LazyTainter : Memory-Efficient Taint Tracking in Managed Runtimes

by

Zheng Wei

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

LazyTainter : Memory-Efficient Taint Tracking in Managed Runtimes

Zheng Wei

Master of Science

Graduate Department of Computer Science

University of Toronto

2014

The leakage of private information is of great concern on mobile devices since they contain a great deal of sensitive information. This has spurred interest in the use of taint tracking systems to track and monitor the flow of private information on a mobile device. Taint tracking systems impose memory overhead, as taint information must be maintained for every piece of information an application stores in memory. This memory cost is at odds with the growing number of low-end, memory-constrained devices, which will also make up the majority mobile device growth in emerging markets. To allow taint tracking to benefit a broader range of mobile devices, we present LazyTainter, which is a memory-efficient taint tracking system designed for managed runtimes. To implement LazyTainter, we enhanced TaintDroid to use *hybrid taint tracking*, which combines lazy and eager tainting. Our experimental results demonstrate that LazyTainter can reduce heap usage by as much as 26.5% when compared to TaintDroid while imposing a less than 1% increase in performance overhead.

Acknowledgements

First and foremost, I'd like to thank my supervisor Professor David Lie for his utmost patience and support. His guidance inspired me with many novel ideas. And his encouragement gave me great help during my Master study.

I am very thankful to Professor Ashvin Goel, who gave me invaluable feedback on my research project which helped me work in the right direction.

I am also grateful to James Huang, Afshar Ganjali, Ben Kim, Wei Huang, Michelle Wong and Sukwon Oh with whom we worked together happily.

Speical thanks go to Mike Qin, Xu Zhao, Yongle Zhang and Reza Mokhtari for giving me various kinds of help.

Finally I'd like to deeply thank my parents for their everlasting support in my life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Mobile devices (smartphones and tablets) are quickly replacing traditional personal computers (PCs) as people's primary computing devices. Equipped with rich hardware sensors and an operating system capable of running 3rd party applications, mobile devices provide capabilities and convenience unmatched by any other types of devices. According to the latest statistics by Flurry Analytics [10], an average user spends almost 3 hours a day using applications and browsing the web on their mobile devices.

This heavy use of mobile devices, coupled with their innate ability to collect and concentrate personal information means that they are a risk to personal privacy. To help understand how and when private information is being collected and leaked by these devices, various static [21, 17, 1] and dynamic [9, 12, 6] approaches have been proposed. While static analysis imposes no runtime overhead, it is inherently imprecise [27]. In contrast, dynamic taint-tracking, which uses *taint tags* to track whether a value contains or is derived from private information or not has been shown to be effective at detecting personal privacy violations. While emulation-based native binary instrumentation incurs considerable overhead [19, 29], the execution overhead of taint tracking can be reasonable for a managed runtime. For example, TaintDroid [9] has made taint tracking realtime with a 14% performance overhead. As a result, TaintDroid's taint tracking functionality has been used in a number of research proposals [2, 12, 24].

Aside from execution overhead, taint tracking also imposes memory overhead, as taint tags must be maintained for each value stored in memory. This is a major concern for mobile devices because they have much less memory than traditional PCs. While the amount of physical RAM on devices has grown steadily, growth in the smartphone market has shifted significantly towards low-end devices [13], with a large number of such devices projected to be sold in emerging markets [25]. According to Google, millions of entry-level devices around the world still have as little as 512MB of RAM [14], leading Google to launch Project Svelte [4] specifically to address this issue. Even with this limited amount of memory, a non-trivial portion of the physical memory is still be used by graphics hardware and the operating system, leaving even less memory for applications. To reduce the memory burden of taint tracking for these low-end devices, we propose *LazyTainter*, a taint-tracking system that reduces memory overhead while maintaining reasonable performance overhead.

The storage for taint information can be allocated either *eagerly* (ahead of time), or *lazily* (on-demand). Eager allocation generally improves performance because the taint-tracking system is free to place the taint storage at a predetermined location relative to the memory value it is tracking taint for, thus simplifying access to the taint information. However, eager allocation wastes memory in exchange for better performance because the amount and granularity of taint storage cannot by dynamically adjusted in response to the tainting behavior of the application. This reasoning implies that there is a trade-off between memory efficiency and runtime overhead that taint tracking systems must choose between.

In this paper, we propose a hybrid approach of taint tracking which is able to combine the benefits of both eager and lazy tainting without the costs. To demonstrate our results, we present *LazyTainter*, which implements the same taint propagation policy as TaintDroid [9]. By using eager-tainting for frequently accessed memory locations, lazy tainting for the rest of memory, and carefully optimizing the storage and allocation of taint information, LazyTainter can reduce heap usage by as much as 26.5% when compared to TaintDroid while imposing a less than 1% increase in performance overhead.

The rest of the paper is organized as follows: Chapter 2 provides background information of Android and TaintDroid, Chapter 3 presents the design of LazyTainter, Chapter 4 gives implementation details, Chapter 5 shows the experimental results, Chapter 6 provides a discussion, Chapter 7 describes related work and Chapter 8 concludes this paper.

Chapter 2

Background

2.1 Android

Android is an operating system designed for mobile devices including cellphones and tablets. Its architecture consists of four layers from top to bottom: *applications*, *application framework*, *libraries* and *Linux kernel*. The top two layers (applications and application framework) are mainly written in Java, while the bottom two layers (libraries and Linux kernel) are mainly written in C/C++. What sits in between application framework and libraries is the Android runtime consisting of a process virtual machine called *Dalvik* and a set of Java core libraries.

Dalvik is mainly written in C++ and executes its own DEX bytecode format [22]. Android applications, which are written in Java, are first compiled into Java bytecode and then converted to DEX bytecode before installation on a device. Each application runs in its own Dalvik instance and an application sandbox is implemented at process boundary to improve security. Processes can communicate with each other through Binder IPC.

Our work is mainly involved with Dalvik. Dalvik is designed for resource-constrained devices and has some key differences from the traditional JVM. Dalvik is a 32-bit register-based VM with 64-bit values formed by adjacent register pairs. Instead of having a fixed number of virtual registers, each method allocates the registers it needs in its stack frame on the execution stack. All register operations occur on these values stored on the stack. Dalvik also has its own garbage collected (GC) heap, which provides for long-term storage of objects. As of the Android version we use, the default GC is a concurrent mark-and-sweep GC, with the option of switching to a copying GC at compile time.

Dalvik has its own instruction set. The opcode size is 8-bit so there will be at most 256 opcodes from `00` to `FF`, though some of them are not used at the moment. Instructions are not limited to a particular type. The same instruction is often used to operate on different types of values if they have the same width. Dalvik instructions have clear semantics and can be roughly classified into several groups. For example, *move/arithmetic/logic* instructions do computation on the stack, *new* instructions create objects on the heap, *get/put* instructions move data between the stack and the heap and *if/goto* instructions manage the control flow. Operations in each group may have different variants. For example, `iget` and `iput` instructions have `quick` variants, whose execution is faster because they use optimized inputs.

Much of the design on Android is focused around conserving memory. Android devices don't have a swap partition, so any memory pressure will cause the kernel to kill processes to reclaim memory. This can be disruptive to the user if the user actually wants to use the killed process in the near future. Even though Android has optimized process creation, starting an application still imposes some delay. In addition, killing the wrong application only to have it restarted by the user causes undesirable drain on the battery.

The Dalvik heap is a virtual memory range acquired with `mmap`. To manage heap memory, Dalvik uses the `dlmalloc` [15] memory allocator. When an application explicitly requests memory with `new`, Dalvik allocates memory to the application from the memory allocator. The amount of currently allocated memory on the heap is called *Heap Alloc*. When a garbage collection is triggered, Dalvik will walk the heap, free unused space and return them to the memory allocator. This shows up as a reduction in the heap usage of the application. The amount of free space on the heap is called *Heap Free*. The total size of the heap is called *Heap Size*, which equals to the sum of Heap Alloc and Heap Free. Dalvik doesn't compact the heap so it cannot shrink heap size when there is used space at the end of the heap. Therefore Heap Alloc is a better measurement of heap usage than Heap Size because there can be quite a bit free memory on the heap.

Although Android is Linux-based, the process creation model is different from that of traditional Linux. In Android there is a nascent Zygote process which preloads a common set of classes and performs initialization that is common to all applications. Application processes are created by forking from Zygote to save initialization time. Memory pages of Zygote are shared by many application processes. In Android, memory pages can be classified into four kinds: shared dirty, private dirty, shared clean and private clean [3]. Shared memory is used by more than one processes, while private memory is used by only one process. Clean memory is unwritten memory, while dirty memory has been written to by the process. Private dirty memory is the most expensive because it's exclusively used by one application and cannot be readily discarded when the operating system needs to reclaim memory. Dalvik heap memory is usually private dirty, though some heap memory can be shared dirty as processes can be forked. Since private dirty memory is expensive, it is a good measure of an application's memory cost.

2.2 TaintDroid

TaintDroid [9] is an information-flow tracking system designed for the Android OS to monitor privacy leaks in realtime. It leverages Android's virtualized environment to provide an efficient and system-wide dynamic taint tracking system with fine-grained labels. TaintDroid taints data acquired from sensitive APIs and identifies when tainted data is sent to the network interface. The principal component of TaintDroid is variable-level tracking, which is implemented in the Dalvik interpreter. TaintDroid defines its own taint propagation logic, covering explicit data flows in almost all the instructions.

Because a mobile device has different sources of sensitive information, such as location, IMEI and microphone, TaintDroid represents each of them using a different bit (called a *taint marking*) in a 32-bit vector (called a *taint tag*). Taint markings don't interfere with each other so up to 32 types of sensitive information can be tracked at the same time. TaintDroid stores the taint tag for a variable adjacent to the variable itself, providing spatial locality. Recall that Dalvik stores registers on the stack. For 32-bit register values (as shown in Figure 2.1), taint tags are interleaved between values so that a register `fp[i]` becomes `fp[2*i]` in TaintDroid's stack layout. For 64-bit register values, a double word `(fp[i], fp[i+1])` now becomes `(fp[2*i], fp[2*i+2])` with its taint tag stored in `fp[2*i+1]`.

Taint tags of object fields which exist on the the heap are also interleaved so that a 32-bit field with offset `k` now has offset `2*k`. The interleaving is done by modifying a function that computes field offsets when a class is linked to Dalvik. For 64-bit fields, if the base address of an object is `base` then a double word `(base[k], base[k+1])` becomes `(base[2*k], base[2*k+1])` with its taint tag stored in `base[2*k+2]`. From this description we see that both the stack size and the object size are effectively doubled. An exception is array objects. In TaintDroid, each array has only one taint tag that is shared by all elements of the array. Therefore, tainting for arrays has minimal memory overhead but is more coarse-grained than tainting for other types of objects. String objects contain several fields, one of which is an array. TaintDroid taints a string object by tainting the array it contains.

Besides variable-level taint tracking, TaintDroid also has message-level taint tracking to propagate taints through IPC channels, method-level taint tracking to propagate taints through native methods and file-level taint tracking to propagate taints through secondary storage. LazyTainter uses the exact same mechanisms as TaintDroid to track taints through these channels and thus we omit a detailed description here. Finally,

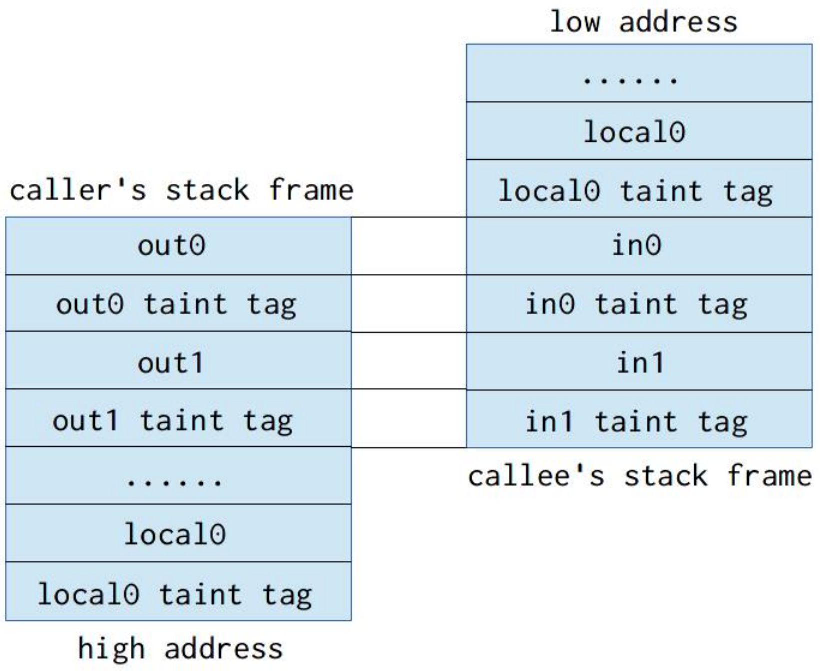


Figure 2.1: TaintDroid stack layout.

since TaintDroid is designed to work in a virtualized runtime environment, it doesn't allow third-party native libraries to be loaded because it cannot track taints for arbitrary native code efficiently. However, native system libraries in the firmware are supported by TaintDroid as they are in the trusted computing base.

Chapter 3

Design

One of the challenges in designing a taint tracking system is finding the right trade-off between precision, speed and memory overhead. Intuitively, improving the performance in one of these properties usually comes at a cost of decreasing the performance in one or both of the other two. For example:

- We can use per-element tainting instead of per-array tainting for arrays. Since this would track taint on a finer granularity, it will reduce false positives and increase precision. However, this will require more memory to store the fine-grained taint information and may also result in lower speed as more work needs to be done to propagate the larger amount of taint information.

- We can use 1-bit taint tags instead of 32-bit taint tags. The reduction in taint tag size may reduce memory overhead because there is less taint information to store. However, a taint tracking system with 1-bit taint tags can only indicate whether a variable is tainted or not without the ability to distinguish different taint sources, such as location, IMEI, etc., resulting in lower precision.

- We can track taints on a per-object basis instead of a per-field basis. This coarse-grained taint tracking will result in less taint information, and thus lower memory overhead, but at the cost of decreased precision since this can result in false taint propagation across fields in an object.

Depending on the characteristics of the underlying system on which the taint tracking system is built, there may be opportunities to increase the performance in one property without great negative consequences in the other two. Thus, a good taint tracking system will exploit these opportunities to maximize performance across the three properties.

Finding a design that provides a good trade-off is especially important for mobile devices because they have limited resources. On these devices, speed and memory cannot be as easily sacrificed for the sake of precision as devices with more plentiful resources. Current Android taint tracking systems like TaintDroid already make such trade-offs. For example, TaintDroid reduces the precision of taint tracking for arrays and strings to save memory and increase speed.

The goal of this paper is to show that one can trade a negligible amount of speed for a significant saving in memory without affecting the precision of a taint tracking system. To achieve this goal, we introduce *lazy-tainting* and apply it to TaintDroid. We use TaintDroid as a starting point as the TaintDroid designers have already made some good design choices to make the system practical, as shown by the many projects that have used or incorporated TaintDroid [2, 12, 24]. As a result, our design goal is to produce a system that has the same precision as TaintDroid, but is more efficient in memory usage with only a minimal decrease in speed. We use TaintDroid as the foundation due to its good design and sound implementation, but we believe our ideas can be applied to other virtualized environments as well.

3.1 Lazy Tainting Granularity

A key design decision that affects the memory overhead of a taint tracking system is the granularity of taint tracking. This decision has already been considered by the designers of TaintDroid, which trades decreased precision for better memory overhead for arrays and strings. Another place where memory overhead of TaintDroid may be reduced, is to track taints at a per-object granularity instead of a per-field granularity. However, as previously mentioned, this causes a loss of precision, which violates our design goal.

To overcome this apparent limitation, we hypothesize that mobile applications on Android exhibit localized tainting behavior, which means taints are only propagated within a small group of objects that are related to private information. Since there is rising public awareness of the privacy implications of mobile application use, we further hypothesize that Android applications, whether benign or malicious, are more likely to send private information to servers as soon as possible rather than to store it in memory for a long time. Together, these factors should cause most Android applications to have a majority of untainted objects and the number of tainted objects should be very small.

If this is the case, then we can decouple taint granularity from precision. Tainted and untainted objects don't have to be tracked at the same level of granularity. The system can use cheap coarse-grained per-object tainting for the vast majority of completely untainted objects and only the expensive fine-grained per-field tainting for the tiny minority of objects with tainted fields.

To evaluate our hypothesis, we modified the mark-and-sweep garbage collector in TaintDroid to profile tainted objects. Specifically, we add two counters, `CntNum` and `CntSize`, to the Dalvik garbage collector and set them to zero when garbage collection is started. During the mark step of the garbage collection, we check the fields of an object when it's being marked to see whether any of them is tainted. If a tainted field is found, we increase `CntNum` by 1 and increase `CntSize` by the object size. We also allocate two other counters to count the total number and size of live objects regardless of whether they are tainted or not. We only count objects when it's being marked for the first time so that the same object is counted only once. We restrict the profiling to data objects only and do not include array objects.

App \ Ratio	CntNum	CntSize
PhotoGrid	0.00	0.00
The Weather Channel	0.00	0.00
Twitter	0.00	0.00
Facebook	0.01	0.03
IMDb	0.00	0.00
Solitaire	0.00	0.01
Horoscope	0.00	0.01
Voice Search	0.00	0.01
Wish	0.01	0.01

Table 3.1: Tainted object ratio. `CntNum` is the ratio of the number of objects with at least one tainted field to all objects. `CntSize` is the ratio of the size of objects with at least one tainted field to all objects.

We tested nine applications in total. For each application, we count the number and size of tainted objects and live objects, respectively. Then we calculate the ratio between tainted objects and live objects in both number and size and tabulate the results in Table 3.1. From the results, we can see that only a tiny portion of objects contain any tainted fields. In addition, the total size of these objects is a small percentage of overall objects. For many applications the tainted ratio is within 1% in both number and size. This result indicates that Android applications are very likely to have a vast number of untainted objects and that a system that uses coarse-grained per-object tainting by default and then *lazily* switches to fine-grained per-field tainting when any field in an object becomes tainted has the potential to yield significant memory savings.

3.2 Hybrid Taint Tracking

Now that we have established that lazily switching from coarse-grained to fine-grained tainting can yield memory savings, we now turn our attention to designing a taint tracking system that incurs low speed overhead. Lazy taint tracking imposes overhead on the speed of execution because the memory for taint storage is not allocated at the same time as the memory for the object itself. In eager tainting implementations, such as TaintDroid, the storage for a value's taint tag is usually allocated at a fixed offset to the value itself, making access to a value's taint tag a simple matter of pointer arithmetic. However, with lazy taint tracking, an object that has no tainted fields will have no taint storage allocated to indicate that no fields are tainted. Only when a field in the object acquires taint will the system lazily allocate storage for the more expensive per-field taint tracking. Because the taint storage is allocated lazily, it cannot be placed at a fixed offset from the base object, but must instead be linked to the base object using pointer, as illustrated in Figure 3.1. This means that when accessing taint, the taint tracking system must first check a pointer to see if any field in the object is tainted, and if so dereference the pointer to access the taint tags for the individual fields. Such checking and deferencing of pointers results in more instructions executed at runtime, as well as worse cache locality.

Given this trade-off between lazy and eager tainting, a key design insight is that the two kinds of tainting can be applied in parallel but to different memory regions. Dalvik has divided its address space into several regions, such as heap and stack. The decision here is which kind of tainting should be used for which memory region. To make this decision, we first make several observations.

First, the stack is much smaller than the heap. In Dalvik, the maximum stack size is 256KB (plus a 768B `STACK_OVERFLOW_RESERVE` region). However, the heap can be as large as 1GB. Even if a smaller soft limit is configured on the device, the heap is still many orders of magnitude larger than the stack. From this observation, we conclude that even if only eager, fine-grained tainting is used all the time for the stack, the memory overhead from doing so is still small and bounded.

Second, all computation occurs on the stack, and values in object fields must be loaded onto the stack before computation and stored back to the object afterwards. As a result, values on the stack are generally accessed more frequently than values on the heap. From this observation, we can infer that using eager, fine-grained tainting for stack values would help minimize the performance impact of the hybrid tainting.

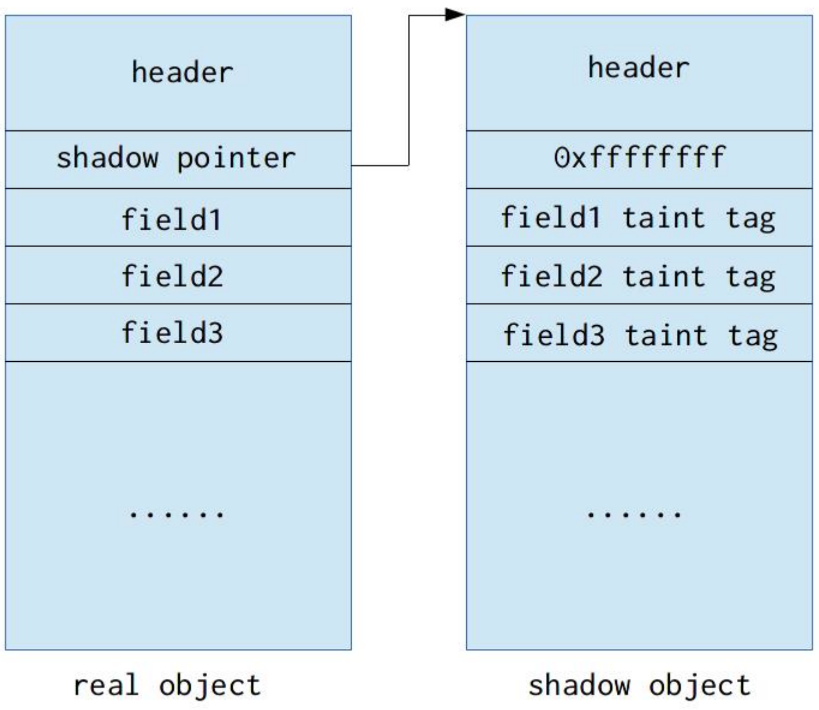


Figure 3.1: A tainted real object in LazyTainter. Shadow objects are used to store taint tags if the real object has any tainted fields.

Finally, Dalvik only stores primitive values and object references on the stack, while objects themselves are stored on the heap. As a result, all stack values have fixed size (32-bit or 64-bit). In addition, the stack is used continuously, making it more amenable to eager tainting. Since the previous chapter established that lazily tainting objects will yield benefits, and objects are only stored on the heap, lazy tainting would be most naturally applied to the heap.

Together, these three observations suggest a *hybrid* approach, which uses eager, fine-grained taint tracking for the stack and lazy taint tracking for the heap. Since the stack is heavily used for computation and the heap for storage, we should minimize performance overhead in the stack and minimize memory overhead in the heap. We note that this hybrid approach is not beneficial under all conditions. When objects are tainted, they actually cost more storage when lazily allocated than if eagerly allocated because there is the extra overhead of storing the pointer. Thus, the lazy approach only yields benefits if applications are likely to have many untainted objects, which has already been established earlier.

3.3 Managing Taint Storage

While allocating taint tags on-demand has been used by several other taint tracking systems [28, 29, 30], they all use a fixed granularity for taint tracking. In general, these techniques are on-demand in that they programmatically map a fixed-size taint tag storage area [28], or simply not starting computation-heavy instrumentation when taints haven't been introduced into the system [30]. In addition, these techniques are generally byte-level and do not take into account type information of the program. In contrast, LazyTainter is designed for a managed runtime of an object-oriented language. This presents several challenges, as well as opportunities.

An opportunity is the natural grouping of related program values into objects, as specified by the programming language. This grouping leads to a natural way of switching between coarse-grained and fine-grained taint tracking. The challenges are how to efficiently allocate and, in particular, deallocate taint storage.

To allocate storage, we leverage the existing heap allocator in Dalvik. Since only objects can reside on the heap, we aggregate taint tags of all fields in an object (called *real object*) into a separate *shadow object*. To visualize this, we again refer to Figure 3.1. To facilitate implementation, the shadow object has the same type and size as the real object and the taint tag of a field in the real object is put at the same offset in the shadow object. This ensures that we have enough taint tag storage in the shadow object for all the fields in the real object.

Since we need to get and set its taint tag when accessing a field in the real object, we must link the shadow object to the real object. The shadow object is linked to the real object by adding a *shadow pointer* in the object header. In a real object, this pointer points to a shadow object (if any of the fields in the real object is tainted) or is set to `NULL` signifying that none of the fields in the real object are tainted. Since this pointer resides in a normal object, it's automatically set to `NULL` when an object is created because memory allocated to an object is cleared by default in Dalvik. This is the intended behavior for a real object because a newly created object has no taints. When one of its fields gets tainted, a shadow object is dynamically created and linked to the real object, and the taint tag corresponding to the tainted field is set appropriately in the shadow object. From now on, setting the taint tag of a real object field will first locate the shadow object via the shadow pointer and then put the taint tag at the correct offset in the shadow object. Getting the taint tag of a real object field works in a similar way.

When objects are deallocated, we must also be sure to deallocate any associated shadow objects. In managed runtimes, unused objects are not explicitly deallocated by the programmer, but are instead identified and deallocated by a garbage collector. During garbage collection the runtime will iteratively traverse all objects on the heap by following pointers to find all reachable objects. Since shadow objects are linked to real objects, when the real object becomes unreachable, the shadow object automatically becomes unreachable. However, the use of shadow objects still presents a challenge when it comes to the traversal of reachable real objects by the garbage collector.

As mentioned earlier real objects and their associated shadow objects have the same type. This is because the total number of types a program will use is not known to Dalvik before running the program, so we cannot statically reserve some portion of the type space for shadow objects. Because shadow objects and real objects have the same type, the garbage collector cannot tell whether it is visiting a real object or a shadow object. However, the garbage collector must treat real and shadow objects differently. For real objects, the garbage collector should continue to follow pointers, while for shadow objects, all fields are taint tags, so the garbage collector should just mark the object and return to the parent object. To enable the garbage collector to differentiate between real and shadow objects, we set the space reserved for the shadow pointer in the shadow object to an invalid address `0xffffffff` during the creation of the shadow object. We then modify the garbage collector to check this field when visiting an object. If it is set to `0xffffffff`, the garbage collector treats the object as a shadow object. This challenge arose because LazyTainter uses dynamic taint allocation on a garbage collected runtime. In systems that use eager tainting exclusively or do not have garbage collection, this problem does not arise. However, the garbage collector is helpful here because it allows taint storage to be deallocated in an automatic and efficient way.

Chapter 4

Implementation

We implemented LazyTainter on Android `4.1.1_r6`, which is the latest version that TaintDroid supports. Since we want to have exactly the same taint propagation logic as TaintDroid, we don't modify any stack operations or method invocation instruction handlers. Similarly, we don't modify the message, method and file-level taint tracking implementation. Instead, we only modify the object layout, the heap-related instruction handlers and the garbage collector. To satisfy alignment requirements and to support different levels of tainting at the same time, we also need to modify some *primitive wrapper* functions, which we describe in more detail below.

Since LazyTainter uses the basic framework of TaintDroid, it has the same limitations as TaintDroid. Third-party native libraries cannot be loaded. Native system libraries can be loaded but native methods are tracked in a coarse-grained fashion. In addition, only explicit data flow is tracked to avoid taint explosion. LazyTainter is implemented on the portable interpreter version of TaintDroid. We describe our reasoning for this choice in Chapter 6.2.

```
struct Object {
    ClassObject*    clazz;
    u4              lock;
    //  Added in LazyTainter.
    Taint           taint;
};

struct ClassObject : Object {
    u4  instanceData[CLASS_FIELD_SLOTS];
    const char*     descriptor;
    char*           descriptorAlloc;
    ...
};

struct ArrayObject : Object {
    u4              length;
    u8              contents[1];
};

struct DataObject : Object {
    u4              instanceData[1];
};
```

Listing 4.1: Object implementation in Dalvik.

4.1 Objects in Dalvik

We first describe how Java objects are implemented in Dalvik, including object types, layout and size. Then we describe different kinds of objects which can exist in a taint tracking system.

Dalvik defines several C++ structs to specify the layout of Java objects in the application, as shown in Listing 4.1. There are three kinds of Java objects in Dalvik: class objects, array objects and data objects, whose headers are defined by structs `ClassObject`, `ArrayObject` and `DataObject`, respectively. Class objects are objects of type `java.lang.Class`. Array objects are arrays of any type. Data objects are non-class and non-array objects. All objects share the same header defined by the common parent struct `Object`. Object fields are placed next to the header.

Currently Android is expected to run on 32-bit platforms, and the size of `Object` is 8 bytes. Since heap memory is managed by `dlmalloc`, an additional 4 bytes of bookkeeping data is required per object. Dalvik further requires all objects to be 8-byte aligned. Therefore, we can calculate the size of an object as `(4 + 8 + field_size)` rounded up to a multiple of 8 bytes. For example, an object with 0 or 1 `int` field is 16 bytes, while an object with 2 `int` fields is 24 bytes. An object can have both primitive values and object references as its fields and it may have zero or more fields.

In terms of taint tracking, objects can be classified into two categories:

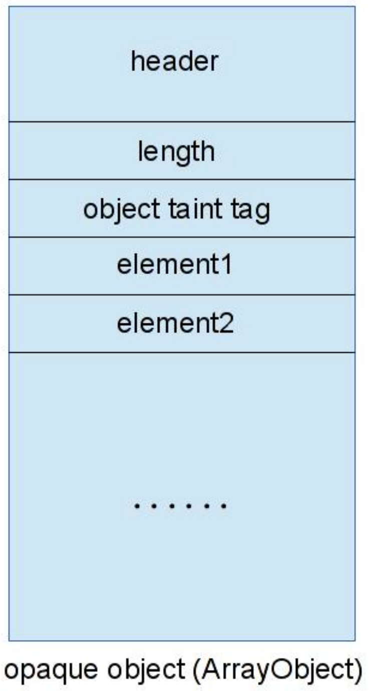
- *Transparent object.* A transparent object doesn't have a per-object taint tag. Instead, each field has its own taint tag and is individually tainted.
- *Opaque object.* An opaque object only has a per-object taint tag which is shared by all its fields. Fields don't have their individual taint tags.

Ideally, all objects should be transparent to make taint tracking as precise as possible. In practice, however, some objects are made opaque to reduce taint tracking overhead. For example, in TaintDroid data objects are transparent while array objects are opaque. A data object maintains a taint tag for each field, while an array maintains only one taint tag for all elements.

Opaque objects have minimal memory overhead because the only additional storage is a 32-bit taint tag. In contrast, transparent objects have their fields interleaved with taint tags and so the object size is almost doubled. Figure 4.1 gives the layout of opaque and transparent objects in TaintDroid.

4.2 Field Reuse

At first glance, lazy tainting can be easily implemented by adding a 32-bit pointer in `Object`. But TaintDroid already adds a 32-bit taint tag field in `ArrayObject`. We notice that a taint tag and a pointer have exactly the same size. Since opaque objects never need the pointer, memory can be saved if we use the same field for dual purposes: a



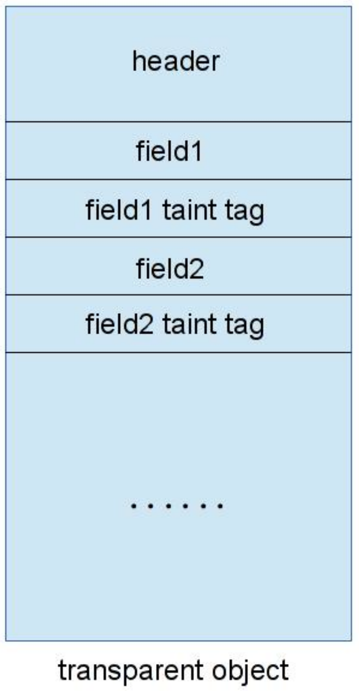


Figure 4.1: Object layout in TaintDroid.

```
typedef union Taint {
    u4 tag;                 // opaque.
    Object* taintObj;       // transparent.
} Taint;
```

Listing 4.2: Field reuse with `union Taint`.

pointer in a transparent object, and a taint tag in an opaque object. To do this we introduce a union `Taint` in the Dalvik source code as shown in Listing 4.2.

Then we add a field of type `Taint` in `Object` as shown in Line 5 of Listing 4.1 and remove the original 32-bit taint tag field in `ArrayObject`. By reusing this field, we in fact save 8 bytes on each array object because Dalvik requires array elements to be aligned at an 8-byte boundary. If we kept both fields, the total size of an array header (excluding `dlmalloc` bookkeeping data and array elements) will be 20 bytes, which is then padded up to 24 bytes. If we reuse it, we only need 16 bytes. An 8-byte per-object memory saving may look small, but there can be many array objects on the heap so the total

Size (Bytes) \ #int	0	1	2	3	4	8
Android	16	16	24	24	32	48
TaintDroid	16	24	32	40	48	80
LazyTainter	16	24	24	32	32	48

Table 4.1: Object size with varying numbers of fields.

saving is nontrivial.

A nice property is that LazyTainter with field reuse *never* uses more memory than TaintDroid in a taint-free environment and in some cases, uses the same amount of memory as vanilla Android. Table 4.1 shows object size with a varying number of `int` fields.

4.3 Garbage Collector

Now we have three kinds of objects on the Dalvik heap: opaque objects, transparent objects and shadow objects. Our next task is to modify the garbage collector (GC) so that a shadow object is automatically recycled with the corresponding real object, which must be a transparent object itself. In Android `4.1.1_r6`, Dalvik uses by default a mark-and-sweep GC. We must modify the GC so that the following properties are satisfied:

1. *No dangling pointer.* A shadow object must be alive if the real object is alive.

2. *No memory leak.* A shadow object must be recycled when the real object is recycled.

The simplest solution is to follow the shadow pointer and mark the shadow object when the real object is being marked during a garbage collection. However, we cannot blindly follow the shadow pointer because:

1. The shadow pointer in an opaque object is in fact not a pointer but a taint tag.

2. The shadow pointer in a shadow object is invalid.

In either case, blindly following the shadow pointer may result in a segmentation fault. To deal with the first case, we add a check in the GC's object marking function to prevent it from following the shadow pointer if the current object is an opaque object. Since an object holds a pointer to its defining class, we can look up its type to see whether it's an array object. If so, then we know the object is an opaque object. Otherwise, the object is a transparent object and we need to check whether the shadow pointer is invalid. The only invalid pointer that we may have introduced into the system is `0xffffffff` so we just need to check against this value. This deals with the second case. If the shadow pointer is `0xffffffff`, then this object is a shadow object and we don't follow the pointer. Otherwise, this object is a real object and we follow the pointer if it isn't `NULL`. The value `0xffffffff` can never be a valid shadow pointer because objects are always aligned at 8-byte boundaries.

To facilitate the concurrent garbage collection, Dalvik divides heap memory into a set of fixed-size cards and maintains a card table. Dalvik has a write barrier requesting any change to an object field to mark the card on which the object resides as dirty. Since the shadow pointer acts as a field in garbage collection, we must also mark the card as dirty whenever a transparent object gets tainted.

Finally, during the second phase of marking, objects on dirty cards will be scanned and their fields will be traversed to reach other objects. We must not traverse if the current object is a shadow object because its fields are actually taint tags. Thus, we again need to test the shadow pointer against `0xffffffff` before traversing. We must use `0xffffffff` to label shadow objects because it's perfectly legal and highly possible that a transparent object has a `NULL` shadow pointer.

4.4 Instruction Handlers

We only need to modify the two Dalvik instructions (and their variants) that move data in and out of data objects: `IGET` and `IPUT`. In TaintDroid, given the field offset, these two instructions assume the adjacent field in the same object will contain the corresponding taint tag. In LazyTainter, we modify `IGET` to first check whether the shadow pointer is `NULL`. If so, it returns a clean taint tag. Otherwise, it follows the shadow pointer and retrieves the taint tag at the same offset in the shadow object. `IPUT` is a little more complicated. If the transparent object is clean and the input value is also clean, then nothing is done. If the transparent object is clean and the input value is tainted, then a shadow object is dynamically allocated. If the transparent object is already tainted, it just uses the existing shadow object instead of creating a new shadow object. We apply the same logic in other places whether we get or set an object field, such as when implementing reflection in Java. We don't have to consider opaque objects here because Dalvik instructions have clear semantics and opaque objects are completely handled by two different instructions `AGET` and `APUT` (and their variants).

4.5 Primitive Wrappers

Double-width (8-byte) fields are required to be aligned at an 8-byte boundary in Dalvik. Since the original Dalvik object header is 8 bytes, the first double-width field is just next to the header. Some internal functions depend on this assumption implicitly. However, since we have introduced another 4-byte shadow pointer to the object header, the header is now 12 bytes and the first double-width field will have an offset of 16 bytes. Therefore, there is a 4-byte gap between the end of the header and the beginning of the double-width field. This breaks the implicit assumption and the double-width value will not be correctly processed by these functions.

We identify places where Dalvik relies on this assumption and find that the majority of code deals with the boxing and unboxing of primitive types. An example of boxing and unboxing would be converting from an `int` to an `Integer` and vice-versa. We fixed these issues by manually shifting the offset by 4 bytes before accessing the field.

Chapter 5

Evaluation

To cover different types of mobile devices, we evaluated LazyTainter on two Android devices: Galaxy Nexus (maguro) and Nexus 7 (grouper, 2012 version). The Galaxy Nexus is a smartphone and the Nexus 7 is a tablet. Both devices have a 1.2GHz ARM Cortex-A9 processor but Galaxy Nexus is dual-core while Nexus 7 is quad-core. Both devices have 1GB RAM and both can run Android version `4.1.1_r6`, the version on which TaintDroid and LazyTainter are implemented. A major difference between these two devices is that Galaxy Nexus has an IMEI number, which has been shown to be a piece of private information that many applications access [9].

We evaluate three aspects of LazyTainter. First and foremost, we evaluate the memory savings that LazyTainter provides with its lazy-tainting technique. Second, we evaluate the performance overhead of LazyTainter over TaintDroid. Finally, to show that LazyTainter is functionally equivalent to TaintDroid, we perform a comparative evaluation and show that LazyTainter is able to catch exactly the same leaks of private information as TaintDroid.

ROM	Data Object (MB)	Heap Alloc (MB)
Android	24.737	38.530
TaintDroid	33.134	47.348
LazyTainter	24.781	38.967

Table 5.1: Heap memory usage for the synthetic workload.

5.1 Memory Savings

As mentioned earlier, the size of Dalvik stack is generally very small, usually less than 256 KB. As a result, the memory overhead due to taint storage on the stack is also limited to 256 KB. Therefore, we focus on the memory overhead due to taint storage on the heap, which dominates the memory overhead of taint tracking in Dalvik.

As illustrated in Table 4.1, when objects are not tainted, the taint storage overhead of LazyTainter is either zero or 8-bytes and independent of the number of fields in an object. In contrast, the taint storage overhead of TaintDroid will increase with the number of fields. Therefore LazyTainter should be more memory-efficient than TaintDroid in a taint-free environment. To confirm this, we create a synthetic workload that allocates one million objects where each object contains two `int` fields. This workload uses small-sized objects which should minimize the memory saving benefits of LazyTainter over TaintDroid. We run this workload using vanilla Android, TaintDroid and LazyTainter ROMs and measure the memory usage using the Dalvik Debug Monitor Server (DDMS), which is part of Google's official Android SDK.

The results are summarized in Table 5.1. Data Object represents the amount of memory allocated to data objects on the Dalvik heap while Heap Alloc represents the total size of allocated heap memory. Both of these measures are tracked by the Dalvik heap allocator. We provide Data Object results because their theoretically expected value is given in Table 4.1. Since there are non-data objects on the heap, Heap Alloc is larger than Data Object. A full discussion of memory usage measurement methods used in this chapter is given in [7]. From these results we can see that TaintDroid incurs a

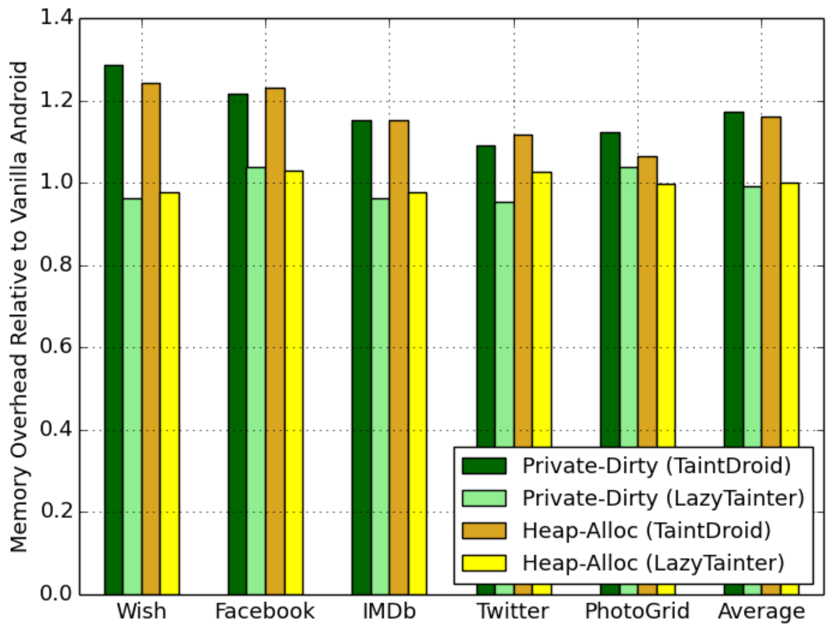


Figure 5.1: Memory usage on Galaxy Nexus.

34% memory overhead on data objects and a 23% overhead on allocated heap size, while the memory overhead of LazyTainter is almost negligible.

The expected result for an object with two fields is derived as follows. Since we put two `int` fields into each data object, the per-object memory overhead of TaintDroid is (32 - 24 =) 8 Bytes. Since we have created one million data objects in total, the overall memory overhead should be 8 MB. The Data Object results give an overall memory overhead of (33.134 - 24.737 =) 8.397 MB for TaintDroid. The 0.397 MB difference is due to additional data objects that must be allocated in an Android application, such as `Activity` objects and various UI elements. Since LazyTainter doesn't introduce any overhead when compared to vanilla Dalvik for data objects with two `int` fields, the memory overhead of LazyTainter versus vanilla Android is essentially zero except for the same additional objects. As most data objects in real applications will be larger than those with two `int` fields, memory savings in practice should be larger since the per-object memory overhead of TaintDroid grows with the object size while the per-object

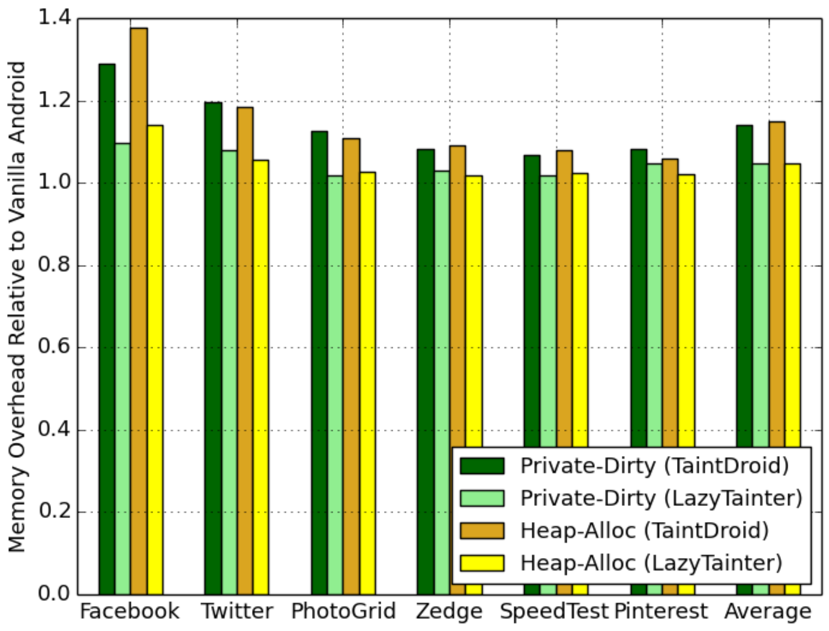


Figure 5.2: Memory usage on Nexus 7.

overhead of LazyTainter is constant for untainted objects.

We then proceed to evaluate the memory savings of LazyTainter on real applications. We created a corpus of several popular Android applications and randomly selected a subset to execute on both the Galaxy Nexus and the Nexus 7 devices. We run the applications on three ROMs: vanilla Android, TaintDroid and LazyTainter and measure the amount of memory used by each application. Since memory usage varies with application usage, we needed a methodology that would provide similar application usage across all three ROMs.

To do this, we use the `monkeyrunner` testing tool that is part of the Android SDK to write a script that will mechanically interact with each application for 2 minutes. `monkeyrunner` ensures that UI events will be delivered to applications in exactly the same order and at exactly the same time. Care was taken to ensure that the script caused a realistic amount of tainted, sensitive data to be read by the application. After the 2 minutes, the script triggers a garbage collection of the Dalvik heap to deallocate

Object \ Ratio \ Opcode	IGET	IPUT
Untainted	1.05	1.08
Tainted	1.03	1.04

Table 5.2: Performance ratio on synthetic benchmarks.

objects and then read the Dalvik memory usage data with the shell command `dumpsys meminfo`. To minimize noise due to variance in network delays, all tests were performed on a high-speed university network accessed using a low-latency wifi connection. For social applications such as Facebook and Twitter, we created fake accounts that would have minimal variation across requests. In addition, we execute all tests five times and use the average across the runs.

We use two different methods of measuring the memory usage of the applications. The first method, Private Dirty, measures the amount of dirty memory used by the application that is not shared with any other applications and represents additional memory overhead incurred solely by the application. This memory may include non-heap memory such as the stack, card table and auxiliary structures that are used by Dalvik. The second method, Heap Alloc, measures the amount of memory tracked by the heap allocator and is the same as the method used to measure memory usage for the synthetic workload. In both measures we only consider memory allocated by Dalvik because our optimization is mainly involved with the Dalvik heap. We also considered a third measure, Proportional Set Size (PSS), where shared memory pages are divided by the number of processes sharing them. PSS is a good measure for RAM usage comparison between different applications. However, Android may kill processes based on memory usage and the varying number of processes introduces noise into per-application measurements taken with PSS. Thus, measurements using PSS are not used in this evaluation.

The results of these measurements are presented in Figures 5.1 and 5.2. Each bar represents the memory overhead of TaintDroid or LazyTainter as normalized to that of vanilla Android. Measurements using both the Private Dirty method and the Heap Alloc method are given. From the results we see that the memory overhead of TaintDroid may vary among applications. If an application carefully manages its memory by recycling unused objects as soon as possible to maintain a small memory footprint (as in the case of Pinterest), then the memory overhead of TaintDroid is usually small as well (around 6%). On the contrary, if an application heavily uses the heap, then TaintDroid can give a large memory overhead (more than 20%). In terms of private dirty memory, the measured memory overhead of TaintDroid varies betwee 7-29%, while that of LazyTainter varies between 0-10%. In terms of heap usage, the measured memory overhead of TaintDroid varies between 6-38% while that of LazyTainter varies between 0-14%. LazyTainter always uses less memory than that of TaintDroid. In the best instance, LazyTainter reduced heap usage by as much as 26.5% when compared to TaintDroid. In some cases, LazyTainter even used slightly less (up to 4%) memory than vanilla Android. We attribute this to differences in memory layouts and variability between runs of the same application.

5.2 Performance Overhead

Since LazyTainter, by its design, leverages another level of indirection to reduce memory overhead, it inevitably incurs performance overhead due to additional instructions required to access lazily allocated taint storage. To measure this overhead, we first create a synthetic workload that intensively measures the execution time of operations where LazyTainter incurs the overhead. This happens when objects fields are being accessed.

In Dalvik, object fields are accessed with `IGET` and `IPUT` opcodes. We thus create an Android application that repeatedly accesses a field in an object 100 million times. We measure performance overhead for both reads (`IGET`) and writes (`IPUT`), as well as measure the overhead when the object is tainted or untainted (i.e. whether a shadow object is allocated for the object or not). We tabulate the ratio between the execution

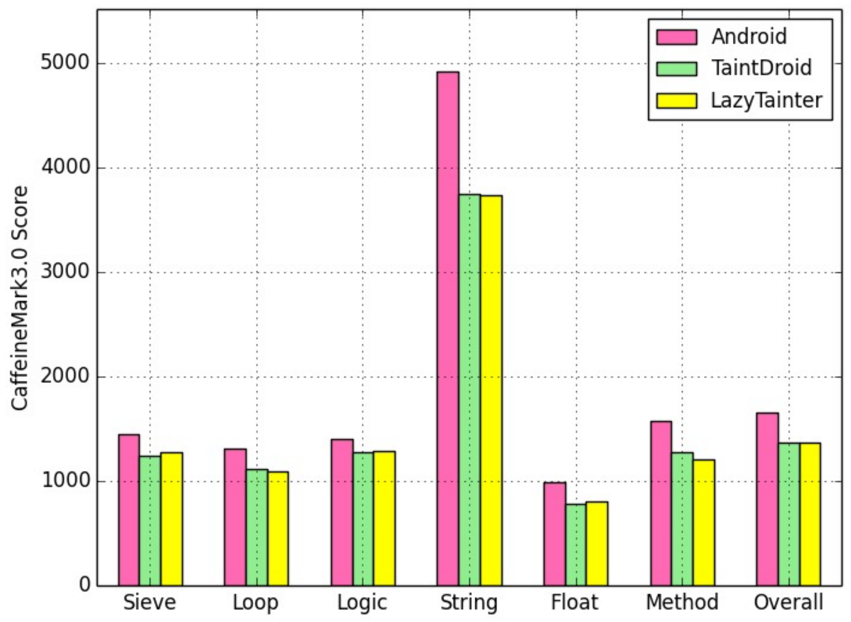


Figure 5.3: CaffeineMark3 result on Galaxy Nexus.

time of the workload on LazyTainter to the execution time of the workload on TaintDroid in Table 5.2.

From this result, we can see that the worst-case performance overhead of LazyTainter over TaintDroid varies between 3-8%. Surprisingly, access to an untainted object incurs more overhead than access to a tainted object. Both operations perform a test to see if a shadow object exists or not, but the access to the tainted object requires an additional memory access to access the actual taint value in the shadow object itself, so one would expect tainted objects to incur more overhead. To investigate further, we disassembled the compiled binary code for the `IGET` and `IPUT` instruction handlers. We found that the logic in both of these instruction handlers resulted in a number of conditional branch ARM instructions to check the value of the shadow pointer. From this, we speculate that the untainted path might have resulted in slightly more branch mispredictions, thus resulting in more overhead than the tainted path. The confirmation of this is left for future work.

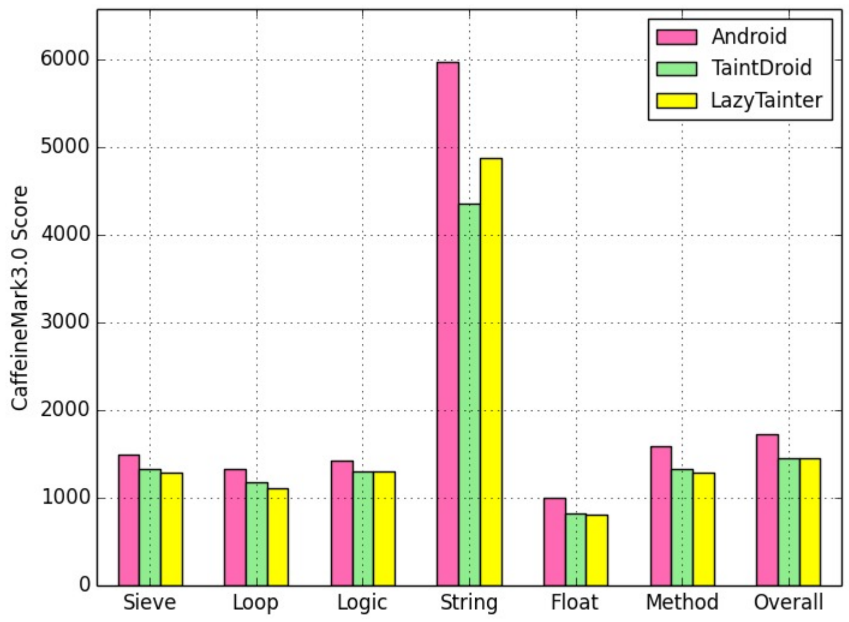


Figure 5.4: CaffeineMark3 result on Nexus 7.

Since all computation in Dalvik happens on the stack and meaningful applications don't waste time on useless heap accesses, to get a better idea of the performance overhead of LazyTainter in practice, we use CaffeineMark3 [5], a popular benchmark tool measuring the speed of Java programs. We run each workload in CaffeineMark3 five times and take the average. We do not taint any memory since the results of the synthetic workload indicate that access to untainted objects incur more runtime overhead than tainted objects. The results are presented in Figures 5.3 and 5.4. The Y-axis represents the CaffeineMark3 score, where higher bars represent faster execution. On the Galaxy Nexus we have overall scores 1656, 1370 and 1366 for Android, TaintDroid and LazyTainter, respectively. On the Nexus 7 the overall scores are 1726, 1457 and 1455. This shows that in practice, the runtime overhead between TaintDroid and LazyTainter is generally within 1% on both devices.

App	Leak Type
PhotoGrid	None
The Weather Channel	Location
Twitter	Location
Facebook	Location
IMDb	Location
Solitaire	Location, IMEI
Horoscope	Location, IMEI
Voice Search	Audio
Wish	Address Book

Table 5.3: Reported privacy leaks.

App	The Weather Channel
`GET /wxdata/loc/get.js?lat=[latitude]&lng =[longitude]&locale=en_US&...`	
App	Solitaire
`GET /post/config?p=android&a=...&m=2.3.2& v=1.3.2&d=[IMEI]&`	
App	Horoscope
`POST /ws_pub/gcm.php?action=register&hwui d=[IMEI]&dt=...`	

Table 5.4: Confirmed privacy leaks.

5.3 Taint Propagation Correctness

Recall that to ensure the validity of our memory and performance overhead measurements, we designed LazyTainter to have the same taint propagation function as TaintDroid. Thus, LazyTainter should detect the same leakage of private information as TaintDroid. To confirm this, we selected a set of applications which use different types of private information and ran them on both TaintDroid and LazyTainter. Then we collected the privacy leaks reported by these two systems. The result is shown in Table 5.3.

Both TaintDroid and LazyTainter reported the exact same privacy leaks. We also manually confirmed some of them by looking at the text sent through the network, as summarized in Table 5.4. This confirms accuracy and effectiveness of both TaintDroid and LazyTainter.

Chapter 6

Discussion

6.1 Reference Tainting

Since the effectiveness of lazy tainting depends on proportion of objects that get tainted, the precision of the taint tracking system is highly important. In particular, incorrect taint propagation can create a larger number of tainted objects, thus resulting in more taint storage overhead for LazyTainter. During the implementation of LazyTainter, we discovered a subtle set of false taint propagation instances for string objects in TaintDroid. TaintDroid taints a string object by tainting its array field. However, Dalvik relies on string interning to minimize memory footprint of string objects. Therefore multiple string references may actually point to the same underlying string object. If one of them gets tainted, all other references will also get tainted implicitly. A similar issue has been observed in the JavaScript interpreter [8]. To address this problem we can further classify opaque objects into *immutable* and *mutable* objects. For immutable objects, we taint the reference to the object rather than the object itself (which effectively treats the object as a primitive value). For mutable objects, there is no reason in caching or interning them since otherwise accesses via different references may interfere with each other. We have implemented a prototype of reference tainting for string objects which eliminates this type of incorrect taint propagation. However, since it changes the taint propagation logic, we did not use this version in our evaluation.

6.2 Implementation Choices

LazyTainter is currently implemented on the portable interpreter. The main reason for this decision is that the portable interpreter requires less memory than the JIT, and is more likely to be enabled on resource constrained phones than the JIT. In fact, Google's Android documentation recommends disabling the JIT entirely for low-memory devices [18]. A secondary reason is that the execution and performance of the portable interpreter is more predictable than the JIT. For example, the overhead of executing various instructions like `IGET` and `IPUT` where LazyTainter adds overhead is fairly constant as the instructions are translated the same way each time.

Both the JIT and the portable interpreter use the same memory layout so we expect the memory savings provided by the portable interpreter implementation to carry over if implemented in the JIT. LazyTainter's additional performance overhead over TaintDroid comes mainly from the additional logic that LazyTainter must implement for `IGET`s and `IPUT`s. These instructions do not make up a larger percentage of total instructions executed, so we expect the overhead in a JIT implementation to be similar to that of the interpreter implementation.

Finally, the measurements of memory savings represent a worse case for LazyTainter because of TaintDroid's decision to track taints for array objects as a whole. The savings for these objects is nominal in both TaintDroid and LazyTainter as the taint storage overhead is only one taint tag. If more precise per-element taint tracking is required, LazyTainter's memory savings should be even larger. We have implemented a prototype of per-element array tainting and confirmed greater memory savings. However, since it is no longer functionally equivalent to TaintDroid, it was not used in our evaluation.

Chapter 7

Related Work

Dynamic taint tracking has enjoyed a long history of use for proposals in information tracking, malware detection and attack detection. A good literature survey of dynamic taint tracking for managed runtimes can be found in [16]. Taint-tracking systems predominantly allocate taint storage eagerly and track taints at a fixed granularity. For example, Newsome et al. [19] enhances Valgrind with a shadow memory to store taint values in a one-to-one correspondence with values in program memory. Yin et al. [29] assume a similar shadow memory is implemented in hardware. Zhu et al. [30] also use a statically allocated table as taint storage, but achieve better performance by using function summaries. Others allocate memory on-demand, but still do so at fixed granularity. For example, Xu et al. [28] intercept segmentation fault signals and allocate a 16KB memory chunk spanning the faulting address if it's within the expected range. LazyTainter will be more memory-efficient than these solutions because it allocates taint storage with an adjustable granularity and uses garbage collection to automatically deallocate unused taint storage.

There has also been some work on tainting language runtimes for Javascript in browsers [26, 8]. Nguyen-Tuong et al. [20] take a similar strategy for PHP on a web server and provide taint tracking at the precision of a character granularity. In all these cases, taint storage is allocated dynamically as variables are instantiated and used, but the granularity of taint-tracking does not adapt to the taint propagation behavior of the program. Thus, these solutions cannot save taint storage for applications that do not propagate taints to a large number of memory locations.

There have been a few instances where researchers have explored adaptive tainting systems. Suh et al. [23] implement taint tracking support in a processor and use only a single taint tag in the hardware page table for pages without a valid physical mapping, thus allowing them to avoid allocating an entire page of taint storage for such pages. Ho et al. [11] dynamically switch between fine byte-level taint tracking in QEMU and coarse page-level taint tracking using the Xen hypervisor. The mechanisms in these previous works differ significantly from the dynamic granularity taint tracking mechanism of LazyTainter.

The closest application of taint tracking to LazyTainter is TaintDroid [9], which implements taint tracking in Android's Dalvik virtual machine. A number of projects have used TaintDroid's information to build other useful functionality on top of Android. For example, Balebako et al. [2] use phones with TaintDroid installed on them to perform a user study to gap between user's perceptions and the reality of privacy leakage on smartphones. AppFence [12] uses TaintDroid with data shadowing to prevent exfiltration of sensitive data without breaking the functionality of applications. CleanOS [24] uses TaintDroid to track sensitive data as it is propagated throughout the smartphone and encrypt it to protect it from being leaked if the phone is stolen. Since LazyTainter is functionally identical to TaintDroid, we believe that these and any other projects that use TaintDroid would work well with LazyTainter.

Chapter 8

Conclusions

Dynamic taint tracking is an effective approach for detecting privacy leakage on mobile devices. However, designing a good taint tracking system requires finding the proper trade-off among precision, speed and memory overhead. In LazyTainter, we are able to reduce the memory overhead of TaintDroid with no reduction in precision and minimal reduction in speed. To do this, we make several important observations. First, the majority of memory used by mobile applications does not contain any private information, motivating a system that uses coarse-grained taint tracking by default and lazily switches to fine-grained taint tracking when a field in an object becomes tainted. Second, the VM heap and stack have very different characteristics in terms of size, access pattern and types of data stored, motivating a hybrid approach that uses eager taint tracking on the stack and lazy taint tracking on the heap. While LazyTainter is implemented for Android, we believe the technique of lazy tainting can be applied to any runtime system that executes object-oriented code, and whose applications exhibit localized tainting behavior.

Bibliography

[1] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[2] Rebecca Balebako, Jaeyeon Jung, Wei Lu, Lorrie Faith Cranor, and Carolyn Nguyen. "Little brothers watching you": Raising awareness of data leaks on smartphones. In Lujo Bauer, Konstantin Beznosov, and Lorrie Faith Cranor, editors, *SOUPS*, page 12. ACM, 2013.

[3] Dan Bornstein. Dalvik VM Internals, 2008.

[4] Dave Burke. Android 4.4 KitKat and Updated Developer Tools. `http://android-developers.blogspot.ca/2013/10/android-44-kitkat-and-updated-developer.html`, 2013.

[5] CaffeineMark 3.0. `http://www.benchmarkhq.ru/cm30/`.

[6] Mads Dam, Gurvan Le Guernic, and Andreas Lundblad. TreeDroid: A Tree Automaton Based Approach to Enforcing Data Processing Policies. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 894–905, New York, NY, USA, 2012. ACM.

[7] Investigating Your RAM Usage. `https://developer.android.com/tools/debugging/debugging-memory.html`.

[8] Vladan Djeric and Ashvin Goel. Securing Script-based Extensibility in Web Browsers. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 23–23, Berkeley, CA, USA, 2010. USENIX Association.

[9] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[10] Flurry Blog. `http://blog.flurry.com/?Tag=UsageStatistics`.

[11] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical Taint-based Protection Using Demand Emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 29–41, New York, NY, USA, 2006. ACM.

[12] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These Aren'T the Droids You'Re Looking for: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 639–652, New York, NY, USA, 2011. ACM.

[13] IDC Press Release. `http://www.idc.com/getdoc.jsp?containerId=prUS24676414`.

[14] Android Kitkat. `http://developer.android.com/about/versions/kitkat.html`.

[15] Doug Lea. A Memory Allocator. `http://g.oswego.edu/dl/html/malloc.html`, 2000.

[16] Benjamin Livshits. Dynamic Taint Tracking in Managed Runtimes. Technical Report MSR-TR-2012-114, Microsoft Research, Nov 2012.

[17] Benjamin Livshits and Jaeyeon Jung. Automatic Mediation of Privacy-sensitive Resource Access in Smartphone Applications. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 113–130, Berkeley, CA, USA, 2013. USENIX Association.

[18] Running Android with low RAM. `http://source.android.com/devices/low-ram.html`.

[19] James Newsome. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[20] Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically Hardening Web Applications Using Precise Tainting. In *In 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[21] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 543–558, Berkeley, CA, USA, 2013. USENIX Association.

[22] Pakistan Security Engineering Research Group, Institute of Management Sciences Peshawar. Analysis of Dalvik Virtual Machine and Class Path Library, 2009.

[23] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 85–96, New York, NY, USA, 2004. ACM.

[24] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 77–91, Berkeley, CA, USA, 2012. USENIX Association.

[25] To target next billions in emerging markets, phone makers balance cost, performance. `http://www.usnews.com/news/business/articles/2014/02/25/phone-makers-look-to-emerging-markets-for-growth`.

[26] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*. The Internet Society, 2007.

[27] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. Jekyll on iOS: When Benign Apps Become Evil. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 559–572, Berkeley, CA, USA, 2013. USENIX Association.

[28] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[29] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and

Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

[30] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. TaintEraser: Protecting Sensitive Data Leaks Using Application-level Taint Tracking. *SIGOPS Oper. Syst. Rev.*, 45(1):142–154, February 2011.