Trusted Language Runtime (TLR): Enabling Trusted Applications on Smartphones

Nuno Santos[†], Himanshu Raj[‡], Stefan Saroiu[‡], and Alec Wolman[‡]
[†]MPI-SWS and [‡]Microsoft Research

ABSTRACT

Despite their popularity, today's smartphones do not yet offer environments for building and running trusted applications. At the same time, current systems designed for traditional desktop or server machines to enable trusted applications are either too heavyweight for smartphones or too difficult to program. This paper presents the Trusted Language Runtime (TLR), a system for developing and running trusted applications on a smartphone. The TLR is lightweight because 1) it makes use of ARM TrustZone, hardware support that offers rich trusted computing primitives, and 2) it leverages the .NET MicroFramework, a language runtime for embedded and resource-constrained devices. The TLR is easy to program because .NET offers the productivity benefits of modern high-level languages, such as strong typing and garbage collection, to application developers.

1. INTRODUCTION

The need for trusted applications on smartphones is greater than ever. As smartphones become the *de facto* personal computing device, people are storing more and more sensitive and personal information on their phones. Unfortunately, the value of this information is starting to make smartphones an attractive target for attacks, including third-party applications with questionable practices [9] as well as outright malware [11]. Even more alarming, researchers have demonstrated the ease with which today's smartphones can be subjected to rootkits [6].

For desktop and server machines, researchers have developed systems that offer environments for running trusted applications [10, 13, 12]. These systems aim to protect a trusted application's code and/or data by guaranteeing its *integrity* (i.e., the trusted application cannot be modified by malicious code) and *confidentiality* (i.e., the trusted application can protect a secret from malicious code). To offer these guarantees, such systems often use the hardware support for trustworthy computing commonly found on x86 platforms, such as the Trusted Platform Module (TPM) and Intel's Trusted Execution Technology (TXT).

However, it is time to rethink the design of these trusted runtime environments in the context of mobile devices. Current approaches do not fit the needs of today's mobile landscape for the following three reasons. First, the vast majority of mobile handhelds are ARM-based instead of x86-based. There is no discrete TPM chip available on these mobile platforms yet. However, ARM provides a hardware solution for trustworthy computing known as the ARM TrustZone [4]. ARM TrustZone provides a trusted execution environment on the CPU cores, with hardware support for memory protection of the trusted environment, flexible control over interrupt delivery to the trusted environment, and the full power of the CPU for cryptographic operations. This can lead to simpler and more powerful software with a smaller trusted computing base (TCB).

Second, unlike desktops, mobile devices are often resource constrained. Any system for running trusted applications on a smartphone must be lightweight. While previous hypervisor-based solutions offer isolation from malicious code [10, 12], such solutions may be too heavyweight for a smartphone considering the impact on memory use, performance, and energy consumption.

Finally, the popularity of smartphones and the emergence of "app stores" has created a cottage industry where hundreds of thousands of developers are writing a highly diverse set of mobile applications. Any system that targets such a large number of third-party developers with varying skills and backgrounds must offer easy-to-use, rich programming abstractions. In contrast, current lightweight approaches (i.e., those not based on virtualization) typically offer runtimes that lack libraries and rich programming support [13].

This paper presents the Trusted Language Runtime (TLR) architecture that facilitates the development of trusted applications on a smartphone platform. TLR offers two abstractions to mobile developers: a *trustbox* and a *trustlet*. A trustbox is a runtime environment that protects the integrity and confidentiality of code and data. Code and data running inside trustbox cannot be read nor modified by any code running outside the trustbox. A trustlet is the portion of an application that runs inside trustbox. A trustlet is a .NET-based class whose interface defines the data that can flow in or out of the trustbox. With TLR, programmers write trusted applications in .NET and specify which *parts of the application* handle sensitive data and, thus, must run inside the trustbox. The developer places these parts in a trustlet class, and the TLR provides all the support needed

HotMobile 2011, March 1–2, 2011, Phoenix, AZ, USA.

to run them in a trustbox. By splitting an application into a small trusted component (a trustlet) and a large untrusted one, the application's attack surface is reduced. Any exploitable bug in the untrusted component does not affect the trusted component's integrity and confidentiality.

To offer its guarantees, the TLR leverages the ARM TrustZone memory protection and interrupt delivery control mechanisms, thus reducing the size of its TCB. In addition to presenting the TLR design, we present an implementation based on the ARM emulator [5]. While TrustZone technology is available for many ARM chips, it is often left disabled by the firmware. Finally, our paper discusses some of the open issues with building trusted computing runtimes on a smartphone that our experience with TLR has unraveled.

2. THE TLR ARCHITECTURE

TLR is motivated by three high-level design goals:

1. Security. The trusted computing base (TCB) of the TLR should not include the operating system and most application code running on the smartphone. None of this untrusted code should be able to interfere with or even inspect trusted code running inside the TLR.
2. Ease of programmability. The effort to build trusted applications with the TLR should be low. Programming in 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. Running the TLR should not require a redesign of today's legacy operating systems or other legacy software running on the smartphone.

2.1 High-Level Design

TLR provides two execution environments: an untrusted one where the smartphone's OS and most application software runs, and a trusted one. The code running in the trusted environment is isolated from any code running in the untrusted "world". Untrusted code cannot inspect or modify the trusted code. To enable interaction, the TLR provides a secure communication channel between the two environments. The TLR ensures both integrity and confidentiality for code and data inside the trusted environment.

The trusted world offers a language runtime with minimal library support: in our implementation we offer the .NET Micro Framework [1]. We find that a resource-constrained runtime environment offers enough flexibility to accommodate the trusted computing needs of mobile applications while keeping the TCB of the TLR small. With the TLR, a developer needs to partition a mobile application in two components: a small-sized trusted component that can run on the resource-constrained runtime of the trusted world, and a large-sized untrusted component that implements most of the application's functionality. This partitioning process is similar in spirit to previous work on privilege separation [7] and partitioning of applications for improved security in distributed systems [8].

Figure 1 illustrates the TLR's high-level design. To meet our goals, we provide four primitives in designing the TLR:

1. Trustbox. A trustbox is an isolation environment that protects the integrity and confidentiality of any code running inside, as well as its state. The smartphone's OS (or any untrusted application code) cannot tamper with the code running in a trustbox nor inspect its state.

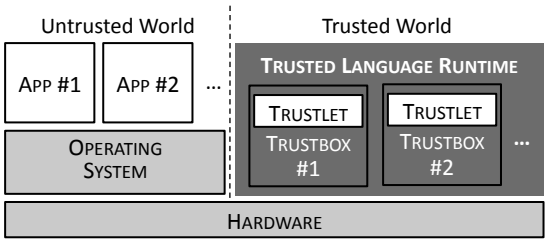

Figure 1: High-level architecture of TLR.

2. Trustlet. A trustlet is a class within an application that runs 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. This identity is used to authenticate the platform and to protect (using encryption) any trusted application and data deployed to the platform. Our implementation uses a public/private key pair. Access to the private key is provided solely to the TLR which never reveals it to anyone.
4. Seal/Unseal data. These abstractions serve two roles: (1) a trustlet can persist state across reboots, and (2) a remote trusted party (i.e., a trusted server) can communicate with a trustlet securely. Sealing data means that data is encrypted and bound to a particular trustlet and platform before it is released to the untrusted world. The TLR unseals data only to the same trustlet on the same platform that originally sealed it. The trustlet's identity is based on a secure hash of its code.

2.2 Typical Development Scenario

To build a trusted mobile application with the TLR, a developer typically performs the following three steps:

1. Determine which part of an application handles sensitive data. To define a trustlet, the developer identifies the application's sensitive data, and separates the program logic that needs to operate on this data into the trustlet. The developer carefully defines the public interface to the trustlet's main class, as this interface controls what data crosses the boundary between the trusted and untrusted worlds. A trustlet may use many helper classes, and in fact may even consist of multiple assemblies, yet there is only one class that defines the trustlet's boundary. Once all necessary classes are compiled into assemblies, the developer runs a TLR post-compilation tool for creating a package that contains the closure of the assemblies, and a manifest.
2. Seal the sensitive data by binding it to the trustlet. Although any application developer can encrypt data without the help of the TLR, the TLR provides special encryption primitives called *seal* and *unseal* [15]. These operations allow a developer to encrypt (seal) an object such that it can only be decrypted (unsealed) on a specific smartphone platform, by a specific trustlet. Both the platform and trustlet identities are specified at seal time: a unique public/private keypair for the platform, and a secure hash (e.g., SHA-1) of the trustlet assemblies. To recover sealed data, the TLR decrypts the sealed data using the platform key,

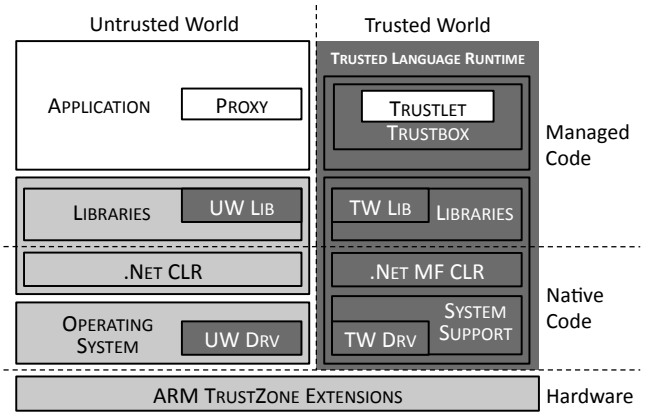

Figure 2: Component diagram of the entire system with a TLR: the darkest shade shows the TLR components, the light gray shows the smartphone's standard system components, and the application components are in white.

and checks that the hash of the trustlet requesting to unseal matches the hash of the trustlet that originally sealed the data. This mechanism allows the application to store trustlet data across multiple sessions in persistent storage, and it allows external parties (e.g. a trusted service) to ensure that sealed data can only be accessed on platforms it trusts.

3. Deploy trustlet and sealed data to the smartphone and run them inside of a trustbox. To ensure that the trustlet state is protected at runtime, the developer instantiates a trustbox by providing the trustlet's manifest. At this point, the TLR loads the trustlet's assemblies and creates an instance of the trustlet main class. The resulting object constitutes the runtime state of the trustlet until the application destroys the trustbox. To allow the application to interact with the trustlet, the application requests that the TLR create a special *entrypoint* object, which is a transparent proxy to the trustlet interface. Whenever the application invokes methods on the entrypoint, the TLR transparently forwards these calls to the trustlet main object.

2.3 The Lack of Remote Attestation

One common primitive used to build trusted applications is "remote attestation" [10]: the ability of a computer to attest its own software configuration to a remote party. Although the TLR could provide a remote attestation mechanism, we chose to omit this from our current design to reduce the overall system complexity. TLR is aimed at smartphones and we envision a usage model where smartphone manufacturers initialize and ship their devices with a trusted (uncompromised) TLR implementation. The manufacturer signs this TLR configuration, and the boot process performs signature verification. As long as the TLR implementation is not compromised, it then protects the integrity and confidentiality of data and code running in a trustbox. With this model, we find the TLR offers adequate trust properties even without remote attestation.

3. THE TLR IMPLEMENTATION

Figure 2 shows a detailed view of the TLR's various components. To meet our goals of an easy-to-use programming environment along with a relatively small TCB, we use the .NET MicroFramework (MF) [1] as the language runtime for the trusted world. This allows applications to be built with modern programming languages, such as C#, that improve programmer productivity through features such as strong type checking and garbage collection. The .NET MF is a much smaller version of the standard .NET Framework, and is specifically designed for resource constrained devices. The .NET MF achieves its small size by eliminating: 1) the JIT compiler, 2) some advanced language features such as generics and multidimensional arrays, and 3) some of the more complex portions of the .NET class libraries. In addition to its smaller codebase, the .NET MF implements the minimal system support needed to run on "bare-metal" without the need of an OS. The main drawback of using the .NET MF is the performance hit due to interpreting the managed code instructions rather than using a JIT compiler.

Beyond the .NET MF, we further reduce the TCB of the trusted world by eliminating all support for I/O. This has two effects. First, it greatly reduces the size of the .NET class libraries. For example, the GUI libraries are no longer needed, reducing the size of class libraries needed by applications. Second, it also eliminates the need for I/O device drivers, and these drivers typically constitute the largest fraction of the TCB in modern operating systems.

To enable communication between the untrusted and trusted worlds, we provide a *secure procedure call (SPC)* mechanism. Four components in the above figure are needed to implement this: the UWLib and TWLib libraries at the language runtime level, and the UWDrv and TWDrv communication drivers at the system support level. The drivers are responsible for implementing the ARM TrustZone context switch in/out of "secure" mode, and the library components are responsible for handing off the appropriate input and output data to the drivers.

To enable application partitioning, the TLR implements the *trustlet* and *trustbox* classes. The trustlet defines the self-contained application code which is to be run inside the isolated trustbox.

We built our prototype implementation of the TLR using an ARM emulator [5]. We chose emulation because the firmware for our development board disables the ARM TrustZone features on the CPU. As a result, we can only implement the SPC mechanism using the emulator with its built-in support for TrustZone. To make the .NET MF v4.1 work inside the emulator, we used the .NET MF porting kit [3] to customize the .NET MF to the ARM Cortex A8 instruction set.

3.1 Programming Model

To build an application that uses the TLR, at a minimum the programmer must implement two components: 1) the main trustlet class that defines the public interface between the trusted and untrusted world, and 2) the code that manages the lifetime of the trustbox. The third programming feature provided by the TLR is optional: applications that need to persist state across different runs of the application can use the seal/unseal facility. We now present the details of how a programmer uses each of these three features. Figures 3, 4, and 5 provide small code samples that demonstrate the use of these APIs.

To implement the trustlet main class, the developer simply defines a new class that inherits from the `Trustlet` class,

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

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

   public override void Init() {}

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

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

   public override void Finish() {}
}
```

Figure 3: Implementation of a trustlet.

and that implements the `IEntrypoint` interface. Any public method defined in this class enables data to cross the barrier between the trusted and untrusted worlds. All `Trustlet` objects also provide two methods, `Init` and `Finish`, which are called when the trustbox is created and destroyed. The developer can override these methods to perform any application specific operations during these events. The strongly-typed nature of the .NET runtime makes it simple to reason about what kinds of data is crossing this barrier. This is important because the programmer must be careful not to let any of the sensitive data protected by the trustbox leak out into the untrusted world.

To manage the lifetime of a trustbox, the TLR provides three methods implemented by the `Trustbox` class. To create a trustbox, an application invokes the `Create` method, which takes as input the trustlet manifest, and creates a new trustbox dedicated to hosting the trustlet. The trustbox reference returned by `Create` can then be used by the application to obtain a transparent proxy to the trustlet entrypoint, by calling the `Entrypoint` method. The transparent proxy is needed to ensure that all calls into the trustlet are routed through the secure procedure call mechanism. Finally, when the application wishes to terminate, it invokes the `Destroy` method to clean up the runtime state of the trustlet.

Finally, the TLR provides the `Seal` and `Unseal` operations. Sealing is a form of encryption that binds the encrypted data to a specific trustlet running on a specific system. To accomplish this, each unique smartphone has a public/private keypair we call the platform id. This platform id is used in combination with the secure hash of the trustlet codebase to identify a particular instance of a trustlet. `Seal` takes three inputs: 1) the object to be sealed, 2) the public key of the target platform id, and 3) a secure hash of the target trustlet. `Seal` returns an envelope

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

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

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

// run online transaction with the bank

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

Figure 4: Calling services on a trustbox.

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

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

// send the sealed list to the customer
```

Figure 5: Trusted service sending confidential data to a trustlet.

which consists of the serialized object concatenated with the trusted hash value, encrypted using the platform id public key. `Unseal` decrypts the envelope (which can only be done using with the platform id private key), and then returns the original data only if the currently running trustlet hash value matches the envelope hash value. As a result, unseal ensures the trustlet identity and integrity.

The current status of our TLR implementation supports all aspects of the programming model described above except for the `Seal` and `Unseal` operations.

3.1.1 A Sample Application

To illustrate how these constructs work together, consider our example shown in Figures 3, 4, and 5. To improve security, banking services typically rely on multiple mechanisms for authenticating their customers during online transactions. In addition to the customer password, banks normally issue a list of *Transaction Authentication Numbers* (TANs) [2], each of which constitutes a one-time password for authorizing a bank transfer. The bank sends a list of TANs to each customer, and whenever the customer performs an online transfer, the bank specifies an index into the TAN list and asks for the TAN associated with that index. Today, banks usually write down the TAN list on a plastic card, and send that card to the customer over an out of band channel (e.g., physical mail).

Banks could take advantage of the TLR to build an application that can protect the confidentiality of the TAN list when stored on a customer's smartphone. To accomplish this, the bank would create a trustlet (whose code is trusted by the bank), and seal the TAN list on a per-customer basis so that it can only be unsealed by the bank's trustlet running on that specific customer's phone. The code running within the trustlet can access the TAN list, retrieve the appropriate TAN number, and pass it to the untrusted environment to be sent to the remote bank server. The trustlet and the remote server communicate using SSL to protect the confidentiality of the data while in flight.

3.2 Runtime Operation

In this section, we describe: 1) the boot process, 2) how secure procedure calls are implemented, and 3) trustbox creation and termination.

3.2.1 System Boot

When an ARM CPU supports the TrustZone feature, the processor boots in secure mode and runs the secure bootloader. Our bootloader is responsible for loading the TLR image into memory and checking its integrity. Because TrustZones provide hardware support for memory isolation, the TLR runtime lives in the address space of the trusted world and cannot be accessed from the untrusted world. Next, the secure bootloader hands off to the initialization code within the TLR runtime. After the TLR initialization code finishes, it uses a mode switch instruction to exit secure mode, at which point the untrusted world bootloader is invoked and the standard OS boot sequence is then executed.

3.2.2 Secure Procedure Call

To enforce the separation of an application between the trusted and untrusted worlds, the TLR needs a mechanism to communicate across that boundary. The TLR provides a *secure procedure call* (SPC) which enables a secure communication channel between the two worlds.

To implement SPC, the TLR uses two kernel-mode drivers: the untrusted world driver (UWDrv) and the trusted world driver (TWDrv) shown in Figure 2. These drivers use the ARM TrustZone instructions to enable switching in and out of secure mode. When the UWDrv receives a SPC request, the driver executes the `smc` special instruction which raises a processor exception. This causes the processor to enter a special privileged mode called *monitor mode*, and then it jumps to the appropriate exception handler which is implemented by the TWDrv driver. This handler implements the context switch by saving the processor state from the untrusted world, and restoring the trusted world processor state. Next, the processor leaves monitor mode, and the TWDrv forwards the request up to the trusted world library (TWLib), which calls a managed code handler to service the SPC request. When this handler finishes, the system returns to the untrusted world using the same mechanism. The drivers are also responsible for marshaling the arguments and return values.

Our current implementation of SPC only runs on the ARM emulator target, and we have not yet optimized our implementation. In particular, we currently only support passing data (the SPC arguments and return values) between worlds through processor registers rather than through shared memory.

3.2.3 Trustbox Creation and Termination

When the application requests the creation of a trustbox, the TLR performs the following steps: 1) it computes the hash of the trustlet assemblies specified by the manifest, 2) it creates a new sandboxed environment inside the trusted world by using a .NET `AppDomain` container – this is how multiple trustlets that live in the trusted world are isolated from each other, 3) it loads the trustlet assemblies into the AppDomain, and 4) it creates an instance of the trustlet's main class. After these operations succeed, the TLR returns a reference to the trustbox, which can be used for future interactions with the trustbox.

When the application calls the `Entrypoint` method on the trustbox reference, the untrusted world library (UWLib) creates a transparent proxy and returns it to the untrusted part of the application. After this step, whenever the untrusted application invokes a method on the proxy, the UWLib forwards this invocation to the appropriate trustlet inside the trustbox, using the SPC mechanism described above. This invocation is fully transparently to the application, and the object state is preserved across these calls.

To destroy a trustbox, the TLR runtime simply deletes the `AppDomain` container of the trustbox thereby freeing all its resources, and discarding its internal state. If the developer wants to save any state persistently across instances, she can implement a trustlet method for sealing the relevant state, and having the application store it persistently.

4. OPEN ISSUES

In this section, we summarize some of the design and implementation challenges we have not yet addressed.

4.1 Memory Management

The TLR is responsible for managing its internal state as well as the memory state of the trustboxes. In our current design, we statically allocate a fixed set of memory pages to the trusted world when the platform boots. However, the memory needs of applications running inside trustboxes may vary over time, and possibly even exceed the capacity that was allocated at boot. To support this, in the future we may need a small OS kernel running in the trusted world to enable demand paging to the flash memory typically found in smartphones. Encryption must be used to ensure that paging does not compromise the integrity and confidentiality benefits that the TLR currently provides.

4.2 I/O for the Trusted World

Our current design of the TLR does not support I/O directly from the trusted world. Removing this limitation would offer two significant benefits. First, misbehavior by the operating system running in the untrusted world can cause denial of service for I/O operations requested by a trustlet, and incorporating support for basic I/O (such as use of the file system or the network) into the trusted world would eliminate this problem. Second, a good source of entropy is fundamental for generating random numbers which are required by cryptographic operations within the trustlets. A common way to collect entropy is to use hardware I/O sources, and therefore offering I/O inside the trusted world would enable this.

Even in the absence of I/O support, TLR enables secure storage and networking. When data is to be stored or sent remotely, trustlets encrypt the data before relaying it to the untrusted world, where the encrypted data can then be passed to the disk or the network. Although the untrusted world could misbehave, such attacks cannot compromise the data confidentiality.

4.3 Performance Overhead

Our use of the .NET MicroFramework offers significant benefits in terms of a reduced TCB, but it comes at a noticeable cost to performance. The primary performance overhead arises from the code being interpreted rather than compiled to native instructions using a JIT compiler. One area for future research is to understand if limited JIT function-

ality can provide a significant boost to performance without bloating the TCB. Another possible avenue for the future is to offload the responsibility for code generation to a trusted JIT service in the cloud, to keep the TLR code base small.

4.4 Trustlet Debugging

To keep the TCB small, our current design offers no debugging support for trustlets. During development, trustlets can be initially debugged and tested within an ARM simulator. However, once isolated in a trustbox, trustlets can only communicate through their interfaces with the outside, making debugging very difficult. While debugging support for trustlets *could* be added to TLR, adding such support is in conflict with the security and isolation goals that TLR trustboxes have. Finding the correct balance between security and ease of debugging is a challenging open issue that TLR shares with many trusted computing systems previously proposed [10, 13, 12].

5. RELATED WORK

Previous work focused on using trusted computing hardware for building systems that provide code and data protection from the underlying OS [10, 13, 12]. Such systems face a tension between security and usability. While some systems depend on a large trusted computing base (TCB) to offer high-level functionality [10], others have stronger security properties by building systems with small TCBs but offer programming abstractions that are low-level [13, 12]. The TLR bridges these extremes by offering a high level programming abstraction while keeping the TCB small.

Another area of research uses privilege separation for partitioning an application into security-sensitive and security-insensitive components. Typically, these systems expose a partitioning interface at the level of the programming language, and enforce this separation by using a runtime [14], or the OS itself [7]. In general, however, they still depend on a large TCB, which includes the OS and the runtime. Our work offers a coarser-grained privilege separation at the language level by compartmentalizing an application while significantly reducing the TCB size.

Finally, there is little published work on building systems that use the ARM TrustZone technology for their trustworthy computing needs. One relevant piece of related work proposes to merge the TPM-based primitives found on x86 machines with those found on ARM in order to build a Linux-based embedded trusted computing platform [16]. That paper uses a VM-based design and offers a special "TrustZone VM" to run trusted code. In contrast, the TLR avoids the energy and performance overheads that come with hypervisor-based virtualization systems.

6. CONCLUSIONS

This paper presents the Trusted Language Runtime (TLR), an environment for running trusted applications on the smartphone. TLR offers a trustbox primitive, which is a runtime environment that offers code and data integrity and confidentiality. With TLR, programmers can write applications in .NET and specify which parts of the application should run inside a trustbox. These parts, called trustlets, are protected from the remaining code running on the smartphone, including its OS and other applications.

TLR uses the ARM TrustZone, which is a hardware technology for trustworthy computing found in ARM chips. The rich hardware support offered by ARM TrustZone combined with the flexibility of the .NET programming environments allows TLR to offer a secure, yet rich programming environment for developing trusted mobile applications. In addition to presenting the design and ARM emulator-based implementation of TLR, this paper discusses some of the open issues that such a platform raises. Our overarching goal is to ignite the discussion of what trusted computing abstractions we need to provide to mobile developers.

Acknowledgements: We would like to thank the anonymous reviewers and Jaeyeon Jung, our shepherd, for their feedback.

7. REFERENCES

[1] .NET Micro Framework. `http://www.microsoft.com/netmf/default.mspx`.

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

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

[4] 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`.

[5] ARM. ARM RealView Development Suite. `http://www.arm.com/products/tools/software-tools/index.php`, last accessed Oct 21, 2010.

[6] J. Bickford, R. O'Hare, A. Baliga, V. Ganapathy, and L. Iftode. Rootkits on Smart Phones: Attacks, Implications and Opportunities. In *Proc. of HotMobile '10*, 2010.

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

[8] 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.

[9] 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.

[10] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proc. of SOSP '03*, 2003.

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

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

[13] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proc. of EuroSys '08*, 2008.

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

[15] Trusted Computing Group. Trusted platform module specification, parts 1–3, 2007. version 1.2 revision 103, Available from `www.trustedcomputing.org`.

[16] J. Winter. Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms. In *Proc. of STC '08*, 2008.