DroidVault: A Trusted Data Vault for Android Devices

Xiaolei Li*, Hong Hu*, Guangdong Bai†, Yaoqi Jia*, Zhenkai Liang*, Prateek Saxena*

*Department of Computer Science, National University of Singapore
†Graduate School for Integrative Sciences and Engineering, National University of Singapore

***Abstract*—Mobile OSes and applications form a large, complex and vulnerability-prone software stack. In such an environment, security techniques to strongly protect sensitive data in mobile devices are important and challenging. To address such challenges, we introduce the concept of the *trusted data vault*, a small trusted engine that securely manages the storage and usage of sensitive data in an untrusted mobile device. In this paper, we design and build DroidVault— the first realization of a trusted data vault on the Android platform. DroidVault establishes a secure channel between data owners and data users while allowing data owners to enforce strong control over the sensitive data with a minimal trusted computing base (TCB). We prototype DroidVault via the novel use of hardware security features of ARM processors, i.e., TrustZone. Our evaluation demonstrates its functionality for processing sensitive data and its practicality for adoption in the real world.**

I. Introduction

The rapid adoption of mobile devices poses an imminent threat to the sensitive data in enterprises and cloud services. Mobile OSes and applications form a large, complex and vulnerability-prone software stack, which is witnessing a sharp rise in malware and OS vulnerabilities [1]. In addition, Android users often install untrusted applications or make modifications (e.g., via "rooting") to the Android OS to bypass restrictions set by vendors, thereby increasing the risk of compromising the software stack. This gives rise to a practical dilemma for data owners: should they trust users' devices and permit the use of sensitive data in devices outside their control, or should they enforce strong control over the sensitive data by banning untrusted devices. Data owners often choose to blindly trust the commodity mobile OSes and user-installed mobile applications.

Trusted Data Vault. Ideally, if mobile platforms can provide mechanisms for data owners to control the usage of the sensitive data, strong data protection can be achieved in existing mobile devices. To enable this, we introduce the concept of *trusted data vault* — a small trusted engine that data owners can trust to securely manage the storage and usage of the sensitive data in untrusted devices. A trusted data vault must balance the misaligned incentives between data users and data owners — data users want unfettered control of the applications and the OS, while data owners need strong control over the sensitive information.

Existing work [2]–[5] has been dedicated into building an isolated secure environment in the mobile devices. Onboard credentials platform [2] designs an architecture for the credential management via a hardware-assisted secure environment. Other work [3]–[5] implements the Mobile Trusted Module, a secure element specified by Trusted Computing Group, through either software-based or hardware-based approaches. However, these solutions either only support limited functionality than secure storage and verification, or rely on a large trusted computing base (TCB) to perform operations on sensitive data.

Approach. In this work, we propose DroidVault, a trusted data vault for Android devices. DroidVault ensures that all the sensitive data remains encrypted throughout its lifetime in the untrusted Android device, and also supports an execution environment for trusted code to operate on the encrypted data. To extend the trust from the data owner's workspace (e.g., the enterprise workspace or cloud storage services) to the mobile client, we expose four important services in DroidVault: *secure network communication*, *secure data storage*, *secure input and output*, *secure data processing environment*.

The main challenge in designing a practical data vault is to enable limited but sufficient functionality with a minimal TCB. Existing secure hardware platforms, such as ARM TrustZone, TPM, M-shield, JavaCard and NGSCB [6], only provide limited available resources for secure environment, such as limited memory and storage, and thus to make a practical deployment, the TCB of the data vault must be kept as small as possible.

We prototype DroidVault as a small trusted hardware-assisted engine through the novel use of hardware security features supported by recent ARM CPUs, namely the ARM TrustZone. It is being widely adopted in ARM-based embedded devices. Unlike software virtualization mechanisms which multiplex two executions on the same CPU, the ARM TrustZone architecture isolates the CPU core and MMU subsystem at the hardware level and creates two environments with different security privilege levels. It supports red/green systems [6]–[8], which partition hardware resources into a highly-constrained trusted (green) environment and a general-purpose untrusted (red) environment. Therefore, it enables DroidVault to co-exist with a completely untrusted Android software stack. We prototype DroidVault in the trusted environment which holds a higher security privilege but only limited resources, and thus DroidVault behaves as a tiny trusted engine for handling sensitive data operations in the same device that hosts a separate untrusted Android OS.

To significantly minimize the TCB of DroidVault, we leverage the network and file system modules from the untrusted

978-1-4799-5482-7/14 $31.00 © 2014 IEEE
DOI 10.1109/ICECCS.2014.13

Android OS. In our implementation, DroidVault has a TCB of about 12K lines of unoptimized code — this is within the range of systems which can be formally checked by existing verification tools [9]. Note that as a prototype for now, we only use a serial console and a hardware keyboard to simulate the secure display and input. We have verified that the USB touchscreen driver in the Android source code has only 1.1K lines of code (LOC). Therefore, the TCB will not increase too much if including the touchscreen driver in the future implementation.

In summary, we claim the following contributions:

- We propose the concept of a trusted data vault and design DroidVault, a usable data vault for the Android platform. To the best of our knowledge, DroidVault is the first end-to-end platform that guarantees sensitive data protection for data owners (note that, we refer data owners to remote data-hosting servers instead of end users) in untrusted Android devices.
- Many commercial vendors build virtualization systems on top of the ARM TrustZone. In contrast, we build a red/green system without relying on virtualization, instead on partitioning. Further, previous commercial systems do not give details of their security design and implementation. Our work is the first in this aspect to our knowledge.
- DroidVault finds a sweet spot between allowing full-fledged functionality and having a small TCB for strong security. We propose a novel combination of using ARM TrustZone primitives and reusing a large fraction of the untrusted Android stack. DroidVault has a small TCB of roughly 12K LOC (only 0.046% of the standalone Android OS).
- We evaluate the applicability of DroidVault to work on the protected data without sacrificing the data privacy and integrity. We also test the performance overhead of file downloads. The results show that the overhead grows linearly with the file size in our unoptimized implementation (6% for 1K $\sim$ 99% for 10M).

II. Overview

Consider that a user Alice who uses an Android device as a client for accessing sensitive files from trusted environments, such as her enterprise server. As an example, Alice needs to retrieve files from her enterprise server via her personal Android device, and process them on the device. In this scenario, the personal Android device, if compromised, gives malicious applications access to Alice's sensitive data. Though recent research [10]–[12] and commercial solutions [13], [14] have enabled protection for sensitive data using encryption, the files still need to be decrypted on the untrusted Android device. Therefore, the sensitive data is exposed in its raw form to a large and complex software stack. The data owner (note that the data owner in our example is Alice's enterprise server, not Alice) has little control over which applications and operations can access the sensitive data.

A. Threat model & Scope

In our threat model, the scope of our approach encompasses a broad spectrum of attacks that steal or corrupt sensitive data by exploiting vulnerabilities in the Android software stack, both at the user level and at the kernel level. Such attacks include misusing permissions [15], escalating privileges [15]–[17], exploiting vulnerable applications [18], [19], including malicious libraries [20], exploiting Android OS vulnerabilities [21] and compromising the Android kernel [22].

DroidVault aims to provide a trusted environment for receiving and processing sensitive files, which extends security guarantees from remote storage servers to the local Android devices. Therefore the sensitive data mentioned in this paper only refers to sensitive files, excluding data in other forms, such as device attributes (e.g., GPS location and device ID).

DroidVault does not aim to protect against denial-of-service attacks or against malicious device users. A compromised Android OS can still deny services to DroidVault or simply delete the local copy of encrypted data. DroidVault relies on a trusted execution environment that cannot be compromised in our threat model. DroidVault guarantees that only trusted code signed by the data owner can operate on the sensitive data in the trusted execution environment. However, it is out of scope if the trusted code itself behaves suspiciously, such as executing an infinite loop or intentionally leaking sensitive data publicly. To gain a strong guarantee, we prototype DroidVault via the ARM TrustZone hardware protection to defeat even threats from a compromised Android OS. However, a malicious user may subvert DroidVault's integrity by using hardware attacks (such as the Direct Memory Access attack or peripherals [23]), cold-boot attacks [24], [25] or by compromising the hardware integrity — these attacks are beyond the scope of DroidVault.

B. Trusted Data Vault

To counter a large threat landscape, we introduce the concept of a *trusted data vault*, a trusted engine that securely enables operations on sensitive data in Android devices. In our motivating example, sensitive data are decrypted before being accessed in the untrusted Android software stack. In contrast, in a trusted data vault, sensitive data are always protected with encryption techniques when the data are outside of the trusted data vault. The operations on the decrypted data can only be successfully executed inside the trusted data vault. The trusted data vault has the following security primitives:

- *Confidentiality/Integrity.* Sensitive data must be encrypted throughout its lifetime (including storage and transmission) in the untrusted OS.
- *Secure Display and Input.* DroidVault guarantees a trusted path to the end display for rendering sensitive data. Similarly, it provides a trusted path from sensitive inputs to the designated code.
- *Operations on Sensitive Data.* DroidVault only allows the authorized code to operate on the decrypted data.

There are a few practical challenges in designing a trusted data vault in mobile devices. First, the size of the TCB in the trusted data vault should be small to be trustworthy. Second, it should be space-efficient due to resource restrictions. To minimize the size of the TCB, our trusted data vault (i.e., DroidVault) supports only limited functionality, which rules out the option of using a virtual machine to host the trusted data vault. We combine the use of new hardware partitioning

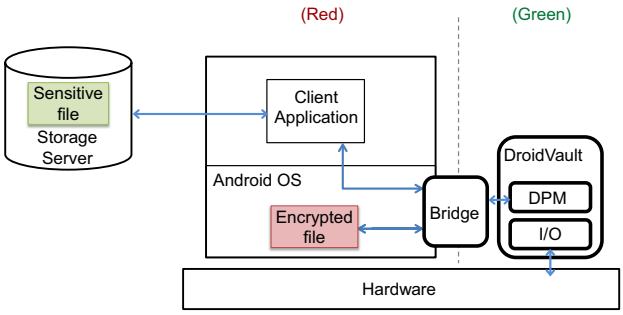

Fig. 1. DroidVault Design

primitives implemented in recent ARM CPUs (i.e., TrustZone), to support a small TCB and achieve these goals.

To protect the sensitive data, any result derived from the sensitive information inside the DroidVault cannot be leaked to the Android OS. Although this limits the functionality that DroidVault currently supports, the size of the TCB is significantly reduced. We show that DroidVault is sufficient to support several common applications which have a clear boundary between their sensitive parts and non-sensitive parts and thus can execute separately in two environments without data exchange, such as file downloading and simple document processing (described in Section V). We do not provide any interface for the Android OS to retrieve any sensitive data from DroidVault. Only authorized code can be loaded into DroidVault from the Android OS and operate on the sensitive data. To minimize the size of the TCB, we restrict the sensitive data and their corresponding computational results inside the trusted data vault in our work, unless the authorized code explicitly exposes its own sensitive data to the Android OS.

III. DroidVault Design

We design DroidVault on the Android platform while taking advantage of the hardware-assisted isolated environment. We focus on analyzing its minimal requirements and security guarantees.

A. DroidVault Components

Figure 1 illustrates the design of DroidVault. To reduce the performance overhead and the size of the TCB, we choose to design DroidVault as a partitioning-based red/green system rather than a virtualization layer. DroidVault is a trusted engine which is isolated from the Android OS. It contains the following main components: the *Data Processing Module (DPM)*, the *Input/Output (I/O) module* and the *bridge module*, which are described below. DPM is the secure data processing environment and also supports secure user interaction through the I/O module. The bridge module provides interfaces for communication between DroidVault and the Android OS.

DPM. DPM is the core module for sensitive data transmission and data operations. It maintains a secure channel with the remote storage server to securely transmit sensitive data. Sensitive data are then encrypted before leaving the DroidVault environment into the Android file system. When Android applications, such as the client application in Figure 1, need to access the encrypted sensitive files, they must load authorized code into DPM for operations on the sensitive data. DPM module verifies whether the loaded code is signed by the data owner. Data owners take the responsibility to develop and sign the code for processing the sensitive data. DPM provides a tightly controlled runtime environment for supporting limited operations. Section V lists a few scenarios for basic data operations. DPM returns no sensitive information in plaintext to the untrusted world. The result can only be displayed inside DroidVault through the secure I/O module.

Bridge Module. To facilitate communication between DroidVault and the untrusted Android OS, DroidVault introduces the bridge module. The bridge exposes interfaces to make certain permitted function calls from one world to the other, and allows passing serializable primitive data between worlds via the shared memory. For example, the bridge module provides a single API `LoadCode` for the Android OS to load the signed code into DPM. The bridge also enables DroidVault to use resources belonging to the Android system, such as the network and the file system. DroidVault ensures that untrusted inputs cannot compromise the TCB and that the security-critical data leaving DroidVault is encrypted.

I/O Module. The I/O module in DroidVault enables secure user input and display. DPM can request the I/O module to display sensitive data directly to users and receive user inputs.

B. Initial setup

DroidVault assumes the availability of two standard hardware primitives — secure persistent storage [26] that cannot be accessed by the untrusted Android system, and secure boot [27]. These primitives are available on existing ARM-based architectures in different ways [26], [28]. To establish trustworthy connections with authenticated servers, DroidVault stores a root certificate that identifies the root certificate authority and therefore verifies other digital certificates using a chain of trust. To prevent the untrusted Android system from masquerading as the DroidVault environment, mutual authentication is required. Data owners need to make sure that the protected files should only be received by the intended DroidVault environment. For this purpose, each DroidVault has a unique public/private key pair (K_{pub}, K_{prv}). The public key K_{pub} should be certified as a public key belonging to a compliant DroidVault system by a trusted authority, such as the device manufacturer or other trusted intermediaries (e.g., the enterprise internal certificate server). The private key K_{prv} is stored in the secure persistent storage which is only accessible inside the secure environment. We have two preparatory steps for deploying this key pair, as described below.

One Time Registration. To establish a secure channel with a remote server, a user needs to notify the data owner with his K_{pub} through one time registration. In enterprise environment, employees can register their public keys with the help of administrator. For data-hosting cloud service providers, users can log in to their accounts and then upload their public keys through particular web interfaces. Users need to contact either the administrator or the service provider if they want to change the uploaded public key in future.

Mutual Authentication. After the public key registration, mutual authentication between a remote server and a compliant

DroidVault system can be achieved. DroidVault stores the root certificate in its secure storage and uses it to verify whether the remote server's certificate is from a trustworthy certificate authority. The remote server also authenticates the incoming connection using the public key registered by the user. The failure of mutual authentication indicates one of the following scenarios: 1) the remote server's certificate is fake; 2) the incoming connection is not from a compliant DroidVault environment; 3) the registered public key is incorrect. As to the scenarios 1) and 2), attackers cannot successfully download any sensitive data. As to the scenario 3) that attackers may change the registered public key with their own (e.g., contact the administrator by impersonating a victim), the victim can verify the registered public key by checking its hash through secure display in the secure world and make an update in time.

C. DroidVault Services

In this section, we discuss the new security services supported by DroidVault components and the security guarantees that DroidVault provides. We illustrate how the security goals stated in Section II-B are achieved.

1) Secure Network Communication: Due to the goal of a small TCB, DroidVault does not include the network driver in the secure world. Thus it needs to securely upload/download files to/from remote servers through the untrusted Android environment.

DroidVault supports secure communication by implementing the Secure Socket Layer (SSL). SSL operations have two phases: data encryption and data transmission. DroidVault handles the data encryption in the secure world, which prepares the data to be transmitted based on encryption. For this purpose, DroidVault provides different types of cryptographic APIs in the secure world, such as the symmetric cryptographic algorithm (e.g., AES-GCM) and the asymmetric cryptographic algorithm (e.g., RSA). DroidVault holds the root certificate in its secure storage to build a chain of trust for other digital certificates and therefore authenticates remote servers.

In the data transmission phase, DroidVault requests network-related system calls (e.g., `socket`, `connect` and `gethostbyname`) from the untrusted Android OS through the bridge module. The received data from the untrusted Android OS are sanitized by DroidVault to protect against exploits from inputs. Note that the sensitive data in the network are encrypted before they leave the secure world. Therefore, the untrusted software stack in the Android OS does not threaten the confidentiality and integrity of the sensitive data.

2) Secure Data Storage: The secure environment only provides limited secure storage, which is not practical to store all the sensitive data. Therefore, we need to extend the secure data storage with the help of the Android file system — an untrusted but relatively large storage space. To store sensitive data in the untrusted file system, DroidVault encrypts the data and invokes file-system-related system calls through the bridge module, which include `open`, `read`, `write` and `close`. The sensitive data are in encrypted form in the untrusted Android file system. Thus DroidVault also avoids including the file system driver into its TCB.

3) Secure Display and Input: To provide an end-to-end channel that directly interacts with device users, DroidVault ensures that the sensitive display and input can never be accessed by untrusted Android drivers. This requires a secure overlay which securely renders any sensitive information on the screen under the complete control of DroidVault. Unlike the design for secure network communication and data storage primitives, where DroidVault can delegate most of the task to the untrusted Android OS, secure display and input must be independently supported by DroidVault through direct control over the display and input devices. We add drivers inside DroidVault to control the display and input. DPM provides the API `Display` to create a secure display session, and the API `Keyboard` to receive inputs.

4) Secure Data Processing: DroidVault encrypts sensitive files to achieve confidentiality. To support operation on protected data, DPM is the key component, which allows the authorized code signed by data owners to operate on the sensitive data. Sensitive data are securely transmitted from a remote storage server into DPM, and then encrypted before stored in the untrusted Android OS. We use *metadata* to record the information which is used to decrypt and authenticate the encrypted sensitive data. The metadata is also encrypted and associated with the corresponding sensitive data in the Android OS.

DPM only allows the authorized code to be executed. Existing work [2], [29] has built adequate frameworks which support popular programming languages (such as Lua and C#) in an isolated secure environment for the ease of third-party development. However, considering the goal of minimizing DroidVault's TCB, it is not necessary to include a whole functional code environment into DroidVault's TCB. The size of the TCB eventually depends on the functionality to be supported. To minimize the code environment, we only provide several common functions including a set of APIs for data operations, network communication, file system access and secure display/input, listed in Table I. Note that the encryption and decryption processes for secure network communication and secure data storage are transparent to the DPM code. For example, `FileRead` directly returns the plaintext of an encrypted file without requiring any further decryption in the DPM code. Any runtime environment which supports these corresponding functions can be fit into DPM (we do not argue which programming language is the most suitable one). The DPM APIs are designed for the code running inside DPM. The authorized code is loaded from the Android OS into DPM through the bridge module API `LoadCode`. Next, we will show the details about how sensitive data are securely transmitted from a remote server to an Android OS through DPM, and how DPM processes the sensitive data.

Secure Channel. Data transmission follows successful mutual authentication (described in Section III-B) between the remote server and the DroidVault execution environment. After establishing a secure channel, both sides share a secret key for further data transmission. Shown in Figure 2, the sensitive data are then securely transmitted from the remote server to DPM (step 1). The Android OS is not able to decrypt the sensitive data without the shared secret key even though the connection goes through its network stack (shown as dash line in Figure 2).

After receiving the sensitive data, DPM encrypts it and

TABLE I. DPM APIs

Operations	DPM APIs
Data Operations	`Integer Compare(Stream s1, Stream s2)`
	`Stream Concat(Stream s1, Stream s2)`
	`Integer IndexOf(Stream s1, Stream s2, Integer fromIndex)`
	`Stream SubStream(Stream s, Integer beginIndex, Integer endIndex)`
	`Stream Replace(Stream s, Stream regex, String replacement, Integer limit)`
	`Integer Length(Stream s)`
Network Communication	`Descriptor HttpsConnect(Stream url)`
	`Integer HttpsSend(Descriptor net, Stream s)`
	`Stream HttpsReceive(Descriptor net)`
	`Integer HttpsClose(Descriptor net)`
File System	`Descriptor FileOpen(Stream fileName, Integer mode)`
	`Stream FileRead(Descriptor file, Integer length)`
	`Integer FileSeek(Descriptor file, Integer offset, Integer whence)`
	`Integer FileWrite(Descriptor file, Stream data)`
	`Integer FileClose(Descriptor file)`
Screen Display	`Integer Display(Stream data)`
User Input	`Stream Keyboard()`

then stores it in the untrusted Android OS. *Key Generator* randomly[1] generates a key K_{AE} and sends the key as an input to *Authenticated Encryption Module*. This module encrypts the sensitive data with K_{AE} and then outputs the ciphertext and the authentication tag (step 2). The ciphertext (i.e., the encrypted sensitive data) is directly stored into the untrusted Android OS. To maintain the information which is used to decrypt the ciphertext in future, we define a metadata structure which contains the authentication tag, K_{AE} and the data authority (`XYZ` in Figure 2). *Metadata Generator* takes these three pieces of information as inputs to compose the metadata and then encrypts it with K_{pub}, the public key belonging to DroidVault (step 3). Therefore, the metadata can only be viewed as plaintext in DPM. The encrypted metadata is then associated with the encrypted sensitive data in the Android OS.

Data Processing. DroidVault only allows execution of the code signed by the data owner. The code can be loaded into DPM from the Android OS, or remotely downloaded from the server as Figure 2. The code is sent to *Code Authority Verifier* to check its integrity and authority (i.e., `XYZ`). The verifier makes sure that the code comes from the authority `XYZ` (step 4). Before loading the encrypted sensitive data to be operated on, DPM firstly loads the encrypted metadata into *Metadata Decryption Module*. After decrypting the metadata with K_{prv}, the private key belonging to DroidVault, DPM retrieves the data's authority (step 5). DPM checks whether the data's authority matches the code's authority (step 6). Only when there is a match, it continues to load the encrypted data. *Authenticated Decryption Module* decrypts the encrypted data with the authentication tag and K_{AE} extracted from the metadata (step 7). The code can then operate on the plaintext of the sensitive file. The sensitive data are only decrypted inside DPM which is inaccessible by the untrusted Android OS.

5) Security Analysis: Considering our motivating example, the client on Alice's Android device loads a piece of code signed with the enterprise server's authority into DPM for downloading her document. DroidVault ensures that the document is securely downloaded from the server, marked with the enterprise authority and then locally encrypted before stored into the Android file system.

[1]The random number generator can be implemented in either software or hardware manner.

Now we give a security analysis from the perspective of data integrity and authenticity guarantees. We use metadata to maintain the keys which are used to decrypt sensitive files. The metadata is distributed into the untrusted Android OS along with the encrypted sensitive data and is only loaded when necessary. This design significantly reduces the burden of a central key management, considering the limited secure storage in the secure environment. DroidVault only needs to store an initial public/private key pair which is used to encrypt/decrypt the metadata. Each encrypted file has a corresponding piece of metadata. The mapping between the encrypted file and its metadata can be maintained in a simple way (e.g., the metadata uses the same file name with the encrypted file but with an additional suffix). If attackers corrupt this mapping, they get no benefit but the failure of file decryption.

We choose the authenticated encryption to ensure data integrity and authenticity. Considering that it is not practical to fit large volume files into DroidVault's memory, the encrypted sensitive data are read block by block for processing. The metadata is first read into memory for the authority certificate matching phase (step 6 in Figure 2). The encrypted file is loaded only after the matching succeeds. Attackers may replace the encrypted file by loading cipher blocks with other authorities (Time-of-check to Time-of-use attacks). DroidVault must be able to authenticate each cipher block and also identify the correct order of these blocks. Therefore, the encryption algorithm for protecting sensitive data requires a counter mode block cipher which combines both confidentiality and authenticity, such as CCM and GCM. In our work, we choose GCM for the authenticated encryption. It is possible for attackers to replace both the metadata and the encrypted sensitive data with different ones that hold the same authority as the code. In this scenario, we argue that both the data and the code belong to the same authority so that no sensitive information is leaked in plaintext unexpectedly. The code itself can identify the corresponding data to be operated on if necessary.

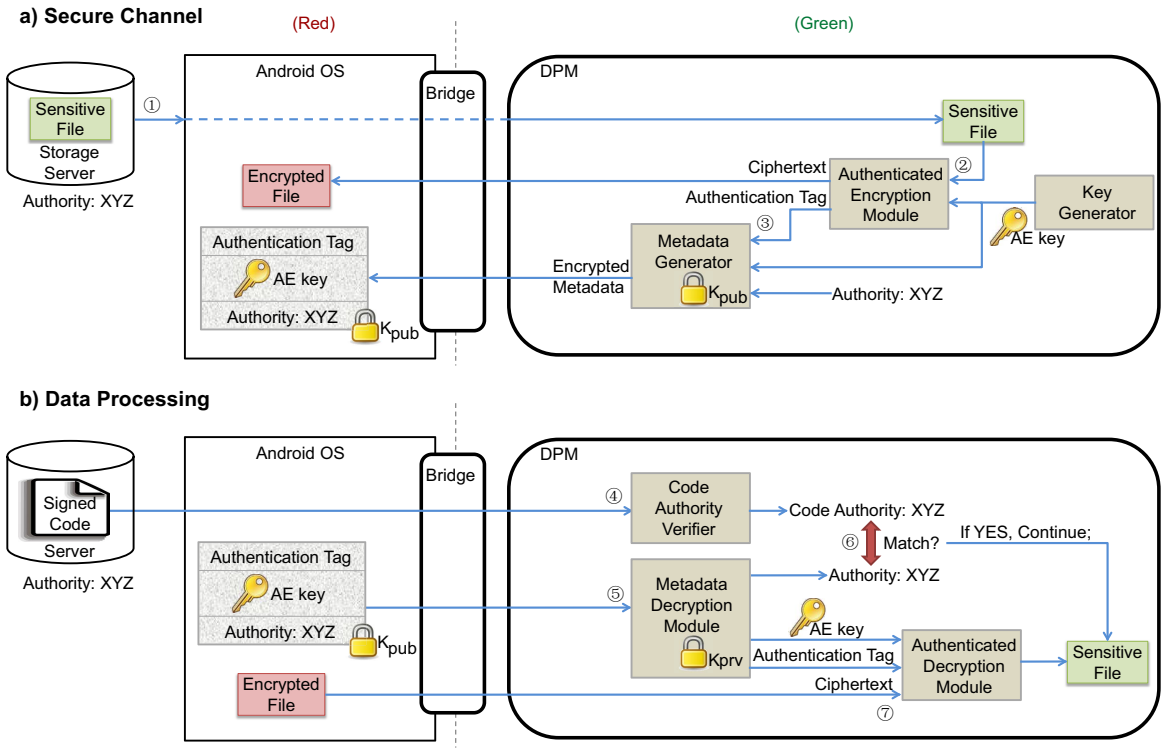

Fig. 2. Secure Channel Establishment and Data Processing in DPM (The dash line means that the sensitive file is not directly exposed to the Android OS even though the channel goes through the Android OS)

IV. Implementation

We implement the DroidVault prototype in the Android Gingerbread 2.3 version on the Freescale i.MX53 Quick Start Board (QSB). We adopt memory manager and interrupt handler partially from Open Virtualization [30], which is an ARM TrustZone open source project currently only supporting the source code for ARM Versatile Express Board and Realview Evaluation Board. Additionally, we implement the basic execution environment for DroidVault including DPM, basic I/O, string library, encryption library, etc., and also port PolarSSL[2], a light-weight SSL/TLS library. The overall LOC is only 12,171, including the unoptimized PolarSSL library which has 8857 LOC. Comparing to the Android source code, our DroidVault's TCB is much smaller than the whole Android OS (26,710,217 LOC, 0.046%) or Dalvik (232,232 LOC, 5.24%). In this section, we describe the implementation challenges when prototyping DroidVault.

Background on ARM TrustZone. ARM TrustZone is a new security extension in the ARM architecture, which has been supported since ARMv6. This feature is increasingly being utilized in emerging enterprise mobile security solutions (e.g., Samsung KNOX). The TrustZone technique is designed to support red/green systems which partition hardware resources into a secure (green) world and a normal (red) world. The two worlds are separated by hardware mechanisms, and both worlds support different privilege levels (unprivileged user level and privileged kernel level). Any interrupt can be configured to be delivered to either of the worlds, but not both. This mechanism can be used to trap all the interrupts into the secure world before they are delivered to the normal world (similar to interrupt handling in the virtualization based system [4]) or to partition the interrupt handlers (as in partitioning based systems [6], [8]). Context switches from the normal world to the secure world, which we refer to as inter-world calls, are activated through a special software interrupt generated by the `SMC` instruction. The secure world can initiate a context switch to the normal world by writing a special value to the `SCR` register. Context switches are handled by software code handlers (rather than hardware as in x86 CPUs). DroidVault handles these context switches with software handlers. The secure world can read and write arbitrary memory of the normal world, while the normal world can only operate on its assigned memory regions. This allows the secure world to build one-way memory isolation, which can be used as a mechanism to share data in the inter-world call. There are other mechanisms supported for secure boot — we do not discuss them here as these are not the focus of DroidVault's core design. Next, we describe the key techniques during prototyping DroidVault on the ARM TrustZone architecture.

World Switch. In the ARM TrustZone architecture, the `SMC` instruction is dedicated to generate a software interrupt that activates a world switch between the secure world and the normal world. `SMC` is only executable inside the kernel space with the privileged mode. It triggers the CPU to enter a special CPU mode, *Monitor Mode*, newly introduced by the ARM

[2]PolarSSL: available at https://polarssl.org/

TrustZone architecture for interfacing two worlds. In monitor mode, we implement the `SMC` handler (256 LOC), which stores all the registers of the current world and then restores the state of the other world.

The ARM microprocessor has 16 general-purpose registers (R0-R15). R0-R7 are used as either temporary registers or argument registers while the rest are preserved for other special purposes. In the ARM TrustZone architecture, all the registers are accessible in the secure world. Some privileged registers are forbidden or blanked in the normal world.

To activate an inter-world call in the kernel space, the bridge module is implemented as a loadable kernel module which adds a handler to the *ioctl* system call in the Android system. When an Android application requests services of the secure world, we use registers R0-R3 to share arguments between the two worlds. R0 and R1 are used to identify the requested service and store the return value from DroidVault to the Android OS, while R2 and R3 are used to store the information about the shared memory between the two worlds, including physical addresses of the input and the output buffer registered by the bridge and the length of each buffer.

When the secure world requests resources belonging to the Android system, we pass all the arguments to the buffer shared with the Android OS, and use registers R0 and R1 to identify the call back request and the system call type. After switching to the normal world, the bridge module takes over and handles the request from DroidVault by parsing the arguments and invoking the corresponding system call. After finishing the system call in the Android system, the bridge module writes the result back to the shared buffer and then uses the `SMC` instruction to switch back to DroidVault. After the world switch, DroidVault restores its previous state and continues the execution.

Porting DroidVault in secure world. We implement the basic execution environment for the secure world on Freescale i.MX53 QSB, including the initialization code, the UART[3] driver and the interrupt handler. We support file-system-related and network-related system calls in the secure world. To reuse the Android file system and network stack for minimizing TCB, these system calls in the secure world are only wrapper interfaces which are further handled in the Android OS using our inter-world calling mechanism. We also port PolarSSL in the secure world. Therefore, the secure world can establish SSL channels with remote servers via the Android network stack. For the secure display and user input, we use a serial console to simulate the secure display and a hardware keyboard as the secure input device. Open Virtualization has supported the display and user input in the secure world on Samsung(R) Exynos 4412, so these two features are not fatal obstacles when porting DroidVault into the ARM TrustZone architecture. Supporting secure display on Freescale i.MX53 QSB is part of our future work.

V. Evaluation

In this section, we discuss the functionality and applicability of our DroidVault prototype. We integrate it with real-world applications and services. We also evaluate the performance of DroidVault.

A. New Applications Enabled by DroidVault

We successfully adapt Dropbox application as a cloud service provider to work with DroidVault. To evaluate the capability of the DPM module, we also develop a few applets for parsing simple documents.

Dropbox File Manager. We build a Dropbox file manager based on the Dropbox SDK to enable secure management of files on Dropbox using an untrusted Android device. This application allows users to securely log in to their Dropbox accounts, browse the Dropbox file system (assuming the file/directory names are not sensitive), upload/download files and search strings in the files. After receiving the user name and password from the secure input, it constructs an HTTP post message to send the password to the server and receive the response via DPM APIs `HttpsSend` and `HttpsReceive`. When a file reaches DroidVault, the file manager encrypts it in the secure world and stores it in the file system. The file manager also allows users to search a string of text in the encrypted file. The secure world generates the `grep`-style output by invoking `IndexOf` API with the particular strings, and displays the result to users on the secure display by invoking `Display` API.

Using DroidVault services, this file manager enables secure file management and string search operation in an untrusted Android device without leaking sensitive data to the device.

Zip Archiver. Zip is a common archive file format. In the zip format, each file record is a *file entry*, which includes the file header and contents; at the end of the zip file, *central directory* contains all the offsets of these entries.

A zip parser typically follows the following steps.

1) Read a file header and check whether it is valid or not by signature matching. If so, obtain attributes of the file in the header.
2) Use the file size, the file name size and the extra data size to get file contents. Move to the next file header.

We investigate the source code of an open source Zip Viewer[4] (written in Java) to evaluate the feasibility of processing zip files through DroidVault. We encrypt a set of zip file samples[5] and modify Zip Viewer to decrypt them inside DroidVault. We develop the code running inside DPM to decrypt the files and extract the file entry names in the zip files through data operations and file-system-related operations listed in Table I. It uses `FileRead` to read the plaintext of the encrypted file header and then parses the header via `Compare`, `IndexOf` and `SubStream`. We consider the file entry names as non-sensitive and thus explicitly return them back to the Android OS. We only need to rewrite 2613 LOC mainly inside *ZipInputStream.java* to request data processing in DroidVault and handle the return results. When the Zip Viewer starts to parse one encrypted zip file for extracting the file entry names,

[3]Universal Asynchronous Receiver/Transmitter translates data between parallel and serial forms.

[4]Zip Viewer: available at http://code.google.com/p/zipviewer/

[5]These files are collected from real-world project files in Google Code: 1) guestbook_10312008 2) schema-upgrades003_019 3) google-secure-data-connector-1.2-0-bin 4) connector-otex-2.6.12-src

TABLE II. THE PERFORMANCE OF ZIP VIEWER WHEN RUNNING WITH DROIDVAULT (MEASURED IN MILLISECOND)

Projects		1	2	3	4
# of Files		12	29	72	162
Size of TAR		20K	60K	4.8M	1.0M
Size of ZIP		4.5K	16K	4.3M	264K
TAR	Without DroidVault	79	95	118	373
	With DroidVault	118	190.3	319.3	1037
	Overhead	49.37%	100.32%	170.59%	178.02%
ZIP	Without DroidVault	60	65	87	235
	With DroidVault	99.4	160	306.8	718
	Overhead	65.67%	146.15%	253.64%	205.53%

TABLE III. THE PERFORMANCE OF FILE DOWNLOADING INSIDE DROIDVAULT (MEASURED IN MICROSECOND)

Size of File	1K	10K	100K	1M	10M
Without DroidVault	4084	5342	17815	148971	1335257
With DroidVault	4366	7008	33518	280805	2663760
Overhead	6.90%	31.19%	88.14%	88.50%	99.49%

we intercept and load our code into DPM to decrypt and parse the zip file. DroidVault returns a list of file entry names back to the Android OS and then Zip Viewer continues to use these results for display.

We also extend Zip Viewer to handle the tar format. Similarly in tar format, each file is organized as one or multiple *content blocks*, preceded by a *header block* which describes its metadata, such as the file name and the size. Each of the block has 512 bytes. Two sequential blocks filled with 0 indicate the end of a file. We only need to slightly adjust 77 LOC.

B. Performance

We build an application that downloads files of various sizes from a remote server with DroidVault. For each file, we download 1000 times and calculate the average download time. Table III shows the download time for files of different sizes.

Comparing with the normal case of file downloading, our solution has three extra steps: 1) after retrieving encrypted data from the SSL channel, the Android OS needs to copy the data into the shared memory with the secure world (Shared Memory Copy); 2) the Android OS triggers a context switch; (Context Switch); 3) after the secure world decrypts the data in the shared memory, it encrypts it locally (Data Encryption). Note that we do not consider the decryption part as an extra step since the normal case also needs to decrypt the data retrieved from the SSL channel. To compare the weight of these three factors, we create our own micro-benchmark to measure the overhead of each step. We measure the time for shared memory copy inside the normal world and the time for data encryption inside the secure world. To evaluate the time for context switch between the two worlds, we modify DroidVault to return to the normal world without any operation inside the secure world. By running 1000 times, we get the average time for context switch around 8.4 microseconds including `SMC` interrupt and context save/restore. Table IV shows our results. The main overhead comes from the context switch and data encryption. The time for context switch depends on hardware platform, which is hard to reduce. However, we can optimize it by increasing the shared memory buffer and thus reducing the number of context switches. The overhead on data encryption depends on the encryption method and the implementation. In the prototype, we have not optimized the code. The performance can be improved by several optimizations, such as adjusting the block size. It can also be significantly improved by hardware implementations [31]. As an optimization to this specific case of file downloading, we can even avoid the extra data encryption step by directly utilizing the encrypted data retrieved from the SSL channel and using the SSL session key to generate the corresponding metadata. We plan to optimize our implementation in the near future.

TABLE IV. THE PERFORMANCE OF OUR MICRO-BENCHMARK TEST (MEASURED IN MICROSECOND)

Size of File	1K	10K	100K	1M	10M
Shared Memory Copy	5	18	202	1523	11818
Context Switch	76	680	6707	67160	683347
Data Encryption	201	968	8794	63151	633338

We also evaluate Zip Viewer to report the performance overhead introduced by DroidVault, which is incurred by encryption/decryption, context switch and data copy between the two worlds. During our experiment, we execute Zip Viewer to read archive files of various sizes in our sample set. Our result is shown in Table II. As the number of compressed files in one archive varies from 12 to 162, the performance overhead increases from tens of milliseconds to hundreds of milliseconds. This is due to that the number of context switches is proportional to the number of file headers (any file-system-related system call in the secure world incurs the context switch from DroidVault to the Android OS). Most of the overhead (50%$\sim$2x) is caused by the context switches during the interaction between the application and DroidVault. Because the overall time is small (less than 1sec), we do not perceive significant delay while interacting with the application.

VI. RELATED WORK

Extending Android to protect sensitive data. Several solutions extend the Android platform to protect the sensitive data. TaintDroid [32] monitors the flow of sensitive information in Android devices to detect the data leakage. AppFence [33], MockDroid [34], Apex [35], Saint [36], Constroid [37], TreeDroid [38], Kynoid [39], TISSA [40], Aurasium [41] and [42] enable the runtime enforcement to support semantic-rich control on sensitive data; for example, an application can specify that any other application granted the network access permission cannot read its sensitive data. Another line of research protects the sensitive data in Android by isolating the code segment according to their sources or security levels. AdDroid [43] and AdSplit [44] separate the code from different origins. They extract libraries out of the host application and use a separate process as a container to isolate them. TrustDroid [45] groups applications into different domains, and the communication among domains is restricted to prevent the data leakage. Some existing work also protects the sensitive data by encryption. For example, CleanOS [46] is a prototype to mitigate the threat of device lost by encrypting sensitive data and evicting the encryption key to the trusted cloud in time. All above solutions in this category rely on the trust of the Android OS. In contrast, DroidVault enables the sensitive

data protection in an untrusted Android system.

Virtualization on Android devices. L4Android [47] is a security framework which supports running multiple Android OSes in parallel through virtualization on top of a microkernel. Each top Android OS runs in a standalone virtual machine. Cells [48] proposes virtual phone environment by configuring virtual device drivers. It is a more lightweight isolated environment comparing with L4Android, and can be treated as a container of available virtual device drivers and Android applications. These virtualization-based solutions achieve the sensitive data protection, but the TCB is quite large, including the whole Android software stack. Even though the resources are isolated, they are still exposed to the malicious applications or the compromised OS.

Data-oriented protection. A few abstractions are designed for data-oriented protection. Lie et al. [49] prevent memory tampering through an abstract of execution-only memory. DataSafe [50] uses memory encryption to protect data, achieving the concept of data capsules [51]. These solutions allow operations on sensitive data under the control of a policy. However, they cannot prevent information leakage through implicit flows and side channels. In contrast, DroidVault provides a stronger guarantee to secure sensitive data with a hardware-assisted isolated environment. Policy-sealed data [52] provides a new trusted computing abstraction to protect customer data hosted by cloud services, based on that the sealed customer data can only be unsealed by nodes that match the customer-defined policy. Similar with our solution, these approaches also use encryption to protect sensitive data and only allow decryption on demand, thus reducing the potential data leakage. However, DroidVault supports richer functionality to operate on the encrypted data.

Trusted execution environment. On-board credentials platform [2] designs an architecture for the credential management via a hardware-assisted secure environment and provisions credential secrets that are only accessible to specific pre-authorized programs inside the secure environment. However, our DroidVault design aims to establish a secure channel with remote data-hosting servers and support secure interaction with end users, which they do not address. NGSCB [6] developed by Microsoft provides an execution environment with high isolation assurance on both software and hardware base. DroidVault can be adapted into the NGSCB architecture. Existing research has provided trusted execution environment based on the virtualization (Terra [53], Proxos [54], etc.) and trusted hardware (Flicker [55], vTPM [56], etc.). The idea is to establish trust in the system based on a small root of trust. Mobile Trusted Module is a platform-independent approach for trusted computing, similar to TPM [57]. It allows a wide range of implementations, such as based on SELinux [3] or hardware support (ARM TrustZone and Secure Element [4], [5]). However, all above solutions mainly focus on the integrity of applications. They do not preserve the application usability by allowing operations on the sensitive data. Our solution is additionally designed to support useful data operations on protected sensitive data. [58] proposes a solution of trusted path on x86 computers which establishes a protected channel between a user's I/O device and a program. Their solution is a hypervisor-based design which is claimed to be portable onto the ARM platform in the future. However, instead of building trust between a user's I/O device and a program, DroidVault aims to extend the trust with remote servers.

VII. Conclusion

We present DroidVault, a trusted engine on the Android platform, to ensure the confidentiality of the sensitive data. It establishes the trust between data-hosting servers and Android devices, and provides a trusted execution environment for processing the sensitive data. We prototype DroidVault on the ARM TrustZone architecture to rigorously isolate the sensitive data from the untrusted Android OS. DroidVault has a significantly reduced TCB compared to the present Android OS. Through our evaluation, we demonstrate that DroidVault can be adopted by legacy cloud storage services and support popular operations on sensitive data.

Acknowledgments

We thank anonymous reviewers for their valuable feedback. This research is partially supported by the research grant R-252-000-519-112 from Ministry of Education, Singapore.

References

[1] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *IEEE SP*, 2012.

[2] K. Kostiainen, J.-E. Ekberg, N. Asokan, and A. Rantala, "On-board Credentials with Open Provisioning," in *ASIACCS*, 2009.

[3] X. Zhang, O. Acıiçmez, and J.-P. Seifert, "A Trusted Mobile Phone Reference Architecture via Secure Kernel," in *STC*, 2007.

[4] J. Winter, "Trusted Computing Building Blocks for Embedded Linux-based ARM Trustzone Platforms," in *STC*, 2008.

[5] K. Dietrich and J. Winter, "Towards Customizable, Application Specific Mobile Trusted Modules," in *STC*, 2010.

[6] M. Peinado, Y. Chen, P. Engl, and J. Manferdelli, "NGSCB: A Trusted Open System," in *ACISP*, 2004.

[7] B. Lampson, "Privacy and Security Usable Security: How to Get It," *Commun. ACM*, vol. 52, no. 11, Nov. 2009.

[8] A. Vasudevan, B. Parno, N. Qu, V. D. Gligor, and A. Perrig, "Lockdown: Towards a Safe and Practical Architecture for Security Applications on Commodity Platforms," in *TRUST*, 2012.

[9] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "SeL4: Formal Verification of an OS Kernel," in *SOSP*, 2009.

[10] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting Confidentiality with Encrypted Query Processing," in *SOSP*, 2011.

[11] C. Weinhold and H. Härtig, "VPFS: Building a Virtual Private File System With a Small Trusted Computing Base," in *EUROSYS*, 2008.

[12] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang, "Enabling Security in Cloud Storage SLAs with CloudProof," in *ATC*, 2011.

[13] "BoxCryptor," https://www.boxcryptor.com/.

[14] "Viivo: Cloud File Encryption," http://viivo.com/.

[15] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission Re-delegation: Attacks and Defenses," in *USENIX SECURITY*, 2011.

[16] W. Enck, M. Ongtang, and P. Mcdaniel, "Mitigating Android Software Misuse Before It Happens," Tech. Rep., 2008.

[17] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege Escalation Attacks on Android," in *ISC*, 2011.

[18] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software," in *CCS*, 2012.

[19] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities," in *CCS*, 2012.

[20] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe Exposure Analysis of Mobile In-App Advertisements," in *WISEC*, 2012.

[21] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic Detection of Capability Leaks in Stock Android Smartphones," in *NDSS*, 2012.

[22] J. Bickford, R. O'Hare, A. Baliga, V. Ganapathy, and L. Iftode, "Rootkits on Smart Phones: Attacks, Implications and Opportunities," in *HOTMOBILE*, 2010.

[23] B. Carrier and J. Grand, "A Hardware-based Memory Acquisition Procedure for Digital Investigations," *Digit. Investig.*, vol. 1, no. 1, 2004.

[24] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Cal, A. J. Feldman, and E. W. Felten, "Least We Remember: Cold Boot Attacks on Encryption Keys," in *USENIX SECURITY*, 2011.

[25] "Danger on ice: Android info thaws in cold boot attack," http://phys.org/news/2013-02-danger-ice-android-info-cold.html.

[26] A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, and J. M. McCune, "Trustworthy Execution on Mobile Devices: What Security Properties Can My Mobile Platform Give Me?" in *TRUST*, 2012.

[27] "ARM Security Technology: Building a Secure System using TrustZone Technology," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/DABGFFIC.html.

[28] "CryptoCell®for TrustZone," http://www.discretix.com/cryptocell-for-trustzone/.

[29] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Trusted Language Runtime (TLR): Enabling Trusted Applications on Smartphones," in *HOTMOBILE*, 2011.

[30] "Open Virtualization," http://www.openvirtualization.org/.

[31] T. Babu, K.V.V.S.Murthy, and G.Sunil, "Aes algorithm implementation using arm processor," *ICWET*, 2011.

[32] 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 *OSDI*, 2010.

[33] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the Droids You're Looking for: Retrofitting Android to Protect Data from Imperious Applications," in *CCS*, 2011.

[34] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "MockDroid: Trading Privacy for Application Functionality on Smartphones," in *HOTMOBILE*, 2011.

[35] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints," in *ASIACCS*, 2010.

[36] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically Rich Application-Centric Security in Android," in *ACSAC*, 2009.

[37] D. Schreckling, J. Posegga, and D. Hausknecht, "Constroid: Data-centric Access Control for Android," in *SAC*, 2012.

[38] M. Dam, G. L. Guernic, and A. Lundblad, "TreeDroid: A Tree Automaton Based Approach to Enforcing Data Processing Policies," in *CCS*, 2012.

[39] D. Schreckling, J. Posegga, J. Köstler, and M. Schaff, "Kynoid: Real-time Enforcement of Fine-grained, User-defined, and Data-centric Security Policies for Android," in *WISTP*, 2012.

[40] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming Information-stealing Smartphone Applications (on Android)," in *TRUST*, 2011.

[41] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical Policy Enforcement for Android Applications," in *USENIX SECURITY*, 2012.

[42] D. Kantola, E. Chin, W. He, and D. Wagner, "Reducing Attack Surfaces for Intra-Application Communication in Android," in *SPSM*, 2012.

[43] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "AdDroid: Privilege Separation for Applications and Advertisers in Android," in *ASIACCS*, 2012.

[44] S. Shekhar, M. Dietz, and D. S. Wallach, "AdSplit: Separating Smartphone Advertising from Applications," *CoRR, abs/1202.4030*, 2012.

[45] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, "Practical and Lightweight Domain Isolation on Android," in *SPSM*, 2011.

[46] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "CleanOS: Limiting Mobile Data Exposure with Idle Eviction," in *OSDI*, 2012.

[47] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter, "L4Android: a Generic Operating System Framework for Secure Smartphones," in *SPSM*, 2011.

[48] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh, "Cells: a Virtual Mobile Smartphone Architecture," in *SOSP*, 2011.

[49] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," *SIGPLAN Not.*, 2000.

[50] Y.-Y. Chen, P. A. Jamkhedkar, and R. B. Lee, "A Software-Hardware Architecture for Self-Protecting Data," in *CCS*, 2012.

[51] P. Maniatis, D. Akhawe, K. Fall, E. Shi, S. McCamant, and D. Song, "Do You Know Where Your Data Are?: Secure Data Capsules for Deployable Data Protection," in *HOTOS*, 2011.

[52] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu, "Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services," in *USENIX SECURITY*, 2012.

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

[54] R. Ta-Min, L. Litty, and D. Lie, "Splitting interfaces: Making Trust between Applications and Operating Systems Configurable," in *OSDI*, 2006.

[55] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An Execution Infrastructure for TCB Minimization," in *EUROSYS*, 2008.

[56] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: Virtualizing the Trusted Platform Module," in *USENIX SECURITY*, 2006.

[57] "Trusted Platform Module (TPM) Specifications," https://www.trustedcomputinggroup.org/home.

[58] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building Verifiable Trusted Path on Commodity x86 Computers," in *IEEE SP*, 2012.