Protecting Mobile Devices from Physical Memory Attacks with Targeted Encryption

Le Guan
University of Georgia, USA
leguan@cs.uga.edu

Chen Cao
Sencun Zhu
Pennsylvania State University, USA
{cuc96,sxz16}@psu.edu

Jingqiang Lin
Data Assurance and Communication Security Research Center, China
IIE, CAS, China
linjingqiang@iie.ac.cn

Peng Liu
Pennsylvania State University, USA
pxl20@psu.edu

Yubin Xia
Shanghai Jiao Tong University, China
xiayubin@sjtu.edu.cn

Bo Luo
University of Kansas, USA
bluo@ku.edu

ABSTRACT

Sensitive data in a process could be scattered over the memory of a computer system for a prolonged period of time. Unfortunately, DRAM chips were proven insecure in previous studies. The problem becomes worse in the mobile environment, in which users' smartphones are easily lost or stolen. The powered-on phones may contain sensitive data in the vulnerable DRAM chips. In this paper, we propose `MemVault`, a mechanism to protect sensitive data in Android devices against physical memory attacks. `MemVault` keeps track of the propagation of well-marked sensitive data sources, and selectively encrypts tainted sensitive memory contents in the DRAM chip. When a tainted object is accessed, `MemVault` redirects the access to the internal RAM (iRAM), where the cipher-text object is decrypted transparently. iRAM is a system-on-chip (SoC) component which is by nature immune to physical memory exploits. We have implemented a `MemVault` prototype system, and have evaluated it with extensive experiments. Our results validate that `MemVault` effectively eliminates the occurrences of clear-text sensitive objects in DRAM chips, and imposes acceptable overheads.

CCS CONCEPTS

• **Security and privacy → Systems security**; *Mobile and wireless security*.

KEYWORDS

memory encryption, taint analysis, physical attack

ACM Reference Format:
Le Guan, Chen Cao, Sencun Zhu, Jingqiang Lin, Peng Liu, Yubin Xia, and Bo Luo. 2019. Protecting Mobile Devices from Physical Memory Attacks with Targeted Encryption. In *12th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '19), May 15–17, 2019, Miami, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3317549.3319721

WiSec '19, May 15–17, 2019, Miami, FL, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6726-4/19/05…$15.00
https://doi.org/10.1145/3317549.3319721

1 INTRODUCTION

In computer systems, application data are first loaded into memory and then processed by the processor. Memory is implemented as DRAM chips in commodity systems, and these data could be scattered over the DRAM chips for a prolonged period of time, even days after a process has been terminated [6]. Unfortunately, DRAM chips are vulnerable to physical memory attacks [17], which render all data in the DRAM chips, either active or residual, unsafe.

This problem becomes worse on mobile devices, which are seldom rebooted and are easily stolen or lost. Sensitive data on mobile devices, such as email contents, bank accounts and passwords, persists as clear-text in memory or on storage even after 10 minutes of suspension [31]. Meanwhile, it was reported that 2.1 million Americans had phones stolen in 2014 [9], and one out of ten smartphone owners were victims of phone theft [24]. In 2014, 3.1 million phones were lost [9]. This is particularly disconcerting for smartphone users who have to store sensitive data on their phones.

With full physical access to the stolen/lost phones, the attackers are able to launch inexpensive yet powerful attacks to access data in the DRAM chips, such as cold boot attacks [26] and bus monitoring attacks [11, 12]. Through such attacks, memory data in the smartphones will be extracted, resulting in private data leakage and even digital identity forgery.

The aforementioned physical memory attacks exploit the vulnerabilities of DRAM chips. In order to tackle these threats, existing solutions eliminate/limit clear-text sensitive data in DRAM chips, by *encrypting* them or *isolating* them. For example, Intel and AMD have their own memory encryption engines to transparently encrypt/decrypt memory transactions between the processor and the DRAM chip [21, 25]. However, we are not aware of any similar designs applicable for mobile devices. Isolation is often achieved by a *secure vault* implemented locally or remotely. Specifically, in a hardware security module (HSM), a dedicated hardware device is connected to or plugged into a computer to execute codes involving sensitive data. However, this solution requires hardware modification and is not suitable or available for mobile devices. Cloud-assisted approaches [31, 33] offload the code handling sensitive data to a remote cloud server. Therefore, they all suffer from the fundamental drawback in usability. The phones must always be connected to the Internet and the communication overhead also introduces additional cost. In intermittent or poor network environment, the users will not be able to utilize this approach.

In this paper, we explore an alternative approach – software-based memory encryption, which encrypts the entire DRAM chip [14, 29] or parts of a process [8, 18, 28]. Compared with aforementioned approaches, software-based memory encryption requires no modification to existing hardware, and are general to different computing workloads. However, it faces the hard trade-off between performance and usability. In particular, Encryption at coarse granularity typically requires no modification to exiting programs, but it introduces significant overhead because many nonsensitive data are unnecessarily encrypted. On the contrary, fine-grained encryption only spends CPU cycles to protect sensitive data, but it needs considerable efforts to reconstruct the program.

For example, Sentry [8] encrypts the address space of an Android app on a per-page basis without any modification to the app. However, more than double the overhead is observed. To alleviate the performance degradation, Sentry only activates memory encryption when the phone enters locked mode. However, protecting unlocked phone is equally important. We discuss this in Section 7.1.

To avoid wasting CPU cycles on the protection of nonsensitive data, Papadopoulos et. al [28] suggested selective memory encryption, in which the developers mark the sensitive static variables in source code, and invoke dedicated functions to allocate dynamic memory for sensitive data. However, this requires the developers to have a profound understanding of the program, and to pinpoint all the occurrences of sensitive data manually.

In this paper, we attempt to find a practical balance between performance and usability. Specifically, we introduce `MemVault`, a new mechanism for Android to encrypt sensitive user data in DRAM chips. `MemVault` features ***targeted encryption*** that selectively encrypts sensitive data objects, and at the same time requires minimal programmer involvement. To achieve this, `MemVault` leverages existing taint analysis tools to keep track of the propagation of well-marked sensitive data sources, and encrypts the tainted memory contents in the DRAM chip automatically. As such, a developer is only required to mark the sources of sensitive data with the provided Application Programming Interface (API). This typically brings in less than four lines of code insertion based on our evaluation with several popular open source apps. When a tainted object is accessed, `MemVault` redirects the access to a small non-DRAM memory vault, where our intermediate memory layer transparently decrypts the accessed object. Throughout the execution, only a small subset of sensitive data appear in clear-text inside the protected memory vault. Because the vault has limited capacity, to avoid frequent cryptographic operations, we use the Least Recently Used (LRU) algorithm to keep the most frequently used objects in the vault. The evicted cold objects are encrypted when they are synchronized to the DRAM.

The protection to the memory vault is two-fold. First, we explicitly erase the vault when an app is terminated. In this sense, we provide "private sessions" for security-critical apps. Note that our memory zeroization is independent of OS level zeroization, which was found to be ineffective [6]. Second, we hold the memory vault in a System-on-Chip (SoC) component (`iRAM`), which significantly raises the effective bar for physical exploits [8]. As a result, sensitive data in clear-text are also protected against physical attacks.

In summary, the contributions of our solution are three-fold:

- We present a new mechanism for Android virtual machine to control the propagation of clear-text sensitive user data with minimal efforts by programmers.
- We propose targeted encryption, which advances the state-of-the-art by only encrypting sensitive data that are worth protection. This is made possible by leveraging TaintDroid [10], a widely adopted dynamic taint analysis tool.
- We implement and evaluate the proposed system for the Android system. Our evaluation results show that `MemVault` effectively protects data on mobile devices from physical attacks, and imposes acceptable performance overhead (37.2% compared with 18.8% in TaintDroid alone).

2 BACKGROUND

2.1 Physical Attacks on Smartphones

Given that smartphones are frequently lost or stolen, it has become a major concern for smartphone users who need to process their private data on the phone. Such threats are particularly relevant for government or military users who frequently process classified data. When attackers have physical access to the devices, they can launch several inexpensive attacks, bypassing all the existing protection mechanisms, including OS-level access control.

A trivial way of retrieving user data is dumping the flash memory of a smartphone. However, since version 4.4, Android has begun to support Full Disk Encryption (FDE) [20]. As a result, by reading the flash memory one can only obtain encrypted data. A persistent attacker may further exploit the DRAM chip to launch more sophisticated attacks. For example, in a cold boot attack [17], the attacker is able to utilize the FROST forensic tool [26] to recover the entire DRAM content of a smartphone. Specifically, the attacker first fully charges and freezes the device. Then he swiftly unplugs the battery of the device to reboot the device. During this short time, the memory content is retained due to the *remanence effect* of DRAM [17]. When the device is booting, the attacker presses a key combination to enter the `Fastboot` mode, in which the FROST image is flashed into the recovery partition. Finally, the attacker selects the `Recovery Mode` from the phone's menu to boot the FROST forensic tool, which reads the DRAM contents.

In a bus monitoring attack, an attacker may attach a bus monitoring tool [11, 12] to the memory bus to intercept the memory transactions. Bus monitoring also facilitates side channel attacks. For example, an attacker can deduce the cryptographic keys by merely observing memory access patterns [32]. In DMA (Direct Memory Access) attacks, an attacker can also utilize the debugging port of an UART controller [8] to issue DMA read requests to the phone, bypassing security checks performed by the processor. This is because the DMA engine is independent of the processor, and can directly communicate with the DRAM chip.

2.2 SoC Components

Different from DRAM, SoC components are integrated into the chip. Therefore, they can exhibit better resilience to physical exploits. Utilizing cache [8, 35] or internal RAM [8, 18], many defense systems have been proposed.

CPU cache is a small amount of static RAM that sits in-between the processor and the DRAM. When a processor loses power, all the cache contents become invalid. As a result, cache is inherently immune to cold boot attack. Since cache data never appear on the memory bus, and DMA transfers data directly from DRAM without passing through cache, cache is also immune to bus monitoring and DMA attacks.

Internal RAM (`iRAM`) or on-chip RAM (`OCRAM`)[1] is another standard SoC component equipped in most ARM processors. It is a small static RAM that is tightly coupled with other system components. Typically, `iRAM` does not need to be initialized to work. Therefore, during device boot, most ARM devices first load their proprietary ROM to the `iRAM` to get executed. The ROM is responsible for initializing other system components, including DRAM. `iRAM` does not lose its contents inherently following a power loss. Instead, the device's ROM explicitly zeroes out its `iRAM` upon booting [8], rendering cold boot attacks against `iRAM` ineffective. In addition, as `iRAM` is encapsulated inside the SoC and has a dedicated data interface to the processor, it is extremely difficult to launch bus monitoring attacks to `iRAM`. Finally, TrustZone [2] and System Memory Management Unit (*SMMU*) [4] could be utilized to enforce memory protection of `iRAM` and to defeat DMA attacks.

To exploit cache for data storage, existing solutions lock a portion of the cache to prevent the data from being evicted into the DRAM [8, 35]. However, cache lockdown is an obsoleted feature, as indicated in the official technical reference manual of ARM's latest processors (Chapter 6.1 and 7.1 in [3]). Moreover, monopolizing a portion of cache causes system-wide performance impact. We consider it sub-optimal to exploit cache for data storage, and instead choose `iRAM` as our memory vault.

2.3 Android and TaintDroid

Android Design. Android is an open-source Linux-based mobile OS developed by Google. It is specifically optimized and tuned for resource constrained mobile devices. Each app runs in its own instance of virtual machine (VM) under a unique user ID. Thus, apps are essentially sandboxed, preventing their direct interference with the OS and other apps. Android apps are written with the Java programming language, which are compiled into a customized byte code format named Dalvik EXecutable (DEX).

In a VM, byte code is loaded into the memory along with core libraries, which include the framework libraries and those facilitating user interface building, graphic drawing, and database access, etc. The core libraries are implemented in either Java or C/C++ to improve performance. Besides, the Android OS starts a set of system services during booting. These services are system processes that are tied to lower-level device functionality and management of the system. The app interferes with these system processes through the Binder IPC mechanism to get services from the OS. Finally, applications can communicate and collaborate with each other, which is also enabled by the Binder interface.

An interpreter is responsible for translating DEX instructions into the machine's native instructions. Before version 4.4, the default Android interpreter is Dalvik, while its successor is called Android Runtime (ART). Among others, ART introduces a new Ahead-Of-Time (AOT) compilation technique, but reuses the DEX bytecode format to keep compatibility.

[1]The naming varies depending on the manufactures.

The DEX byte code is register-based, indicating that it has rich semantics, and requires fewer memory references during execution. A DEX method has a fixed number of registers determined during compilation time. Each local variable in a Java method is represented by a DEX method register. The Dalvik VM maintains an internal downwards-growing interpreter stack, which stores method frames, including method arguments, local variables, and return values, etc. A variable can either be a primitive type such as `long`, `int`, and `double`, or a reference to a Java object, which is essentially a pointer to a C++ data structure stored in the internal interpreter heap.

TaintDroid. TaintDroid [10] is a system-wide dynamic taint tracking and analysis tool developed for realtime privacy monitoring on Android phones. In a nutshell, it marks the sensitive data (i.e., taint source) when they are generated by the system services or Android APIs. If a tainted data is involved in a computation, the result will also be tainted (i.e., taint track). Finally, if any tainted data leaves the device into the network (i.e., taint sink), a system alert is issued.

To enable system-wide tracking, TaintDroid employs a multi-level approach. At variable-level, taking advantage of the rich semantics of each instruction, the interpreter is instrumented so that the involved destination variables can be tainted based on source variables. For local variables and method arguments, the taint tags are stored following the corresponding variables/arguments in the interpreter stack (see Figure 2). For object fields, a taint tag is stored with the actual field value in the interpreter heap. Note that for an array object, a single taint tag is associated with the entire array to save memory usage. To track propagation across applications, TaintDroid implements message-level tracking. To support taint tracking in system-provided native libraries and to retain taint information across app invocations, TaintDroid also implements method-level tracking and file-level tracking, respectively.

3 ASSUMPTIONS AND THREATS-IN-SCOPE

This work focuses on memory protection, and assume that data files stored in the flash are encrypted. Since version 4.4, Android has supported Full Disk Encryption (FDE), and this feature was enabled by default in version 5.0 [20]. Since the key used to encrypt the disk is also stored in memory, we assume exiting solutions such as ARMORED [16] to protect this key.

`MemVault` leverages TaintDroid, thus inherits all its security assumptions. Among others, we assume that the Android firmware is trusted. This include the kernel, native libraries, java DEX code, as well as app code (native and DEX). This assumption is supported by many security mechanisms developed by Google and the open source community. For example, the system partition of Android is protected by the `dm-verity` kernel mechanism, and the underlying Linux kernel has already been proactively protected in commercial products [5]. In TaintDroid, to track information flow in native code, each native method is instrumented manually based on its parameters. We also assume the apps in the smartphone do not intentionally leak user data. We do not assume the presence of secure deallocation techniques at OS level [7]. In `MemVault`, sensitive pages are managed within the Dalvik virtual machine.

We consider local attackers who have physical access to victims' smartphones. They can launch passive physical attacks which get a read-only copy of the phone's DRAM. This can be achieved by cold boot attacks [17, 26] and bus monitoring attacks [11, 12]. MemVault does not directly tackle DMA attacks, but relies on orthogonal approaches based on ARM TrustZone [2] or SMMU [4] to deal with DMA attacks.

4 SYSTEM DESIGN

4.1 Architecture Overview

Design Challenges. MemVault keeps track of sensitive user data, and eliminates their occurrences in the DRAM by encrypting them. One possible approach is to encrypt sensitive objects in situ. The benefit is that we do not need to change the memory layout of an app. However, to access an object, it has to be decrypted in the DRAM chip first. This unfortunately exposes them in clear-text in the vulnerable DRAM. Another approach is to move the entire memory segment to the iRAM. However, the capacity of the iRAM is usually restricted.

In our design, we encrypt all the sensitive data objects in the DRAM, and access their clear-text copies in the iRAM. We call the memory regions backed by iRAM as *memory vaults*. We encountered the following technical challenges when designing our system.

- Sensitive data can be stored in both primitive variables and data objects, which appear in the stack and heap of an app, respectively. Since stack and heap exhibit different characteristics in terms of memory management, we must adopt different strategies to regulate sensitive data stored in them.
- In Android, an object can be accessed by both interpreted code and native code. By instrumenting the Dalvik VM, we can implement a unified redirection layer that handles encrypted object accesses. Unfortunately, native code can access an object arbitrarily. It bypasses the redirection layer and directly accesses the encrypted objects in the DRAM, causing program crash.
- An object may contain references to other objects. When such an object is encrypted, garbage collection (GC) cannot access reference information without first decrypting it. If we frequently decrypt and encrypt objects for GC, significant overhead could occur.

Overview. Figure 1 shows an overview of the memory management with and without MemVault protection. We utilize TaintDroid [10] to track the propagation of the taint sources. The tainted contents, marked in red, appear in two locations. Tainted primitive variables are stored in the downwards-growing interpreter stack, together with other local variables and stack frame information. Tainted data objects appear in the interpreter heap. Without MemVault, all of them appear as clear-text in the DRAM chip (left half in Figure 1).

With MemVault, only a small working set of sensitive data appear as clear-text in a memory vault (i.e., iRAM in our design), while the DRAM chip only stores encrypted sensitive data. In particular, when a primitive variable is to be tainted, MemVault copies the current stack frame to a peer stack in the memory vault, and then works on

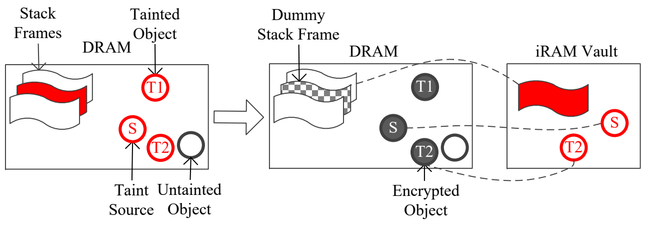


Figure 1: Data objects and stack frames in an Android app. In an unprotected Android app (left), all the data objects and stack frames appear in the DRAM. With MemVault protection (right), tainted data objects are encrypted in the DRAM (shown in shade), and tainted stacks in the DRAM contain no sensitive variables. The interpreter accesses the clear-text copies in the iRAM vault.

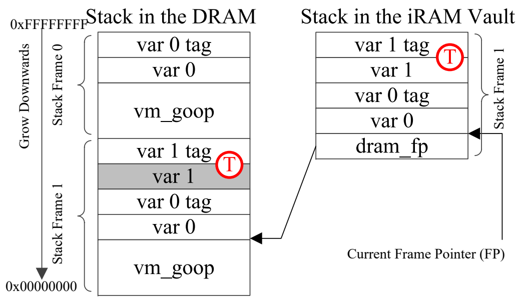


Figure 2: Stack memory layout. Red circled 'T' marks a tainted variable. Note that the current frame pointer points to the copy in the iRAM vault.

the new stack, leaving a dummy and nonsensitive stack frame in the DRAM. To save memory, MemVault switches back to the normal stack in the DRAM whenever all the taints are cleared in the current stack frame, or a new method without tainted arguments is invoked.

When an object is to be tainted, MemVault allocates a memory region of equal size in an iRAM heap, copies the object to the newly allocated memory, and then redirects the reference to the object. Because the size of the iRAM is limited, we employ the LRU algorithm to manage objects in the vault. Only the "hottest" objects are kept in clear-text in the vault. The "cold" objects are encrypted and evicted to the normal DRAM heap. For native methods needing access to the tainted objects, we manually instrument them to access the correct object copies.

When an app is started (i.e., onCreate()), an encryption key is randomly generated. It is also kept in the iRAM to defeat physical attacks. Then the interpreter allocates a heap in iRAM shared by all the threads of the process. On the contrary, a stack in the iRAM is allocated on demand – a new iRAM stack is allocated only if a thread has a tainted stack frame. Finally, when the app is terminated (i.e., onDestroy()), the encryption key is discarded and all the associated iRAM resources are erased and reclaimed.

4.2 Stack Protection

As described in Section 2.3, primitive variables such as `long` and `int` are stored in the downwards-growing interpreter stack. TaintDroid tracks the taint propagation to the method arguments and local variables.

The left side of Figure 2 depicts the memory layout of a typical stack in Android with TaintDroid. In the figure, two methods are invoked, represented by two stack frames. During execution, some variables in stack frame 1 are tainted. To prevent sensitive data from appearing in the DRAM, we have two straightforward approaches. First, we may move the entire stack to the memory vault. However, copying such a large data structure (16KB in TaintDroid) is time-consuming; in addition, a lot of nonsensitive data are kept in the valuable `iRAM`. Second, we may encrypt primitive variables in situ. However, then we would need to maintain a data structure for dereferencing each encrypted variable, which is costly because the maintained data structure is larger than the variable itself.

In `MemVault`, we maintain a mirror stack in the vault, which only stores tainted stack frames. The stack in the vault is only activated when the current stack frame is tainted. As shown on the right side of Figure 2, the stack frame 1 is mirrored in the vault, which contains the real sensitive data.

Next, we explain how we maintain this "dual" stack. When a variable is to be tainted, `MemVault` allocates a mirroring stack frame in the vault, copies the contents of the current stack frame to the vault stack, updates the current frame pointer to the new stack frame, and finally resumes the app execution. Note that the "store" instruction leading to taint propagation is never operated on the corresponding variable in the DRAM stack. The stack frame in the vault contains method variables and their taint tags, plus a pointer to the corresponding stack frame in the DRAM (represented by `dram_fp`). The stack maintenance information (represented by `vm_goop`), such as the pointer to the previous stack frame, is only kept in the DRAM stack. Before executing code that triggers stack maintenance (e.g., method invocation/return), `MemVault` restores the current frame pointer to the value stored in `dram_fp`. Therefore, existing Android code can utilize `vm_goop` to manage the interpreter stack. This keeps our modification to the stock Android system minimal.

Because the capacity of `iRAM` is limited, `MemVault` minimizes vault memory usage. Specifically, vault stacks are only assigned to the threads that handle tainted data. In our experiments, it turns out that for a typical Android app, among more than 10 auxiliary system threads, there is only one main thread that processes tainted data.

4.3 Heap Protection

Compared with variables in the stack, data objects are much larger and are scattered in the heap. We protect tainted objects on the DRAM with in-situ encryption, and maintain a dedicated data structure, `Trampoline`, for dereferencing encrypted objects. As shown in Figure 3, the `Trampoline` data structure is pointed to by an added `trampoline` field in the corresponding C++ data object.

Specifically, an additional heap in the memory vault is shared among all the threads of an app. Only the "hottest" tainted objects are stored in clear-text in this vault heap. When the interpreter

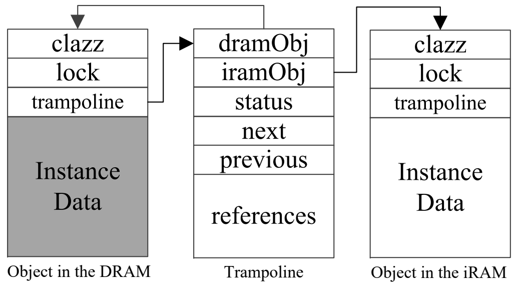


Figure 3: Objects on heap. Data in shade are encrypted.

accesses an object, it first queries the `trampoline` field of that object. A `null trampoline` field indicates that the object is not tainted, and accordingly the interpreter accesses the DRAM object normally. Otherwise, the object is tainted and encrypted in the DRAM heap, and the added `Trampoline` data structure is queried to further dereference the object. In particular, the `status` field indicates whether the object is hot or not (i.e., whether there is clear-text copy of that object in the vault heap). If so, the access is redirected to the corresponding copy referenced by the `iramObj` field. Otherwise, according to the LRU algorithm, `MemVault` evicts certain cold data objects in the vault heap (if the vault is full), allocates a new memory region in the vault heap, and decrypts the DRAM copy (referenced by `dramObj`) to the vault heap. Note that the `next` and `previous` fields in the `Trampoline` data structure are used to maintain a double-linked list that facilitates the implementation of the LRU algorithm.

Key Management. Since the size of an object varies and may not align to 16 bytes, we utilize the AES cryptographic algorithm in counter mode (CTR) as stream cipher. The key is randomly generated by the built-in OpenSSL random number generator when the app is started. It is stored in a reserved `iRAM` page, along with other intermediate key schedule materials. Note that each key is only effective in the life-cycle of a process. Each app invocation generates a different key. The initialization vector (IV) is chosen as the virtual address of the object in the DRAM. In this way, `MemVault` does not need to store an IV for every tainted object while still being able to ensure a different IV for each object. Note that cross-app attack is not considered in this paper, therefore there is no security violation of using the same set of IV for two different apps.

In the following, we present the problems raised by native code and garbage collection, and describe how we deal with them.

Dealing with Native Code. In Dalvik, a `String` is implemented as a fixed-length object with a `char` array reference. Because a `String` object itself does not contain any data, we never encrypt it. A tainted `String` object is indicated by the taint tag in the referenced `char` array. If all the methods of the `String` class are implemented solely by Java code, by instrumenting the interpreter, `MemVault` can correctly locate the correct `char` array and redirect the access as explained before. Unfortunately, to improve efficiency, the Dalvik virtual machine optimizes the implementation of the `String` class with a great deal of native code. Without awareness of the redirection layer in the interpreter, the native code would mistakenly access the encrypted copy of the tainted char array.

We address this issue by manually instrumenting these native methods. In particular, for all the native methods that operate on the raw char arrays, we manually insert a redirection code to locate the correct array. In Listing 1, we list a code snippet showing the modifications we make to the String.charAt method implemented in native code. Note that the line 3 (in green) overwrites the pointer of the char object.

```
1 offset = dvmGetFieldInt((Object*) arg0, STRING_FIELDOFF_OFFSET);
2 chars = (ArrayObject*)dvmGetFieldObject((Object*)arg0,
      STRING_FIELDOFF_VALUE);
3 + chars = String_getRealValue(chars);
4 pResult->i = ((const u2*)(void*)chars->contents)[arg1 + offset];
```

Listing 1: Manually inserting redirection code for the `String.charAt` native method.

Apart from native methods in the String class, other native methods involving primitive arrays also need manual instrumentation. In our prototype, we found these methods mostly located in the classes Charsets (e.g., toUtf8Bytes method) and System (e.g., arraycopy method). In total, we patched seven methods for Android 4.4.3. As a result, the evaluated apps do not crash.

Dealing with Garbage Collection. Garbage collection (GC) periodically scans for live objects, and deallocates the dead objects. To be able to mark all the live objects, the Dalvik virtual machine needs to traverse through all the object references, starting with the GC roots. Unfortunately, such traversals cannot pass through encrypted objects, because the references in encrypted objects are opaque. To address this issue, one option is to incorporate the redirection code into GC, and decrypt the involved objects for GC traversing. However, this necessitates decrypting all the tainted objects for each GC invocation, leading to a considerable performance overhead.

In MemVault, we follow a different approach that avoids the aforementioned overhead. In particular, in the Trampoline data structure, we maintain an array of clear-text object references contained in the corresponding data object, as shown in Figure 3. These references are updated whenever a reference field is overwritten in the actual object. When a garbage collection occurs, we only need to supply it with this references array. Note that references are not sensitive themselves. In the sweeping phase, the dead objects in the vault heap are cleared and freed, along with the DRAM copies.

4.4 Memory Vault Protection

We have shown how MemVault constrains sensitive data within the memory vaults. This section explains how we secure the memory vaults themselves.

First, the memory vault is backed by an SoC component – the iRAM chip in our prototype. We have explained how iRAM defeats physical memory attacks in Section 2.2. Because the vulnerable DRAM chip never stores clear-text sensitive data, and the SoC iRAM that stores decrypted sensitive objects is resistant to physical attacks, the entire smartphone is protected from physical attacks.

Second, in the Linux kernel, we set the VM_IO flag for the virtual memory regions backed by iRAM. As a result, the Linux OS regards the vaults as memory-mapped I/O regions, which are excluded from the core dump image when a crash occurs. Therefore, the attack targeting core dump [22] is prevented.

Third, when an app is terminated or suspended, MemVault explicitly erases the associated memory vaults. Since sensitive data only appear in the vault, this is a VM-wide erasion. Note that our erasion is independent of OS-level secure deallocation [7], which makes our system tolerant to OS vulnerabilities that reuse uncleared pages [6].

5 IMPLEMENTATION

We have implemented a MemVault prototype for Android 4.4.3. The prototype runs on an i.MX 6Quad SABRE experiment board, which features a four-core ARM Cortex-A9 processor, with 1 GB DDR3 DRAM and 256 KB iRAM. The iRAM is temporarily used by the device ROM to initialize other SoC components during system bootup. After that, the OS is free to use all the iRAM.

5.1 iRAM Management

The iRAM chip is an SoC component with very limited capacity. In our experimental board, we can only use 256 KB or 64 pages in iRAM. Our implementation maximizes its usage.

The mirrored stack in the memory vault has four pages, which is the same as the DRAM stack in TaintDroid [10]. The rationality here is that the size of the stack in the vault can never exceed that of a normal stack in the DRAM. The vault heap, which holds hot objects, also occupies four pages and is further maintained by the dlmalloc allocator [23]. As a result, we can readily use the popular malloc/free interfaces to allocate and reclaim the memory for tainted objects. When a malloc invocation fails, the least used object in the vault heap is evicted. The eviction continues until there is enough space for the new allocation request.

At a low level, a Linux driver is responsible for handling requests of iRAM memory issued by the interpreter. In particular, to request a virtual memory region backed by the iRAM, the interpreter opens a special device iram, and calls the mmap system call to associate the newly returned virtual memory with a region in the iRAM. The Linux driver calls the vm_iomap_memory internal kernel function to build page tables for the new virtual memory area (vma). By default, the resulting vma is marked with the VM_IO flag, which tells the OS that the corresponding area is a memory-mapped I/O region.

5.2 Virtual Machine Instrumentation

MemVault is implemented as a modified Dalvik VM. Table 1 lists the instrumentations we made to the semantics of the involved DEX bytecode instructions. In the table, R and E stand for return and exception variables, respectively, which are maintained by the interpreter. $\otimes$ stands for an opcode. For a detailed explanation of the DEX bytecode instructions, we recommend readers to refer to the online DEX bytecode documentation [13] and TaintDroid [10]. In the bottom of the table, we have explained the meanings of the newly added instrumentations.

Here we explain some instrumentations that are not straightforward to understand. Instructions (*unary-op* v_A v_B), (*binary-op* v_A v_B), (*binary-op* v_A v_B v_C), and (*binary-op* v_A v_B C) do not have the S_VS instrumentation that appears in many others. This is because

Instruction Format	Instruction Semantics	Instrumentation
const-op v_A C	$vA \leftarrow C$	`S_DS`
move-op v_A v_B	$v_A \leftarrow v_B$	$\emptyset$
move-op-R v_A	$v_A \leftarrow R$	`S_DS & S_VS`
return-op v_A	$R \leftarrow v_A$	$\emptyset$
move-op-E v_A	$v_A \leftarrow E$	`S_DS & S_VS`
throw-op v_A	$E \leftarrow v_A$	$\emptyset$
unary-op v_A v_B	$v_A \leftarrow \otimes v_B$	`S_DS`
binary-op v_A v_B v_C	$v_A \leftarrow v_B \otimes v_C$	`S_DS`
binary-op v_A v_B	$v_A \leftarrow v_A \otimes v_B$	$\emptyset$
binary-op v_A v_B C	$v_A \leftarrow v_B \otimes C$	`S_DS`
aput-op v_A v_B v_C	$v_B[v_C] \leftarrow v_A$	`R & M_VH`
aget-op v_A v_B v_C	$v_A \leftarrow v_B[v_C]$	`R & S_DS & S_VS`
sput-op v_A f_B	$f_B \leftarrow v_A$	`R & M_VH`
sget-op v_A f_B	$v_A \leftarrow f_B$	`R & S_DS & S_VS`
iput-op v_A v_B f_C	$v_B(f_C) \leftarrow v_A$	`R & M_VH`
iget-op v_A v_B f_C	$v_A \leftarrow v_B(f_C)$	`R & S_DS & S_VS`
invoke-op $v_C \ldots v_G$ m_B	call $m_B(v_C \ldots v_G)$	`I`

`R`: Redirect access to the object if necessary.
`S_VS`: Switch to vault stack, if working on DRAM stack and the resulting stack is tainted.
`S_DS`: Switch to DRAM stack, if working on vault stack and the resulting stack is untainted.
`M_VH`: If the destination object is newly tainted, move it to vault heap.
`I`: If the thread is working on the DRAM stack and any of the callee arguments is tainted, switch to vault stack. If the thread is working on vault stack and none of the arguments is tainted, switch to the DRAM stack.

Table 1: Interpreter Instrumentations

in these instructions, variable v_A is assigned with a value determined by other values on current stack. If the current stack is not tainted, there is no tainted variable to taint v_A, and the resulting stack is definitely untainted. In addition, the instruction (*binary-op* v_A v_B) does not need the `S_DS` instrumentation. In this instruction, the only mutated variable is v_A, which retains all its existing taint tags, indicating that none of existing taint tags can be erased.

5.3 Tainting APIs

In TaintDroid [10], sensitive objects are defined by the system. They are tainted when going through the predefined framework APIs. For example, in the framework API `handleLocationChanged`, the returned location information is automatically tainted.

In `MemVault`, we protect a broader range of sensitive data, including user generated data. Obviously, the Android framework cannot determine whether a user input is sensitive or not – only the developer and the user know. Therefore, we require developers to identify the origin of this kind of user-generated sensitive data in the source code, and explicitly invoke our APIs to taint them.

We expose two APIs for developers to taint an object. The design of APIs is based on the characteristics of the sensitive data. The first API, `MemVault.addTaintArray` is specifically designed to handle array-oriented sensitive objects. In fact, the actual sensitive data that a user inputs is typically contained in a low-level array object, and the array is further encapsulated into another high-level object. For example, in an Android app with a login interface, the high-level object representing the typed password is an instance of the class `EditText`. However, the actual password is contained in a `char` array, which is a field of the `SpannableStringBuilder` object contained in `EditText`. The `MemVault.addTaintArray` API takes an `EditText` object as argument, locates the actual inner array that holds sensitive data, and finally taints the array.

The second API is `MemVault.addTaintGeneral`. As indicted by its name, it is used to taint a general data object. This is useful when an object contains primitive sensitive variables. In this case, the object itself is tainted (rather than an inner array). In both APIs, if the object was not tainted earlier, the API also moves the sensitive object to the vault heap and encrypts the copy in the DRAM.

6 EVALUATION

We are interested in several questions. First, how easy is it to incorporate `MemVault` for protecting existing Android apps? We expect a system that requires minimal re-engineering effort. To this end, we looked into four open-source Android apps that process sensitive data, and manually labeled the taint sources with our APIs. Second, is the proposed solution effective? In other words, is the tainted data really encrypted in the DRAM? Third, how much performance degradation does `MemVault` bring to the protected apps? Lastly, to what extend does `MemVault` affect the overall performance of the system given that taint tracking is a system-wide operation?

The evaluation results were measured in the aforementioned i.MX 6Quad SABRE experiment board running an AOSP Android 4.4.3 OS with FDE enabled. We note that although FDE was only enabled by default in Android 5.0, it has become available in Android 4.4.3. As `MemVault` is built on top of TaintDroid, we treated the performance observed from native Android as our baseline, and compared it with those observed from TaintDroid and `MemVault`.

6.1 Adoption

We have instrumented four open-source apps to benefit from the protection of `MemVault`. WordPress[2] is a content management system. Users can view and maintain their blogs through the client app. BankDroid[3] is a hub of bank accounts. Users can retrieve bank transactions of different banks in a single app. For these two apps, `MemVault` aims to protect the login credential and bank account information. KeePass[4] is a password manager. It utilizes a master key to encrypt user passwords of different websites. We use `MemVault` to protect both the master key and stored password entries. K-9[5] is a popular email client for Android. We use `MemVault` to protect the email account information and email contents.

Accounting for the `import` statements, we need only two new lines of code (LOC) for WordPress and BankDroid, and four new LOC for KeePass and K-9. Take the K-9 email client as an example; when the app is launched, the stored account information is loaded from database to memory as a string. The account information takes the format of "protocol+://address:password@server". Obviously, this string contains sensitive data and should be marked as a taint source. We used our API to mark the string as sensitive, as shown in Listing 2. Similarly, when an email message is loaded from the database as a byte array, we label that array as well.

```
private synchronized void loadAccount(Preferences preferences) {
    Storage storage = preferences.getStorage();
    mStoreUri = Base64.decode(storage.getString(mUuid + ".storeUri
      ", null));
   + MemVault.addTaintArray(mStoreUri);
    ...
}
```

Listing 2: Tainting a sensitive object.

[2]https://github.com/WordPress/WordPress
[3]https://github.com/liato/android-bankdroid
[4]https://github.com/bpellin/keepassdroid
[5]https://github.com/k9mail/k-9

6.2 Data Exposure Prevention

To examine if an app under protection still has sensitive data in the DRAM, we directly dumped its address space while it was running, and searched for the occurrences of sensitive data. We devised two ways to get its memory image. First, we compiled the `fmem` kernel module[6] into the Linux kernel. With `fmem`, we were able to dump the entire physical memory of the device. Note that the `fmem` kernel module was only used in our experimental validation to emulate a physical attack. In our attack model, we assume attackers lack the ability to inject arbitrary code in the kernel space. Second, we registered in the Dalvik virtual machine a signal handler for memory dumping. When the signal was received, the interpreter explicitly dumped the memory regions corresponding to the Dalvik heap and stack.

We input pre-known account information/passwords/email contents into each tested app, and used our dumping tool to capture the memory images. Table 2 summarizes the results. In the table, the results in columns **DRAM** and **IRAM** were obtained by analyzing memory images retrieved by feeding the `fmem` module with the physical memory ranges of DRAM and `iRAM` respectively. The memory images used in column **Heap&Stack** were obtained by our signal handler method. Without the protection of `MemVault`, we observed a bunch of sensitive data in both the **DRAM** and **Heap&Stack**. With `MemVault`, we did not observe any occurrence of sensitive data in the heap and stack segments of the running process. This indicates that `MemVault` successfully eliminates all the occurrences of sensitive data within the VM. In the `iRAM`, as expected, we were able to locate lots of sensitive data. Furthermore, we did not find in the DRAM image any sensitive data.

We also searched for the sensitive data in the system after we terminated the test apps. As soon as the apps were terminated, we could not find any occurrence of sensitive data in both the `DRAM` and `iRAM`. This indicates that sensitive data were completely erased by `MemVault` after the apps were terminated.

6.3 Overhead of Protected Apps

We evaluated the overhead imposed to individual apps by measuring the delay when loading an app, and runtime overhead.

App Load Time. We modified the Android framework code to record the time spent on launching an app. The start time is recorded in the `startSpecificActivityLock` method when the activity manager starts the app, while the end time is recorded at the time when the `attachBaseContext` method is called, which means the app has been displayed on the screen.

We measured the app launching time for 100 times, and calculated the average values, which are shown in Table 3. Both TaintDroid and `MemVault` impose small overheads in starting an app. This is because Android handles UI mainly through native code.

Runtime Overhead. Evaluating the performance overhead of the four apps is challenging, because none of the apps provides a straightforward quantitative output of its performance. We instead evaluated the overhead imposed to individual apps by measuring additional power consumption when the app performs the same

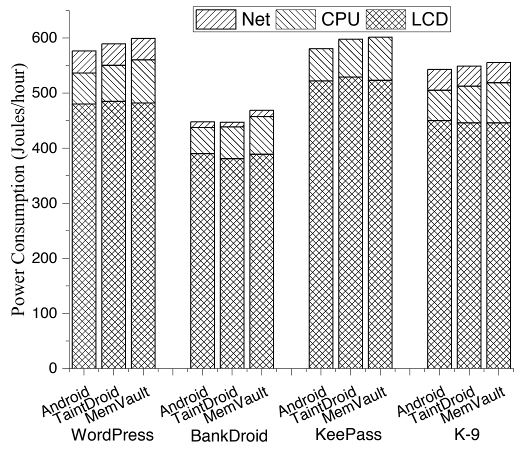


Figure 4: Power Consumption (the lower, the better).

amount of tasks. Our assumption is that the more CPU cycles are spent on the task, the more overhead is imposed.

We used an Android UI exerciser Monkey [1] to launch each app and let it perform a set of typical workloads. Each task was repeated for at least one hour. In the meanwhile, we kept the PowerTutor power monitor [34] running in background, and recorded the power consumption of each tested app.

Figure 4 shows the energy consumption for each app. Both TaintDroid and `MemVault` consumed more energy on CPU than native Android did, but incurred little or no overhead on LCD and Network. This is due to the additional CPU cycles spent on taint tracking and object encryption. While TaintDroid consumed about 18.8% more power compared with native Android, `MemVault` consumed 37.2% more. Since `MemVault` is built on TaintDroid, the additional energy consumption (i.e., 18.4%) on CPU indicates the performance overhead of `MemVault` compared with TaintDroid. Note that in this experiment, we ran apps with tainted objects; therefore, there were frequent object encryption/decryption operations. We show in Section 6.4 that for non-sensitive apps, `MemVault` only introduces 9.6% additional overhead compared with TaintDroid.

6.4 System-wide Overhead

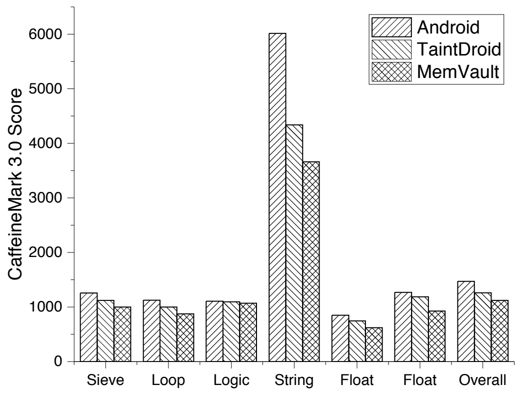
As mentioned earlier, taint tracking is a system-wide operation. `MemVault` incurs an inevitable performance overhead even if an app does not process any sensitive data. In this section, we show the results of two popular Android benchmarks running without tainted data. CaffeineMark 3.0 measured the performance overhead at Java level. Since `MemVault` was implemented as instrumentation to the Dalvik VM, the results of CaffeineMark precisely reflected the overhead introduced by our instrumentation. In addition, we utilized Geekbench3 to measure the introduced overhead to workloads that simulate real-world scenarios.

As shown in Figure 5a, `MemVault` exhibits small performance degradation in all the tests except `String`. As explained in TaintDroid, the exceptionally high overhead is due to "additional memory comparisons that occur when the JNI propagation heuristic checks for string objects in method prototypes" [10]. In `MemVault`, because of additional redirection checks, the overhead is further magnified.

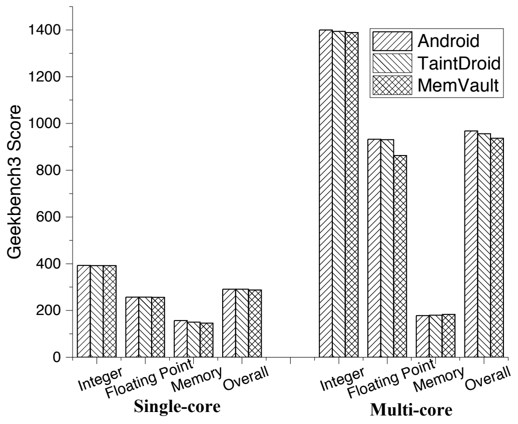
[6]https://github.com/NateBrune/fmem

APP	DRAM		IRAM		Heap&Stack	
	Android	MemVault	Android	MemVault	Android	MemVault
WordPress	Password	none	n/a	Password	Password	none
BankDroid	Account Number & Password	none	n/a	Account Number & Password	Account Number & Password	none
KeePass	MasterKey & Password	none	n/a	MasterKey & Password	MasterKey & Password	none
K-9	Password & Email	none	n/a	Password & Email	Password & Email	none

Table 2: Identified Sensitive Data in DRAM, IRAM and Interpreter Heap/Stack



(a) CaffeineMark Benchmark.



(b) GeekBench3 Benchmark.

Figure 5: System-wide Benchmarks (the higher, the better).

	WordPress	BankDroid	KeePass	K-9
Android	985	239	79	269
TaintDroid	1001	247	82	277
MemVault	1008	248	83	277

Table 3: App Start Time (in *ms*).

Overall, the measured overhead is 23.9% (1120 v.s. 1471), compared to 14.3% (1260 v.s. 1471) in TaintDroid. GeekBench3 is inclined to real-world workloads. For example, the integer test includes many crypto and compression algorithms, and floating point test includes image processing algorithms. They are mostly implemented in native code. Results in Figure 5b show that for apps involving massive native code, the overhead is negligible.

7 RELATED WORK

7.1 Software-based Memory Encryption

Memory encryption is a straightforward approach to protect clear-text sensitive data from physical attacks to the DRAM chip. Cryptkeeper [29] and RamCrypt [14] attempt to encrypt the whole address space or data segments of a process. However, to execute the program, a small working set must be kept in clear-text, rendering data in this region insecure.

Bear [18] addresses the working set problem by employing `iRAM` to temporarily store the clear-text data. The authors conducted comprehensive experiments to evaluate the impact of memory encryption with different encryption granularities. However, it works on a clean-slate micro-kernel, and thus the implication to commodity environments is not revealed. Papadopoulos et. al [28] implemented both full memory encryption and selective memory encryption based on compiler-based instrumentation. CaSE [35] constructs a trusted execution environment to run self-contained applications within the CPU cache. Because the application must fit into one way of the cache (32KB), CaSE is not scalable for complex Android apps that requires more than tens of megabytes of memory. Sentry [8] hacks the Linux page table handler to implement page-based memory encryption on Android. As encryption is performed at a coarse granularity, many nonsensitive data are unnecessarily encrypted, imposing high runtime overhead. Therefore, the protection is only activated when the phone enters suspended mode. The argument is that if the phone is lost in unlocked status, the attacker is already able to obtain user credentials (such as contacts) through User Interface (UI). However, there exists a class of credential data that are not displayed in the UI. For example, a password is never shown in the UI by default. But such data indeed appear as clear-text in the DRAM, and are hence vulnerable to physical attacks. To keep such credential data secret, it is necessary to provide the same protection to unlocked phones.

In `MemVault`, we track the propagation of sensitive data and selectively encrypt data that are really worth protecting, regardless of the status of the phone. Table 4 summarizes exiting memory encryption solutions. Note that the performance statistics are collected from relevant papers and vary depending on concrete settings.

	Architecture	Software Environment	Granularity	Completeness†	Code Modification	Unlimited Memory Usage	Overheads
Cryptkeeper [29]	x86	Linux	4KB	✗	None	✓	1.09x∼9.00x
RamCrypt [14]	x86	Linux	4KB	✗	None	✓	1.25x∼2.70x
Bear [18]	ARM	Micro-Kernel	16B∼128KB	✓	Significant	✓	1.50x∼3.40x
Work in [28]‡	x86	Linux	16B	✓	None/Significant	✓	1.17x∼10.00x+
CaSE [35]	ARM	Self-contained	Whole App	✓	Significant	32KB Limitation	1.03x
Sentry [8]	ARM	Android	4KB	✓	None	✓	1.48x∼2.74x
MemVault	ARM	Android	Object	✓	Trivial	✓	1.37x

†: Completeness represents whether the working set of process is protected. ‡: The paper proposed two modes which exhibit very different characteristics.

Table 4: Comparison with Other Software-based Memory Encryption Solutions

7.2 Hardware-based Memory Encryption

In recent years, commodity processors began to incorporate hardware components to support hardware-based memory encryption. Two representative solutions are Intel SGX [25] and AMD SEV [21]. In SGX, the memory controller is augmented with a crypto engine that transparently encrypts the transaction between the processor and a special region of DRAM. Unfortunately, these solutions mainly target the cloud-centric server market and are missing in ARM-powered mobile devices. Last, it is also shown that hardware features can be used to assist memory encryption [15].

7.3 Cloud-assisted Solutions

Cloud-assisted solutions attempt to eliminate local sensitive data on the device. In CleanOS [31], the credential data objects are also tracked by TaintDroid. After a certain time of inactiveness, the tainted credential data objects are encrypted by a key that is later escrowed in the cloud. When these objects are accessed, only authenticated users could obtain the escrowed key in the cloud to decrypt the credential objects in the phone. However, due to the trade-off between security and usability, there exists a timing window during which the credential data appear in clear-text in the phone. Moreover, CleanOS does not protect sensitive data on the interpreter stack. In TinMan [33], the code handling credential data is offloaded to the cloud server, and the cloud server stores the real credential data. In the prototype, TinMan can only protect user credentials that require an input box in GUI. A major concern with cloud-assisted solutions is availability issue. In particular, in an intermittent or poor network environment, users may not be able operate their phones normally. `MemVault` does not need cloud service. Instead, sensitive data are stored locally in secure `iRAM`.

8 LIMITATIONS AND FUTURE WORK

The current prototype only supports Android 5.0 or lower, because TaintDroid was designed for the Dalvik VM, which has been replaced by the new ART runtime. Fortunately, the DEX bytecode format was reused to retain compatibility. Therefore, it is still feasible to insert instrumentation code based on instruction semantics during the compilation. In fact, TaintART [30] has been proposed to achieve the same goal of TaintDroid but targeted the new ART runtime. It is possible to port `MemVault` for newer Android based on TaintART.

`MemVault` provides developers with straightforward APIs to mark the taint sources, which is easy to adapt as evaluated in Section 6.1. However, developers still need to be slightly involved. SUPOR [19] and UIPicker [27] show that it is possible to automatically find the sensitive user input by detecting the semantic information within the application layout resources and program code. To free developers from the burden of manually labeling the taint sources, we plan to incorporate these techniques to `MemVault`.

`MemVault` is unable to protect sensitive data processed by the native code. For example, the buffer of the touchscreen driver to receive password input cannot be tainted in our system. We assume the Android kernel and framework effectively erase these sensitive data in time. For example, the buffer can be immediately cleaned after being fetched by the app. In this way, the stolen or lost phones cannot contain any sensitive data.

Lastly, only encrypting sensitive data may leak structural information of an app, such as memory layout. This could be potentially leveraged by attackers.

9 CONCLUSIONS

`MemVault` minimizes the exposure of sensitive data on the DRAM chip of mobile devices, defeating physical memory disclosure attacks in which attackers have physical access to the phones. Utilizing TaintDroid [10], `MemVault` tracks the propagation of sensitive data and constrains them within a centralized memory vault in the `iRAM`. `MemVault` runs locally on mobile devices, so it avoids the latency and communication overhead introduced in a cloud-assisted solution. `MemVault` selectively encrypts the tainted objects, so the performance is improved significantly compared with coarse-grained page-based memory encryption solutions for smartphones.

We have implemented a prototype of `MemVault`. Through extensive experiments, we show that with a little effort from programmers to label the sensitive data inputs, `MemVault` effectively minimizes the occurrences of sensitive data in the system, while the introduced overhead is acceptable for real-world apps.

ACKNOWLEDGMENTS

The authors would also like to thank the anonymous referees for their valuable comments and helpful suggestions. Peng Liu was supported by ARO W911NF-13-1-0421 (MURI), NSF CNS-1505664, NSF CNS-1814679, and ARO W911NF-15-1-0576. Jingqiang Lin was partially supported by National Natural Science Foundation of China (Award No. 61772518), and Cyber Security Program of National Key RD Plan of China (Award No. 2017YFB0802100).

REFERENCES

[1] Android Developers. 2017. UI/Application Exerciser Monkey. (2017). https://developer.android.com/studio/test/monkey.html.

[2] ARM Ltd. 2009. Security Technology Building a Secure System Using TrustZone Technology (white paper). (2009).

[3] ARM Ltd. 2014. ARM Cortex-A57 MPCore Processor Technical Reference Manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0488d/index.html.

[4] ARM Ltd. 2019. The Arm System Memory Management Units. https://developer.arm.com/products/system-ip/system-controllers/system-memory-management-unit.

[5] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *ACM CCS'14, 2014*. ACM, 90–102.

[6] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. 2004. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*.

[7] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. 2005. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *USENIX Security '05*.

[8] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2015. Protecting Data on Smartphones and Tablets from Memory Attacks *(ASPLOS '15)*. 177–189.

[9] Consumer Reports. 2015. Smartphone thefts drop as kill switch usage grows. http://www.consumerreports.org/cro/news/2015/06/smartphone-thefts-on-the-decline/index.htm.

[10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones *(OSDI'10)*.

[11] EPN Solutions. 2017. Analysis tools for DDR1, DDR2, DDR3, embedded DDR and Fully Buffered DIMM modules. http://www.epnsolutions.net/ddr.html.

[12] FuturePlus System. 2006. DDR2 800 bus analysis probe. http://www.futureplus.com/download/datasheet/fs2334_ds.pdf.

[13] Google inc. 2017. Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.

[14] Johannes Götzfried, Tilo Müller, Gabor Drescher, Stefan Nürnberger, and Michael Backes. [n. d.]. RamCrypt: Kernel-based Address Space Encryption for User-mode Processes *(ASIA CCS '16)*. 6.

[15] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. 2015. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 3–19.

[16] J. Götzfried and T. Müller. 2013. ARMORED: CPU-Bound Encryption for Android-Driven ARM Devices *(ARES '13)*. 161–168.

[17] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. 2008. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *17th USENIX Security Symposium*. 45–60.

[18] Michael Henson and Stephen Taylor. 2013. Beyond full disk encryption: protection on security-enhanced commodity processors *(ACNS '14)*. Springer, 307–321.

[19] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps. In *24th USENIX Security Symposium*. USENIX Association, Washington, D.C., 977–992.

[20] Google inc. 2017. Full-Disk Encryption. https://source.android.com/security/encryption/full-disk.

[21] David Kaplan. 2016. AMD x86 Memory Encryption Technologies. USENIX Association, Austin, TX.

[22] Vadim Kolontsov. 1996. Solaris (and others) ftpd core dump bug. http://insecure.org/sploits/ftpd.pasv.html.

[23] Doug Lea and Wolfram Gloger. 1996. A memory allocator. http://g.oswego.edu/dl/html/malloc.html.

[24] Lookout. 2014. Phone Theft in America: What really happens when your phone gets grabbed. https://blog.lookout.com/phone-theft-in-america.

[25] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution *(HASP '13)*. Article 10, 1 pages.

[26] Tilo Müller and Michael Spreitzenbarth. 2013. FROST: Forensic Recovery of Scrambled Telephones *(ACNS '13)*. 373–388.

[27] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. 2015. UIPicker: User-Input Privacy Identification in Mobile Applications. In *24th USENIX Security Symposium*. Washington, D.C., 993–1008.

[28] Panagiotis Papadopoulos, Giorgos Vasiliadis, Giorgos Christou, Evangelos Markatos, and Sotiris Ioannidis. 2017. No Sugar but all the Taste! Memory Encryption without Architectural Support *(ESORICS '17)*. Springer, 362–380.

[29] P. A. H. Peterson. 2010. Cryptkeeper: Improving security with encrypted RAM. In *2010 IEEE International Conference on Technologies for Homeland Security (HST)*.

[30] Mingshen Sun, Tao Wei, and John C.S. Lui. 2016. TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime *(CCS '16)*. 331–342.

[31] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. 2012. CleanOS: Limiting Mobile Data Exposure with Idle Eviction *(OSDI '12)*. USENIX, Hollywood, CA, 77–91.

[32] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology* 23, 1 (01 Jan 2010), 37–71.

[33] Yubin Xia, Yutao Liu, Cheng Tan, Mingyang Ma, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. TinMan: Eliminating Confidential Mobile Data Exposure with Security Oriented Offloading *(EuroSys '15)*. Article 27, 16 pages.

[34] Z Yang. 2012. Powertutor-a power monitor for android-based mobile platforms. *EECS, University of Michigan, retrieved September* 2 (2012), 19.

[35] N. Zhang, K. Sun, W. Lou, and Y. T. Hou. 2016. CaSE: Cache-Assisted Secure Execution on ARM Processors. In *2016 IEEE Symposium on Security and Privacy (SP)*. 72–90.