an ARM System-On-Chip (SoC) covering the processor, memory, and peripherals. These mechanisms can be leveraged by system designers to run secure services in isolation from the operating system (OS). The secure services that can be built range from a simple library to a complete OS [14]. We briefly describe the relevant TrustZone mechanisms.

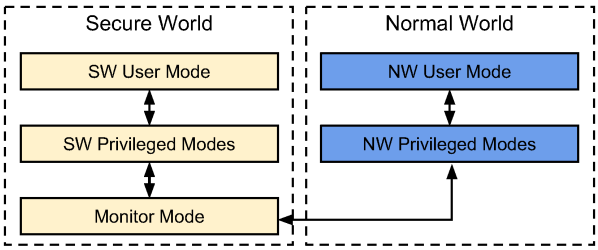


Figure 1. Processor modes of ARM-based device with TrustZone extensions.

With TrustZone, the processor can execute instructions in one of two possible security modes, referred to as the *normal world*, where untrusted code executes, and the *secure world*, where secure services run (see Figure 1). These processor modes have independent memory address spaces and different privileges. While code running in the normal world cannot access the secure world address space, code running in the secure world can access the normal world address space in certain conditions. A special processor bit, the *NS* bit, indicates which world the processor is currently executing in, and this bit is sent over the memory bus and certain I/O buses for peripherals. This enables the system designer to allocate memory solely to the secure world, and to control which devices are accessible from the different worlds. Hardware interrupts can trap directly into the secure world interrupt handler, which then enables flexible routing of those interrupts to either world.

Because the processor executes in one security mode at a time, to execute software in another security mode the processor must switch worlds. World switch is done via a special instruction called the Secure Monitor Call (*smc*). When the CPU executes the *smc* instruction, the hardware switches into the secure monitor, which (i) performs a secure context switch into the secure world, and (ii) enables sharing data by copying data across worlds.

2.2 Microsoft .NET Micro Framework

We use the .NET Micro Framework (NETMF) [6] as a starting point for the TLR design. The NETMF is a lightweight implementation of Microsoft's .NET optimized for embedded systems, such as sensor networks, robotics, and wearable devices [13].

The NETMF design was driven by three main tenets. First, an emphasis was put on offering a user-friendly and robust programming environment. To this end, NETMF enables application programmers to use fully featured development tools like Microsoft Visual Studio, high-level languages like C#, and a collection of code libraries to program embedded systems. Applications are compiled into *managed code*, an intermediate language comparable to Java bytecodes, and interpreted by a CLR runtime (the equivalent version of the Java virtual machine in .NET terminology). The CLR included in NETMF provides a type system, code execution safety, and garbage collection. Second, NETMF was

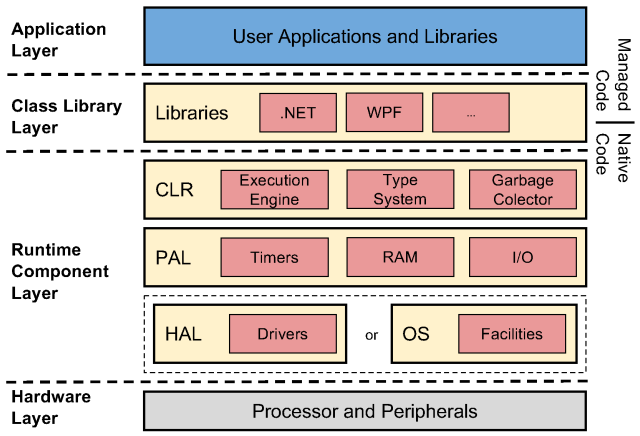


Figure 2. Architecture of the .NET Micro Framework.

tailored for resource constrained devices. For improved efficiency, NETMF precludes the existence of an underlying OS and runs directly on metal. Internally, NETMF owns all execution and it includes only the bare system functionality for managing memory, CPU, and peripherals. To optimize resources, some .NET features are not currently supported (e.g., no support for multidimensional arrays, and no templates). Third, NETMF was geared towards customizability for a wide variety of devices. For this reason, it includes a hardware abstraction layer (HAL) and a platform abstraction layer (PAL).

Figure 2 shows the NETMF hardware and software architecture in more detail. The hardware layer consists of the processor and peripherals. The runtime layer includes the CLR, the HAL, and the PAL. The CLR contains modules for managed code execution, thread scheduling, memory management, and other system services. The HAL and PAL manage the underlying hardware components. The class library layer is an object-oriented collection of classes implemented in C# that application developers can use to program embedded applications. The application layer contains the managed code of apps.

3. Goals, Assumptions, and Threat Model

TLR was designed to meet the following three goals:

1. **Small TCB size:** The TCB of the TLR should not include the OS nor any untrusted application code.
2. **Ease of programming:** Programming the TLR should be as simple as programming any of today's managed code environments such as Java or .NET.
3. **Compatible with legacy software environments:** Deploying the TLR should not require a radical redesign of today's legacy operating systems or other legacy software running on the mobile device.

We require that mobile devices where the TLR is deployed support the TrustZone security extensions, and that their hardware behaves correctly. We assume the existence

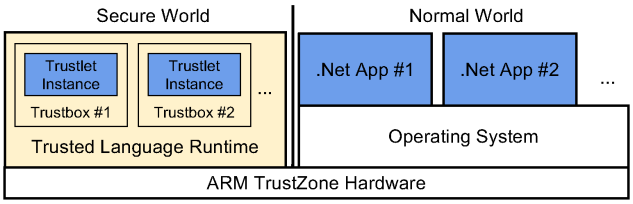


Figure 3. TLR high-level architecture.

of external *trusted parties*, such as offline certification authorities or online services with which security-sensitive apps deployed on the TLR can communicate over secure channels. Lastly, we rely on the correctness of cryptographic primitives and algorithms. Note that we make no assumptions about the correctness of the operating system on the mobile platform.

TLR protects the application state against the following adversary. The attacker can compromise the OS and have access to the TLR interface, which is provided through specific TrustZone mechanisms. The attacker can reboot the mobile platform and gain access to data residing on persistent storage. She can eavesdrop the network and interfere with the communication between the TLR and any third party trusted components outside the device. However, we do not consider side-channel attacks nor physical attacks that fall outside the defense capabilities of TrustZone technology, namely those that require disassembling the chip packages of application processors or memory modules.

4. Overview of Trusted Language Runtime

Figure 3 illustrates the TLR's high-level architecture. TLR provides two execution environments: an *untrusted* one where the smartphone's OS and most application software run, and a *trusted* one, where the TLR code and security-sensitive application components run. These environments map to TrustZone's normal and secure worlds, respectively. Code running in the trusted environment is isolated via TrustZone from all code running in the untrusted one. TLR provides a secure communication channel between the two environments.

With the TLR, a mobile application must be partitioned into two components: a small trusted component running in the secure world, and a larger untrusted component implementing most of the application's functionality. This partitioning technique is similar to privilege separation [17] and to partitioning of applications for improved security in distributed systems [19].

In the trusted world, TLR provides a language runtime based on NETMF. Application code running in the TLR can only perform computations and has no access to peripherals, which are all managed by the untrusted OS. We deliberately limit TLR's access to peripherals to keep a small TCB. Adding I/O access requires building drivers inside the TrustZone, that may have large codebases. Even without access

to peripherals, TLR offers enough functionality to meet the security needs of many mobile applications. Section 5 will describe four such applications.

4.1 TLR Primitives

1. Trustbox. A trustbox is an isolation runtime environment that protects the integrity and confidentiality of code and data. The smartphone's OS (or any untrusted application code) cannot tamper with code running in a trustbox nor inspect its state.
2. Trustlet. A trustlet is a class within an application designated to run inside a trustbox. The trustlet specifies an interface that defines what data can cross the boundary between the trustbox and the untrusted world. The .NET runtime's use of strong types ensures that the data crossing this boundary is clearly defined.
3. Platform identity. Each device that supports the TLR must provide a unique cryptographic platform identity used for platform authentication and for protection of any trusted code and data deployed on the platform (using encryption). We use a public key pair as the platform ID, and the TLR ensures that the private key is never revealed to any external component.
4. Seal/Unseal data. The *seal* primitive encrypts data and binds it to a particular trustlet and platform identity. The trustlet's identity is based on a secure hash of its code (e.g., SHA-2). The *unseal* primitive yields the data contained in the sealed envelope only if this operation is performed (i) inside a trustbox, and (ii) by the trustlet and platform ID originally specified upon seal. To recover sealed data, unseal decrypts it and checks that the hash of the requesting trustlet matches the hash of the trustlet that sealed the data. Seal and Unseal serve two roles: (i) allowing a trustlet to persist state across reboots, and (ii) enabling a remote trusted party (e.g., a trusted server) to encrypt data so that only a designated trustlet can decrypt it.

Typical development scenario. To build a trusted mobile application with the TLR, a developer typically performs the following four steps: (i) partition off a small part of the application that needs to handle sensitive data into a trustlet, (ii) deploy the trustlet on the designated platform, (iii) ensure that sensitive data can only be accessed on a designated platform by sealing the data to the trustlet running on the designated platform, and (iv) deploy the sealed data to the designated platform to run it inside its trustbox, thereby ensuring that the trustlet state is protected at runtime.

5. Use Cases

To illustrate TLR's practicality, we show how to implement four common security use cases in TLR: storing one-time passwords, user authentication, secure mobile transactions, and access control to sensitive data. We describe these use cases in the context of four applications. While our description is done at a high-level, more details of these applications' security protocols can be found in Appendix A.

5.1 One-time Passwords

A commonly used form of one-time passwords is *Transaction Authentication Numbers* (TANs) [8] for online banking purposes. Today, some banks send paper cards to their customers with a list of TANs. When a customer initiates an online transfer, the bank specifies an index into the TAN list and asks for the associated TAN. The customer, in addition to typing their personal password, must respond with the correct TAN, otherwise the transaction is aborted. Each TAN is used once; when all TANs are used, the bank sends a new TAN list to the customer.

To avoid the burden of carrying a physical TAN list, banks can use the TLR to provide secure storage of *digital* TAN lists. A mobile app using the TLR keeps track of the TAN list on the customer's smartphone, and provides an interface for querying a TAN based on a TAN index. The TAN indices are not simply sequential integers, instead they are capabilities: hard-to-guess numbers chosen from a large address space.

The bank seals TAN lists on a per-customer basis such that it can only be unsealed by the bank's trustlet running on the customer's smartphone. When the customer initiates a bank transfer, the bank sends a TAN index challenge to the trustlet, and the trustlet unseals the TAN list, finds the index, and responds to the bank with the appropriate TAN value that authorizes the transfer. Because TAN indices are only used once, there is no opportunity for man-in-the-middle replay attacks. Because TAN indices are hard-to-guess, there is no opportunity for a malicious entity to obtain the TAN list. The trustlet interface only requires two methods: `LoadTanLst`, and `GetTan` (see Figure 5 for code excerpts).

5.2 User Authentication

Our second use case shows how to use the TLR for user authentication purposes by implementing a mobile ticketing app. Unlike existing digital ticket mechanisms that reveal the ticket during validation (e.g., QR codes), user authentication can be done by a TLR app without disclosing the ticket details. Apps that only send proof of ticket possession reduce the possibility of ticket theft by malware.

We illustrate how mobile ticketing could be done for a public transit company. There are three actors: a ticket issuer (the public transit service), a mobile ticketing app with a trustlet, and ticket verifiers (terminals at the entrance to the bus or subway). The issuer sends sealed tickets to the trustlet at purchase time. To obtain access, the user validates the digital ticket by asking the trustlet to issue a ticket proof to the terminal. If the proof is valid, the terminal produces a visual or audio output reflecting the verification result.

5.3 Secure Mobile Transactions

Our third use case uses the TLR to perform secure transactions. The mobile app enables payments at *point of sale*

(POS) terminals by simply waving a smartphone in front of the POS. A trustlet stores the customer's credit card details and engages in a payment protocol with the POS (e.g., via Near Field Communication (NFC) wireless technology). The TLR's role is to secure the runtime state of the mobile app.

We illustrate this mobile payment scenario involving three actors: a *bank*, which issues credit card information, the *mobile payment trustlet*, which keeps track of the credit card details, and the POS vending terminal. The bank seals the credit card information to the trustlet. To perform a transaction, the POS terminal issues a sealed challenge to the trustlet that includes the transaction amount. If the user authorizes the transaction, the trustlet answers the challenge, otherwise it aborts. Possibly at a later point in time, the trustlet communicates with the banks to record the transactions.

5.4 Access Control to Sensitive Data

Our final use case describes enforcing access control policies for security-sensitive data in the context of an *e-health* app. The app's goal is to store a personal clinical history on the smartphone and to give health providers secure access to this information during patient visits.

Because giving health providers unrestricted access to health records could raise serious privacy concerns, the TLR can restrict health providers' privileges to this data. Assuming that a central Health Care Authority (HCA) defines a user's access control policy for their clinical history, the TLR can enforce these policies and regulate access to health record information stored on mobile devices.

Our e-health app involves three actors: the *HCA*, the *health providers*, and the *e-health trustlet*. The HCA seals the health records and access control policies to the e-health trustlet. When a health provider asks for a record, the request is sealed to the trustlet. The trustlet unseals it, checks whether the provider's permissions meet the access control policy, and returns the relevant information if the policy is met.

6. Detailed Design

This section describes the internals and key design features of the TLR.

6.1 TLR Components

Figure 4 shows a detailed view of the TLR architecture. TLR spans the two TrustZone security worlds: the *TLR-domain* running in the secure world and the *OS-domain* running in the normal world. The TLR-domain hosts the TCB of the system: the *TLR core* components and the application trustlets. The OS-domain hosts the untrusted system components: the OS, the *TLR stubs*, and the untrusted application partitions. The TLR stack consists of four layers:

1. Application layer: corresponds to the logic of a mobile app split between a trustlet and an untrusted partition.

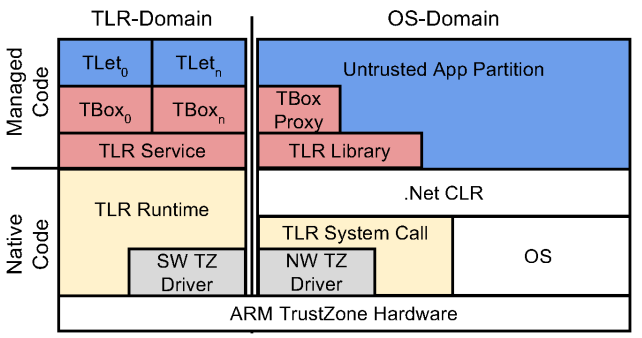


Figure 4. Component diagram of the entire TLR system. Components in each layer use the same color.

2. Trustbox layer: manages the state of the trustlet instances living in trustboxes. In the TLR-domain, dedicated service threads host the runtime state of trustlets. In the OS-domain, the TLR libraries and proxies contain the logic that bridges the untrusted app partition with the trustlet logic.

3. Runtime layer: executes the managed code of trustlet instances and serves their memory allocation needs. In the TLR-domain, the TLR runtime manages the service threads, interprets trustlets' managed code, and manages trustbox resources. In the OS-domain, a a TLR-specific system call bridges the trustbox requests coming from the local application processes (running the untrusted app code) and the TLR.

4. Trustzone layer: consists of drivers providing the low-level TrustZone mechanisms responsible for world switching, interrupt handling, communication, and protection.

6.2 Programming Model

To implement a trusted application with the TLR, a developer performs the following five steps. As we explain each step, we refer to Figure 5 which contains code snippets of the TAN-based e-banking application previously described in Section 5.1.

1. Specify the security-sensitive logic: The security-sensitive app logic must be enclosed in a trustlet class. The developer creates a trustlet by defining its interface, implementing the class, and creating a meta-data file—the *manifest* (see Figure 5). The trustlet interface must inherit from the `IEntrypoint` interface, and the trustlet main class must inherit from the `Trustlet` class and implement the trustlet interface. The public methods that implement the trustlet interface enable data to cross the barrier between the trusted and untrusted worlds. The programmer must be careful not to let any sensitive data protected by the trustbox leak out into the untrusted world. The manifest's role is to specify the trustlet's class and interface.

2. Instantiate a trustlet inside a trustbox container: The developer must instantiate the trustlet class inside a TLR trustbox. To create a TLR trustbox, an application invokes

Trustlet Interface

```
public interface ITanWallet : IEntrypoint
{
   public void Load(Envelope tanLst);
   public Tan GetTan(long id);
}
```

Trustlet Main Class

```
public class TanWallet: ITanWallet, Trustlet
{
   private TanList _tanLst = null;

   public void Load(Envelope tans) {
      try {
         _tanLst = (TanList) this.Unseal(tans);
      } catch(Exception e) {
         throw new Exception("Cannot unseal TAN list.");
      }
   }

   public Tan GetTan(long id) {
      Tan tan = _tanLst.Search(id);
      if (tan == null) {
         throw new Exception("ID invalid.");
      } else {
         return tan;
      }
   }
}
```

Trustlet Manifest

```
<trustlet name="TanWallet">
   <interface name="ITanWallet" />
   <implementation name="TanWallet" />
</trustlet>
```

Snippet of Main Class

```
// setup the TAN wallet trustlet in a trustbox
Trustbox tbox = Trustbox.Create("TanWallet.pkg");

// obtain a reference to trustbox entrypoint
ITanWallet twallet = (ITanWallet) tbox.Entrypoint();

// load the TAN list issued and sealed by bank
twallet.Load(myTanLst);

// obtain a TAN with id requested by bank
Tan tan = twallet.GetTan(id);
```

Snippet of Third Party Service

```
// the bank generates TAN list for customer
TanList newLst = customer.GenTanLst();

// seal the list
Envelope sealedLst = Trustlet.Seal(customer.PlatformID(),
   Trustlet.Hash("TanWallet.pkg"), newLst);
```

Figure 5. Code sample of a TLR application that manages Transaction Authentication Numbers (TAN) for online banking services (written in C#).

the `Create` method, which takes as input the manifest describing the trustlet class to be instantiated. The role of the `Destroy` method is to clean up the trustbox and release all its resources.

3. Interact with the trustlet instance: The untrusted app partition interacts with the trustlet by invoking its methods. Because the trustlet instance and the untrusted application partition reside in separate domains, the calls must be routed across domains. To make this process transparent to the developer, the TLR library returns a proxy object with a method interface compatible with the trustlet.

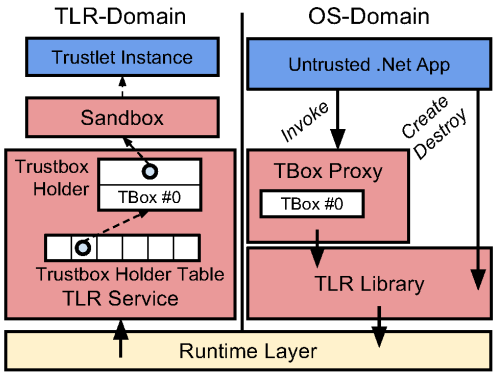


Figure 6. Details of the trustbox layer.

4. Validate trustlet identity and integrity: Because arbitrary trustlet code can be instantiated inside trustboxes, third parties must validate the identity and integrity of the trustlet instances before uploading security-sensitive data into the trustbox. For this, the TLR provides the `Seal` and `Unseal` primitives, whose behavior is presented in Section 4.

5. Compiling and packaging the application: In addition to the standard compilation and linking operations, two additional steps are required. First, using a pre-compilation tool, we generate transparent proxies for trustlet instances. Proxies marshal the parameters and return values of the trustlet method call invocation, and encode them into messages exchanged with the TLR. Second, using a packaging tool, we bundle the manifest and the code of all trustlet classes into a single package. This package is signed so that the trustlet's identity and integrity can be verified during unseal.

6.3 Trustbox Management

After compiling and packaging an app, users can execute it on a smartphone. The TLR automatically manages the trustboxes created by the app, loads and instantiates trustlet code in the trustboxes, and routes method calls across worlds. All these tasks are performed by the trustbox layer (see Figure 4).

The trustbox layer handles three issues. First, trustbox requests submitted by different applications must be properly routed to the intended trustboxes. Second, trustlet instances residing in trustboxes must be isolated from each other. Third, the runtime state of trustboxes and respective trustlet instances must be consistent across world changes.

To satisfy these requirements, the TLR maintains dedicated *service threads* for servicing the trustboxes owned by each application. Each service thread uses *trustbox holder* data structures (see Figure 6) that contain a trustbox ID and a sandbox object. The sandbox object is a container for the state of a trustlet instance: it handles loading of trustlet classes into memory, enforces isolation across trustbox domains, and provides an interface for invoking the methods of the trustlet instance. In .NET, the sandbox object is implemented using `AppDomain` objects.

Based on these mechanisms, the trustbox layer handles the three main events of the trustbox lifecycle as follows:

1. Trustbox creation: When the application requests creation of a trustbox, the TLR library sends a request to the respective service thread running in the TLR-domain. This service thread then: (i) creates a new trustbox holder, containing a new ID and a sandbox; (ii) computes the hash of the trustlet code specified in the manifest; (iii) loads the trustlet classes into the sandbox; and (iv) creates an instance of the trustlet's main class.

2. Trustbox invocation: When the application calls the `Entrypoint` method, the TLR library creates a transparent proxy and returns it to the untrusted part of the application. When the untrusted application invokes a method on the proxy, it forwards this invocation request to the TLR-domain service thread. There, the request is decoded and the corresponding method is invoked on the corresponding trustlet instance.

3. Trustbox destruction: Finally, destroying a trustbox triggers a request to the service thread for releasing all resources associated with the respective trustbox holder. To persist state across instances, the developer can use he seal primitive to encrypt the relevant state and store it persistently.

6.4 Runtime Support

The runtime layer must handle the following three issues. First, because trustlets are managed rather than native code, the TLR must be able to interpret managed code. Second, because multiple applications can execute concurrently, the TLR runtime must be multitasked. Thus, each trustlet service thread is bound to the respective application process running in the OS-domain. Finally, because the runtime layer provides a message delivery to the trustbox layer, an appropriate interface must be devised for this service, preferably without requiring significant OS changes.

To address these issues, the runtime layer implements several mechanisms In the TLR-domain, the TLR runtime includes subcomponents that allow managed code to execute: a managed code interpreter, a type system enforcer, and a garbage collector. The TLR runtime waits for incoming requests from the OS-domain and interprets the managed code required for servicing them. To enable service threads to execute trustlets, service threads are implemented as managed code threads scheduled by the TLR runtime. A service thread is bound to an application process by annotating the service thread descriptor with the PID of the application process.

In the OS-domain, the OS coordinates the service of trustbox requests by the TLR runtime via a special TLR system call added to the OS. Because the TLR runtime is not aware of application processes' lifetimes, the OS instructs the TLR runtime to create and destroy service threads as application processes are created and killed, respectively.

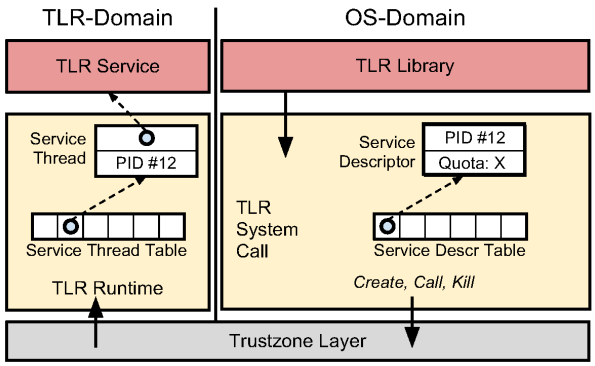


Figure 7. Details of the system layer.

The OS can exchange the following three messages with the TLR runtime:

1. Create service thread: The first time a TLR system call is issued by an application, no service thread for that process exists, and so it must be created. To keep track of the binding between service threads and application processes the OS maintains a descriptor table. To bind an application process, the OS issues a "create service thread" request to the TLR runtime. The TLR runtime creates a new service thread and annotates it with the caller's PID contained in the request.

2. Call service thread: This operation forwards the trustbox message requests received via the system call interface to the calling application process. After validating that the calling process is bound, the OS dispatches the request to the TLR runtime. The TLR runtime retrieves the trustbox message from the payload of the request, queues the message, and resumes the execution of the respective service thread. The service thread dequeues the request and processes it in the trustbox layer.

3. Kill service thread: This operation is issued by the OS to terminate a service thread and free its resources. This operation can be issued explicitly by an application or, alternatively, the OS periodically kills service threads of terminated applications. To kill a service thread, the OS simply sends a request to the TLR runtime and updates the local OS data structures.

6.5 Cross-world Communication and Interrupt Handling

Communication between the TLR-domain and the OS-domain is handled by the TLR's TrustZone layer (see Figure 8). This layer implements: (i) context save and restore for switching between normal and secure world; (ii) a simple message passing interface for exchanging data between the two worlds; and (iii) interrupt handling and re-routing if an interrupt arrives destined for a normal world device driver when the secure world is executing.

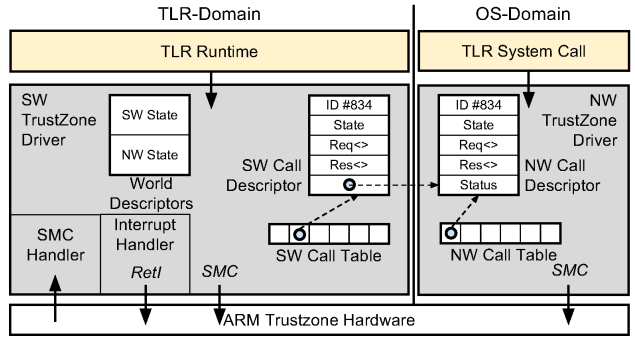


Figure 8. Details of the TrustZone layer.

To implement world switching, the secure world TrustZone driver maintains two *world descriptors*, each of which contains a snapshot of the processor state (CPU registers). The secure monitor implements world switching by detecting the direction of the transition, saving and restoring processor state from the corresponding world descriptors, and then toggling the processor mode bit. To ensure security, the world descriptors are stored in SW memory. A world switch can be activated in one of two ways: (i) a synchronous world-switch is caused when the *smc* instruction is issued, and (ii) an asynchronous world-switch is caused when a hardware interrupt fires in the secure world destined for a normal world device driver. Interrupts generated while the processor is executing in the NW do not cause a world switch.

To provide cross-world communication, the *secure monitor* implements data copying between the two worlds. This secure monitor must execute in the secure world so it can simultaneously address memory in both worlds.

One challenge is the support for interrupt handling and re-routing. Interrupts can be triggered while the processor is executing in the secure world, yet the normal-world OS retains control of system resources and contains all the I/O device drivers. Therefore, whenever an interrupt fires in the secure world, the secure world interrupt handler should switch worlds and hand control to the OS.

Because interrupts can fire in the middle of TLR runtime calls, the normal world OS must detect whether the secure world exiting is caused by TLR call completion or by an interrupt. If a TLR call completes, the OS can fetch the return parameters and schedule the application process. If any TLR calls are in progress, the OS must periodically switch worlds to continue executing those calls. When re-entering the secure world, the TLR decides if it should resume executing a prior service thread, or switch to a different service thread.

The TrustZone layer keeps track of the state of all ongoing TLR runtime calls. For each call, the TrustZone drivers maintain a descriptor that contains the call ID, input parameters, output parameters, and the current status of the call. Because this state must be accessible in both worlds, this descriptor table is replicated in both worlds (see Figure 8), and synchronized upon world transitions. Figure 9 shows the state machine for an individual TLR call.

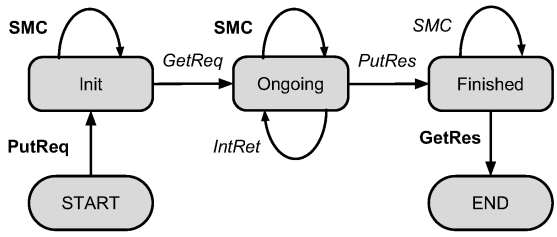


Figure 9. State machine of a TLR call as implemented in the TrustZone layer. Events in bold take place in the NW, and events in italic in the SW.

Init state: When the OS initiates a new TLR runtime request, the call status field is set to the `Init` value. On transition into the secure world, the input parameters are copied from the normal world and the call status is changed to `Ongoing`.

Ongoing state: This state indicates that a TLR call is in progress. This indicates to the OS that it should continue scheduling re-entry into the secure world until the call status changes to `Finished`.

Finished state: When a call is finished, the secure world TrustZone driver copies the output results to the normal world, and initiates a world switch.

6.6 Memory Management

To serve its memory needs, the TLR requires physical RAM dedicated to the secure world. The secure world bootloader uses low-level system firmware to statically allocate a fixed quantity of physically contiguous system RAM for secure world memory. TrustZone ensures that the normal world cannot address or otherwise access any of this RAM. The TLR then uses an internal memory allocator for all its internal state needs. In future work, we are considering allowing the OS to fine-tune how the TLR allocates memory to trustlets. Because the OS already controls allocation of system resources in the normal world, we can allow it to specify quotas to the TLR for trustlets, and balance those requests with fairness constraints.

6.7 System Boot

When an ARM SoC supports TrustZone, the processor initializes in secure mode and runs the secure world bootloader. This bootloader copies the TLR image into secure world memory and checks its integrity using a hash value signed with the platform ID. Next, the bootloader initializes the TLR runtime, and then performs the first world switch into the normal world, at which point the untrusted bootloader begins the standard OS boot sequence.

6.8 Platform Identity

The TLR requires a single, persistent per-device public key pair. This key pair constitutes a unique platform identity, similar to the role that the Endorsement Key (EK) plays for TPMs [22]. The device manufacturer must certify the

platform ID in the same way that TPM manufacturers issue EK certificates.

To protect the platform ID, the key pair is provisioned by the device manufacturer for each TrustZone-enabled smartphone and tablet. The result of provisioning is writing the platform ID into secure fuses (a write-once persistent memory) accessible only from the secure world. The TLR offers methods for trustlets to obtain the public part of the platform ID, and makes sure that the private key is never leaked.

To preserve users' privacy, TLR also borrows ideas from solutions developed for TPMs. First, is the notion of a privacy CA [32]: the device attests its identity to the trusted privacy CA, which in turn issues new platform credentials signed by the privacy CA. These new credentials improve anonymity because the device can now have multiple trusted identities, and can choose which one to use depending on which entity the trustlet is communication with. An alternative design borrows ideas from group signatures (similar to TPM Direct Anonymous Attestation [16]). The device could be equipped with a shared platform ID, and others could not distinguish between different devices sharing their IDs.

6.9 Seal and Unseal Primitives

Seal and Unseal use the device platform ID for cryptographic operations. Sealing a piece of data to a trustlet T running on device D consists of encrypting the data with a symmetric key, and then encrypting the symmetric key and T's hash with D's public key. Unsealing is the reverse: using D's private key to decrypt the symmetric key and the hash of T, validating the hash of the running trustlet matches, and finally decrypting the data using the symmetric key.

Sealed data is made persistent in encrypted form, but the TLR does not have direct access to the disk. Instead the TLR passes encrypted blobs to the disk managed by the untrusted OS. To prevent rollback attacks and enforce state continuity [31], we adopt the following defense. TrustZone-enabled devices use an eMMC storage controller that offers a security feature called replay-protected memory blocks (RPMB) (see [38]). The RPMB is a storage partition where operations from the secure world are authenticated and protected against rollback attacks. To achieve this, a key derived from the platform ID is also injected at manufacturing time into the eMMC controller. This key, shared between the secure world and the storage controller, is used to authenticate writes and a secure counter is also used to prevent rollbacks.

6.10 Trusted I/O Path

The TLR support for a trusted I/O path is limited to the storage functionality described above. TLR also adapts the same protocol to enable authenticated and replay-protected communication with external cloud services, even though all network communication is handled by the untrusted OS.

If we were to enable trusted I/O to the device's peripherals from the secure world, this would enable broader functionality within trustlets. For example, trustlets could directly interact with the device's sensors and directly display screen output. However, the cost of this approach would be a much larger TCB that incorporates device-specific drivers. In contrast, the current TLR design is generic – it works with all TrustZone-equipped devices.

Instead, we envision a future in which manufacturers equip their sensing hardware and other I/O peripherals with a secret key shared between the peripheral and the secure world. For input data, this allows sensors to gather and encrypt data before passing it to the untrusted OS, and only the TLR in the secure world can decrypt and use the data. For output, and it allows the I/O device to authenticate the data sent from the secure world before performing the output operation. This alternative allows the TLR to remain generic and eliminates driver code from its TCB.

7. Implementation

We implemented a TLR prototype for a TrustZone hardware testbed and leveraged existing open source software.

7.1 Hardware Testbed

Finding a TrustZone-compatible hardware testbed was not easy. Although the ARM TrustZone technology is prevalent in modern ARM-based SoCs, in most devices this technology is locked and cannot be used by application developers. We bypassed this limitation using a development board: NVidia's Tegra 250 Dev Kit [9].

The Tegra 250 board is equipped with dual-core 1 GHz Cortex A9 processors, 1 GB of RAM, 512 MB of flash memory, and multiple peripherals. Because the processor boots the OS in secure world, we can override the secure world environment and boot TLR. However, this board disallows flashing a unique key into the boards secure fuses, preventing us from implementing the platform ID in hardware. Moreover, the primary boot loader is closed-source, preventing us from modifying the secure world setup code in the first stage bootloader.

We address the first limitation by hard-coding the platform ID credentials in software. To address the second limitation, we boot the TLR using a customized second-level bootloader (u-boot [10]). This results in the unnecessary inclusion of the first-level bootloader in the TCB. These shortcomings are not fundamental and could be overcome with source access to a full-featured TrustZone board.

7.2 Software Implementation

In the OS-domain, we use Linux and Mono [4] v2.6.7 (an open source .NET framework implementation). We implement the TLR library for Mono and modify the Linux kernel in two ways: (i) implement the TLR system call, and (ii) change the kernel to eliminate all dependencies on resources only available in the secure world. In particular, we (i) modify the interrupt masks appropriately, (ii) disable certain cache control initialization code, and (iii) remove some

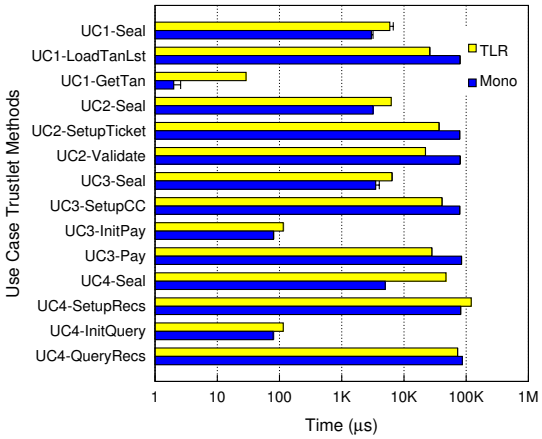


Figure 10. Execution time of trustlet methods from our use case prototypes.

processor initialization code. The goal of all these changes is to enable Linux to boot correctly in the normal world.

In the TLR-domain, our implementation covers the TLR and the bootloader. To build the TLR we leverage the codebase of NETMF v4.1 [6]. We borrow the CLR and PAL code from the NETMF and implement the remaining components in the HAL and application layers (see Figure 2). To customize the NETMF, we use the NETMF porting kit [12]. We also customize u-boot to initialize the TLR in the secure world, and to yield to the OS in the normal world.

8. Evaluation

This section presents a performance evaluation of the TLR and an analysis of its TCB size, programming complexity, and security.

8.1 Performance

We evaluate two sources of performance overhead for TLR applications when compared to standard .NET applications: (i) the overhead due to NETMF, a less efficient .NET stack than full-blown .NET, and (ii) the overhead due to executing additional TLR primitives.

Methodology. To evaluate the performance of trustlet code and TLR primitives, we run multiple experiments based on micro-benchmarks. We run our experiments in the hardware testbed described in Section 7. In all our measurements, we run ten trials and report the mean time and standard deviation.

We use our four use case applications as well as a specific benchmark suite. The use case prototypes allow us to measure the TLR performance with realistic apps. In total, we evaluate 14 methods: three for Use Case 1 (online banking transfers), three for Use Case 2 (mobile ticketing), four for Use Case 3 (mobile payments), and four for Use Case 4 (e-health application). For an in-depth description of these methods' roles see Appendix A. In addition, we implemented a specific benchmark suite to help discover the source of inefficiencies of the TLR runtime. This benchmark suite consists of five CPU-intensive programs: *MatrixMult*, which is a straightforward $O(n^3)$ matrix multiplication program; *Poly*, which computes the value of a 100-degree polynomial using floating point match; *Sudoku*, which is a sudoku solver; *CryptoRSA*, which performs RSA cryptographic operations (signatures, encryptions, and decryptions) using 1024-bit keys; and *CryptoAES*, which performs AES cryptographic operations (encryptions and decryptions) with 256-bit keys. We run each test both on the TLR and on an efficient .NET runtime (Mono) and then compute the difference.

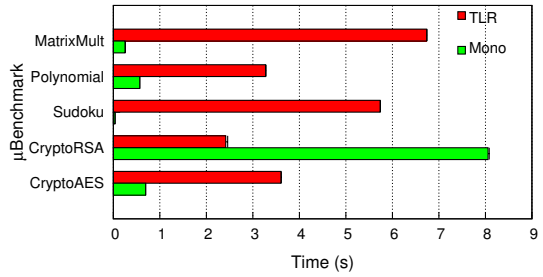


Figure 11. Performance of our benchmark suite executed on the TLR and on Mono.

Performance of trustlet code execution. Figure 11 plots the evaluation results of our use case prototypes, showing the execution time of all trustlet methods on the TLR and on Mono. The results show that Mono slightly outperforms the TLR: 57% of the methods execute on average $4.27\times$ faster in Mono, whereas 43% of the methods execute on average $2.34\times$ faster in the TLR. We find these numbers quite surprising, because we expected Mono to significantly outperform the TLR. Our expectation is justified by the fact that, in Mono, the trustlets' managed code is pre-compiled by a built-in jitter into native code, which runs on bare metal. In contrast, in the TLR, all the managed code is interpreted by the TLR, with the exception of certain libraries, such as the cryptographic library, which are implemented in native code.

To investigate why TLR's overhead is not more pronounced, we ran additional experiments using our specific benchmark suite. Figure 11 shows that, with the exception of *CryptoRSA*, all other programs of the benchmark run on average $54\times$ slower on the TLR. This difference is particularly large for programs whose code the TLR must entirely interpret, such as *Sudoku*, where the difference reaches a factor of 176. However, this difference is clearly inverted in the *CryptoRSA*, which runs $3.3\times$ faster on the TLR. This suggests that Mono's implementation of RSA is particularly inefficient.

We infer that, in the evaluation of our use case prototypes (see Figure 10), the TLR outperforms Mono when the trust-

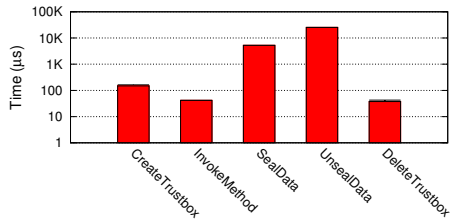


Figure 12. Baseline execution time of TLR primitives.

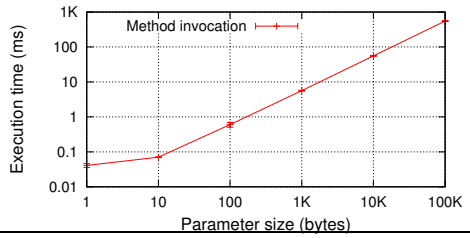


Figure 13. Performance of cross world method invocation varying the size of the method parameters.

let code makes heavy use of RSA. Since Mono's inefficiency is not fundamental, in general we should expect the TLR's performance slowdown to be more pronounced.

Performance of the TLR primitives. We implemented micro-benchmarks that stress each of the five operations related to the trustbox lifecycle: *trustbox creation*, *trustlet method invocation*, *data seal*, *data unseal*, and *trustbox deletion*. Because these operations' durations depend on their parameter sizes, we further measure the factors responsible for such variation, namely the cross world communication (relevant in trustbox creation and trustlet method invocation) and cryptographic computations (relevant in seal and unseal).

Figure 12 shows the minimum execution time of the TLR primitives. While seal and unseal take on average 15.2ms, the remaining primitives execute on average in 75.8μs. This difference is explained by the heavy use of cryptographic operations in seal and unseal. With the exception of delete trustbox, which executes in a constant time of 38μs, the duration of the TLR primitives depends on their input parameters, namely 1) the amount of data that needs to be transferred across worlds, or 2) the amount of data that needs to be encrypted or decrypted.

Figure 13 plots the execution time of method invocation while varying the size of the parameters transferred between worlds. The method execution time increases linearly at an approximate rate of 5.6ms/KB. This overhead is explained by the need to marshal the parameters and pass them by value to the TLR. In fact, the parameters cannot be transferred across worlds by reference (which would take a constant time) because TLR's internal data structures inherited from NETMF are incompatible with Mono's.

Lastly, to shed some light on the impact of cryptographic operations on the performance of TLR primitives, Figure 14 plots our evaluation results of seal and unseal as we vary the

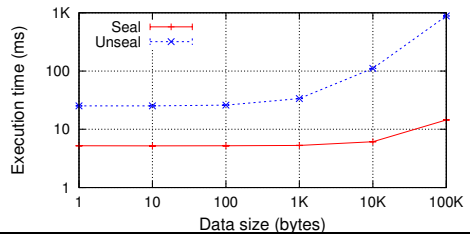


Figure 14. Performance of seal and unseal primitives varying the size of sealed and unsealed data, respectively.

size of the data to be sealed and the size of the envelope to be unsealed, respectively. Sealing 1KB takes 5.3ms and unsealing the same amount of data takes 33.6ms; these operations are dominated by the time complexity of the RSA algorithm used in their implementation. Seal and unseal are efficient, because the TLR makes use of the OpenSSL library, which implements cryptographic operations in native code.

8.2 TCB Size

To evaluate the TLR's TCB reduction, we compare its TCB size against that of two representative systems: TrustVisor v0.2 [29], and a setup based on Mono v2.6.7 and Linux v3.5.1 (Mono+Linux). While the former gives us an idea of the minimum TCB size achieved by a state-of-the-art system for securing native code apps, the latter points to the TCB size required by managed code .NET apps.

Table 1 presents the TCB sizes of the TLR, TrustVisor, and Mono+Linux. This table indicates which part of the codebase corresponds to native code (typically in C or C++) and which part to managed code (typically C#). The table also indicates which code belongs to the core versus to libraries of the system. To measure the TCB size, we use the metric of *lines of code* (LOC), which counts all lines of a system's codebase (including comments and empty lines).

Comparing TLR with TrustVisor, we see that TLR is approximately 6 times larger than TrustVisor: the TLR and TrustVisor comprise, respectively, 152.7 KLOC and 25.3 KLOC. This difference can be explained by the fact that, unlike the TLR, TrustVisor neither needs to implement a managed code runtime nor to include managed code libraries.

Comparing the TLR with the Mono+Linux setup, we see that the TLR drastically shrinks the TCB. While the TCB size of Mono+Linux is 11.9 MLOC [1], the TLR's is 152.7 KLOC, i.e., approximately 78 times smaller. The TLR cuts down the TCB size due to its design, which limits the services offered to trustlets and blends the OS and runtime functionality into a compact single system.

8.3 Programming Complexity

Since assessing the complexity of programming applications for the TLR is difficult, we mostly make an account of our experience with building the use case prototypes and bench-

[1] Appendix B explains how this number was obtained.

Code (LOC)	TrustVisor	TLR	Mono+Linux
Managed Libs	N/A	19.9K (C#)	3,305.3K (C#)
Native Libs	18.1K (C)	80.5K (C++)	1,308.6K (C)
Native Core	7.2K (C)	52.3K (C++)	7,302.9K (C)
Total	25.3K	152.7K	11,916.8K

Table 1. TCB size of the TLR, TrustVisor, and Mono+Linux.

Use Case	Code Size (LOC)	# Methods
Online banking	179	3
Mobile ticketing	450	3
Mobile payments	754	4
E-health app	974	4

Table 2. Programming complexity of the use case prototypes measured in code size and number of methods.

mark programs. In our experience, programming with TLR apps is relatively easy. Once we sketched the security protocols for the four use cases (see Appendix A), programming their respective trustlets was done by a grad student in 3.5 days. Table 1 shows the codebase size and the number of methods implemented by each trustlet. These numbers show that the average code size is relatively small, consisting of 590 LOC in C#, and the trustlet interfaces are simple, consisting of three to four methods. Although real world applications would likely demand a bigger programming effort, our intuition is that such an effort is comparable to app development for standard .NET.

8.4 Security Analysis

The attack surface of the TLR comprises the `smc` instruction exposed to the OS and the managed code interface exposed to the trustlets. The `smc` interface is relatively narrow, which limits the exposure of code vulnerabilities. The managed code interface offers a larger attack surface: an attacker could exploit a bug in the TLR runtime by injecting carefully crafted code sequences in the trustlet code.

TLR can only provide limited protection against physical attacks: an attacker with the capability of tampering with the hardware could disable the TrustZone protections and bypass TLR. However, such attacks require a high degree of sophistication: since the core of the system (the SoC) is packaged in a single die, an attacker would need to break into the SoC and tamper with the TrustZone hardware.

9. Related Work

TLR borrows many concepts from previous work on building trusted execution environments (TEE). Some previous systems such as XOM [26] and AEGIS [36] require specific hardware that is not yet available on commodity processors.

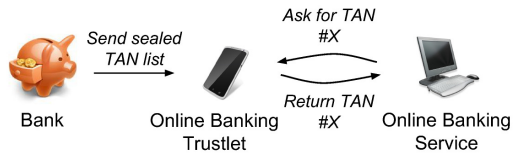
For x86 platforms, a typical TEE system is based upon a Trusted Platform Module (TPM) and a trusted kernel [21]. The TPM is used to bootstrap trust in the system, and the trusted kernel provides runtime state protection for trusted application code. Because the security of such systems depends on the correctness of the trusted kernel, a lot of effort has been put into reducing the TCB size. In the first generation of these systems, the trusted kernel consisted of a full blown Virtual Machine Monitor (VMM) as in Terra [21], Proxos [37], and Overshadow [18]. Researchers also proposed systems based on microkernels, as in Nizza [23] and Nova [35]. Currently, the state of the art consists of tiny kernels implemented using late launch technology as in Flicker [28], hardware virtualization as in TrustVisor [29], and the handler code of the System Management Mode (SMM) as in SICE [15]. Reducing the TCB size, however, makes these systems difficult to program. In contrast, TLR aims to provide richer programming abstractions while keeping the TCB small.

For ARM TrustZone, the closest piece of work to TLR is Nokia's OnBoard Credentials (ObC) [25]. ObC can execute programs written in a modified variant of the LUA scripting language running in an isolated environment protected by TrustZone. TLR's primary benefit over ObC is bringing the trustbox and trustlet concepts to a mature managed-code environment, as well as providing a richer set of abstractions. Developers will find trusted computing primitives easier to program with .NET because it offers the productivity benefits of modern high-level languages, such as strong typing and garbage collection, to application developers.

Other previous systems leverage ARM TrustZone to implement specific security services. One system uses TrustZone to implement trusted sensors [27], which enable mobile apps to obtain guarantees of authenticity and integrity of sensor readings. In the commercial sphere, Proxama uses ARM TrustZone to enable secure mobile payments [7].

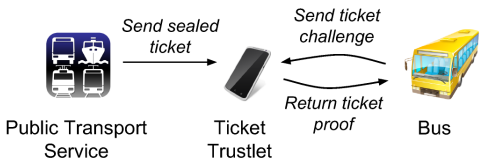
There is a large body of work on improving the security of OSes and mobile apps. In particular, researchers have paid considerable attention to protecting personal user data (e.g., address book, user photos, password information, GPS location) and preventing its unauthorized access and leakage by proposing novel techniques, such as new access control mechanisms [33] and information flow analysis [20]. This work, however, is complementary to ours: TLR does not need to trust the OS unlike this previous work.

Another research area uses privilege separation for partitioning an application into security-sensitive and security-insensitive components. These systems expose a partitioning interface at the level of the programming language, and enforce this separation by using a runtime [30], or the OS itself [17]. Such approaches still depend on a large TCB, which includes the OS and the runtime. TLR offers a coarser-grained privilege separation by compartmentalizing an application, but has a smaller TCB.



Setup:	1. Bank→Device: $\text{seal}_{T,D}([\text{TANLST}])$
Query:	1. Svc→Device: *index* 2. Device→Svc: $\text{TANLST}(index)$

Figure 15. Use Case 1: Online banking transfers.



Purchase:	1. PTS→Device: $\text{seal}_{T,D}([tinfo, KT, \text{cert}_{PTS}(KT)])$
Validate:	1. Bus→Device: n_b, *time* 2. Device→Bus: $[\{n_b\}_{KT}, \text{cert}_{PTS}(KT)] \mid \text{fail}$

Figure 16. Use Case 2: Mobile ticketing.

10. Conclusions

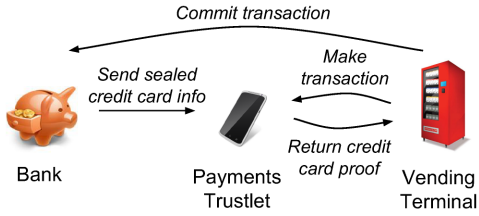
This paper presents the design, implementation, and evaluation of the Trusted Language Runtime (TLR), a system for running security-sensitive applications on mobile devices. TLR offers programming primitives that allow small application components to execute within a trusted environment isolated from the operating system and other applications. The TLR protects the integrity and confidentiality of application code and data within the trusted environment. TLR provides these features using the ARM TrustZone hardware support for trusted computing found in ARM SoCs. Our evaluation shows that the TLR achieves a significant reduction in the TCB of mobile apps with an acceptable performance cost.

Acknowledgments

We are grateful to the anonymous reviewers for their comments. This work was partly supported by national funds through FCT (Fundação para a Ciência e Tecnologia) under project PEst-OE/EEI/LA0021/2013, and by PCAS project (co-financed by the European Commission through the contract no. 610713).

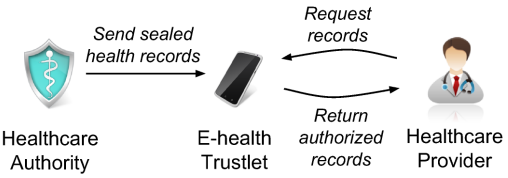
Appendix A: Use Case Protocols

This section presents a brief description of the security protocols that implement the use cases presented in Section 5.



Setup:	1. Bank→Device: $\text{seal}_{T,D}([CC, PIN, KT, KT_{ID}, \text{cert}_B(KT), \text{cert}_M(VT)])$
Pay:	1. Device→VT: n_d 2. VT→Device: VT_{ID}, *amount*, *date*, n_d, n_v, $h(PIN,n_d,amount)$, σ_{VT}, $\text{cert}_M(VT)$ 3. Device→VT: $[VT_{ID}$, *amount*, *date*, KT_{ID}, n_v, σ_{KT}, $\text{cert}_B(KT)] \mid$ abort
Commit:	1. VT→Bank: VT_{ID}, *amount*, *date*, n_v, KT_{ID}, σ_{KT}

Figure 17. Use Case 3: Mobile payments.



Deploy:	1. HA→Device: $\text{seal}_{T,D}([[R_0, \ldots, R_n], \text{ACPOL}, KT, \text{cert}_{HA}(KT)])$
Query:	1. Device→HP: n_d 2. HP→Device: $[R_k^{ID}], n_h, \{n_d\}_{KH}, \text{cert}_{HA}(KH)$ 3. Device→HP: $[\langle R_i \ldots R_j \rangle_K$, $\{K, n_h\}_{KT}, \text{cert}_{HA}(KT)] \mid$ fail

Figure 18. Use Case 4: E-health application.

To represent cryptographic operations, we use following notation. For asymmetric cryptography, K and K^P denote private and public keys, respectively. For symmetric keys, we drop the superscript. Notation $\langle x \rangle_K$ indicates data x encrypted with key K, and $\{y\}_K$ indicates data y signed with key K. We represent nonces as n.

The protocols implementing each of our four use cases are represented in Figures 15–18, respectively.

Appendix B: TCB of Mono+Linux Setup

The Linux+Mono TCB includes part of the Linux kernel (6.9 MLOC), Mono's runtime (471 KLOC), native code libraries, such as Glib2 (1.3 MLOC), and managed code libraries shipped with Mono (3.3 MLOC). Since real Linux deployments do not include all device drivers shipped in the kernel, to avoid reporting an artificially bloated Linux kernel, we conservatively exclude the drivers' source code.

References

[1] Android. http://www.android.com.

[2] Dalvik VM Internals. https://sites.google.com/site/io/dalvik-vm-internals.

[3] Apple iOS 6. http://www.apple.com/ios.

[4] Mono. http://www.mono-project.com/Main_Page/.

[5] Common Language Runtime (CLR), . http://msdn.microsoft.com/en-us/library/8bs2ecf4.aspx.

[6] .NET Micro Framework, . http://www.microsoft.com/netmf/default.mspx.

[7] Proxama. http://www.proxama.com/products-and-services/trustzone.

[8] Transaction authentication number. http://www.wikipedia.org/wiki/Transaction_authentication_number.

[9] Tegra 250 Dev Board. https://developer.nvidia.com/tegra-250-development-board-features/.

[10] U-boot Bootloader. http://www.denx.de/en/News/WebHome/.

[11] Microsoft Windows 8. http://windows.microsoft.com/en-us/windows-8/meet.

[12] Porting the .NET Micro Framework. Microsoft Technical White Paper, 2007. http://msdn.microsoft.com/en-us/netframework/bb267253.aspx.

[13] Understanding .NET Micro Framework Architecture, 2010. http://msdn.microsoft.com/en-us/library/cc533001.aspx.

[14] ARM. ARM Security Technology – Building a Secure System using TrustZone Technology. ARM Technical White Paper, 2009. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[15] A. M. Azab, P. Ning, and X. Zhang. SICE: A Hardware-Level Strongly Isolated Computing Environment for x86 Multi-core Platforms. In *Proc. of CCS*, 2011.

[16] Brickell, Ernie and Camenisch, Jan and Chen, Liqun. Direct Anonymous Attestation. In *Proc. of CCS*, 2004.

[17] D. Brumley and D. Song. Privtrans: automatically partitioning programs for privilege separation. In *Proc. of USENIX Security '04*, 2004.

[18] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proc. of ASPLOS*, 2008.

[19] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. of SOSP '07*, 2007.

[20] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc of OSDI'10*, 2010.

[21] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proc. of SOSP*, 2003.

[22] T. C. Group. TPM Main Specification Level 2 Version 1.2, Revision 130, 2006.

[23] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza Secure-System Architecture. *CollaborateCom*, 2005.

[24] M. Hypponen. Malware goes Mobile. *Scientific American*, November 2006.

[25] K. Kostiainen, J.-E. Ekberg, N. Asokan, and A. Rantala. Onboard Credentials with Open Provisioning. In *Proc. of ASIACCS*, 2009.

[26] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proc. of ASPLOS*, 2000.

[27] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software Abstractions for Trusted Sensors. In *Proc. of Mobisys*, 2012.

[28] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proc. of EuroSys*, 2008.

[29] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proc. of IEEE S&P*, 2010.

[30] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proc. of POPL '99*, 1999.

[31] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. Mccune. Memoir: Practical State Continuity for Protected Modules. In *Proc. of IEEE S&P*, 2011.

[32] PrivacyCA. PrivacyCA. http://privacyca.com.

[33] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *Proc. of IEEE S&P*, 2012.

[34] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Trusted Language Runtime (TLR): Enabling Trusted Applications on Smartphones. In *Proc. of HotMobile*, 2011.

[35] U. Steinberg and B. Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Eurosys*, 2010.

[36] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proc. of ICS*, 2003.

[37] R. Ta-Min, L. Litty, and D. Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Congurable. In *Proc. of OSDI*, 2006.

[38] V. Tsai. eMMC v4.41 and v4.5. http://www.jedec.org/sites/default/files/Victor_Tsai.pdf.