LazyTainter: Memory-Efficient Taint Tracking in Managed Runtimes

Zheng Wei
University of Toronto
zheng.wei@mail.utoronto.ca

David Lie
University of Toronto
lie@eecg.toronto.edu

ABSTRACT

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 makes up the majority mobile device growth in emerging markets. To make taint tracking affordable and 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 with *hybrid taint tracking*, which combines lazy and eager tainting, to reduce memory usage with only negligible performance loss. Our experimental results demonstrate that LazyTainter can reduce heap usage by as much as 26.5% when compared to TaintDroid while imposing a negligible 1% increase in performance overhead.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Information flow controls

General Terms

Security, Efficiency

Keywords

Android, Taint Tracking, Memory Efficiency

1. INTRODUCTION

Mobile devices are quickly replacing traditional personal computers 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 [1, 17, 21] and dynamic [6, 9, 12] approaches have been proposed. While static analysis imposes no run-time overhead, it is inherently imprecise [26]. In contrast, dynamic taint-tracking, which uses *taint tags* to track whether a value contains private information has been shown to be effective at detecting personal privacy violations. While native binary instrumentation can incur considerable performance overhead [19, 28], the overhead of taint tracking can be reasonable for a managed run-time. For example, TaintDroid [9] is able to track taints in real-time with a mere 14% performance overhead. As a result, TaintDroid gains great popularity among mobile device users and its taint tracking functionality has been widely 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 memory on high-end devices has grown steadily, the bulk of the smartphone market remains in low-end devices [13]. According to Google, millions of entry-level devices around the world still have as little as 512MB RAM [14], leading Google to launch Project Svelte [4] to address this issue specifically. When one takes into account that some amount of this limited memory pool must be reserved for the operating system and system services, this leaves even less for each application. As mobile applications mature, they are likely to include more features and thus require more resources, making memory a potentially limiting resource on entry-level devices. While one can increase the speed of a device by overclocking the CPU, there is no similar way to increase available RAM.

As a result, it is important to examine how the memory overhead of security mechanisms like taint-tracking can be reduced on mobile devices. To address this, we propose a mechanism that reduces the memory overhead of taint-tracking while maintaining reasonable performance overhead. Specifically, we notice that the storage for taint information can be allocated either *eagerly* (ahead of time), or *lazily* (on-

SPSM'14, November 07 2014, Scottsdale, AZ, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3155-5/14/11 ...$15.00.
http://dx.doi.org/10.1145/2666620.2666626.

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 be 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 use a hybrid approach of taint tracking which is able to maximize the benefits of both eager and lazy tainting and minimize the costs. To demonstrate our results, we present *LazyTainter*, which uses eager-tainting for frequently accessed memory locations, lazy-tainting for infrequently accessed locations, and carefully optimizes the storage of taint information. To ensure a fair evaluation of eager-tainting, we design LazyTainter to be functionally equivalent to TaintDroid. However, LazyTainter uses up to 26.5% less heap memory and imposes a less than 1% increase in performance overhead when compared to TaintDroid. This shows that the memory saving benefits come at no cost to taint-tracking precision. Finishing the same task with less memory is almost always an advantage. The system may enjoy a larger cache size to do more effective caching. User applications may keep more data in memory to be more responsive. Depending on the usage scenario, mobile device users can benefit from the memory savings in different ways.

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

2. BACKGROUND

2.1 Android

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

Dalvik executes its own DEX bytecode format [22]. Android applications, which are written in Java, are 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.

Dalvik is a 32-bit register-based virtual machine designed for resource-constrained devices. Registers are allocated on the stack frame by each method. All computation occurs on the stack. Dalvik also has a 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.

Dalvik has its own instruction set. The opcode size is 8-bit so there will be at most 256 opcodes from `00` to `FF`. Dalvik opcodes have clear semantics and can be roughly classified into several groups. For example, `ADD`/`SUB` opcodes do arithmetic computation on the stack, `NEW` opcodes create objects on the heap and `IGET`/`IPUT` opcodes move data between the stack and the heap.

The Dalvik heap is a virtual memory range acquired with `mmap` and managed by 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 termed *Heap Alloc*. When a garbage collection is triggered, Dalvik will walk the heap, free unused space and return them to the memory allocator. The amount of free space on the heap is termed *Heap Free*. The total size of the heap is termed *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 space being used at the end of the heap. Therefore Heap Alloc is a more precise measurement of heap usage than Heap Size because it considers fragmentation.

Although Android is Linux-based, its process creation is different from that of traditional Linux. In Android, there is a nascent Zygote process which performs initialization that is common to all applications, and application processes are created by forking from Zygote. Therefore, 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]. 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. Android doesn't use a swap partition. If an application's memory cost is too high, the system may kill processes to reclaim memory.

2.2 TaintDroid

TaintDroid [9] is a taint 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 network. 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*), so up to 32 types of sensitive information can be tracked in parallel. TaintDroid stores the taint tag for a variable adjacent to the variable itself. For 32-bit register values (Figure 1), taint tags are interleaved between values so that register `fp[i]` becomes `fp[2*i]` in TaintDroid's stack layout. For 64-bit register values, which are formed by adjacent register pairs, 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 heap are also interleaved so that a 32-bit field with offset `k` now has off-

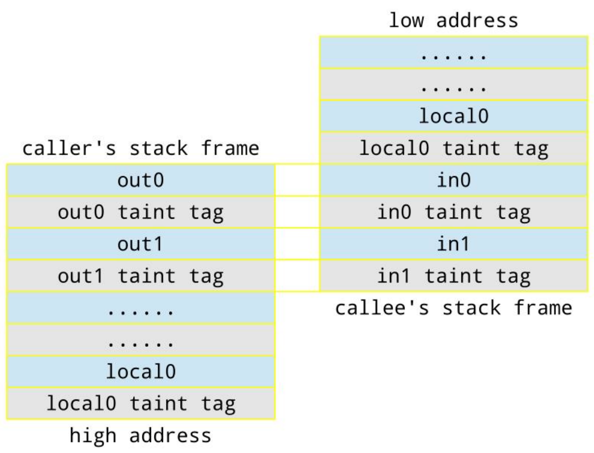

Figure 1: TaintDroid stack layout.

set `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, 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 one taint tag that is shared by all elements of the array. Therefore, tainting for arrays only has minimal memory overhead. 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 exactly the same mechanisms as TaintDroid to track taints through these channels and thus we omit a detailed description here. Finally, since TaintDroid is designed to work in a virtualized runtime environment, third-party native libraries are not supported. However, TaintDroid does support native system libraries in the firmware because they are in the trusted computing base.

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 one of these properties usually comes at a cost of degrading the others:

- 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.
- We can use 1-bit taint tags instead of 32-bit taint tags. The reduction in taint tag size may lead to a reduction in 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 between different taint sources, such as location, IMEI, etc.
- 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 improve one property with only minimal negative impact on the other two. Thus, a taint tracking system might 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. Taint tracking systems such as TaintDroid have already made such trade-offs. For example, one of the trade-offs that TaintDroid makes is reducing the precision of taint tracking for arrays and strings to save memory and increase speed. Another trade-off is eagerly allocating taint tag storage to trade memory for 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 at all. To achieve this goal, we introduce *lazy tainting* and apply it to TaintDroid. We use TaintDroid as the starting point because we believe it is the most popular and influential taint tracking system for mobile devices, as proved by the many projects [2, 12, 24] that have used or incorporated TaintDroid. Also, TaintDroid is well implemented and documented. Its high-quality code has greatly facilitated our work.

Finally, we note that while we demonstrate our ideas on TaintDroid, we believe our ideas are general enough to be readily adapted and 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 the mobile applications we use, we further hypothesize that Android applications 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 with only a tiny portion of tainted 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 1: Tainted object ratio (rounded to 0.01). `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.

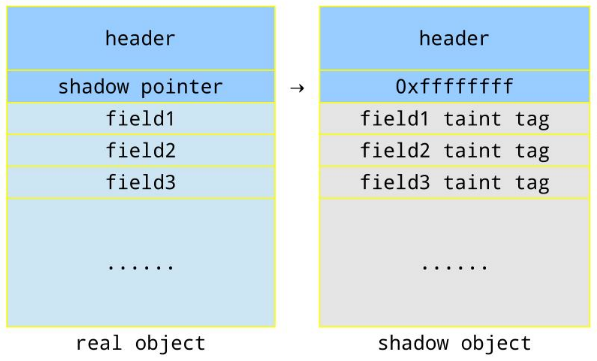

Figure 2: A tainted real object in LazyTainter. Shadow objects are used to store taint tags if the real object has any tainted fields.

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 all live objects regardless of whether they are tainted or not. We count objects only 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 and do not include array objects. The testing firmware is the same as TaintDroid except for the profiling patch. All taint sources in TaintDroid are available during the profiling so that applications are free to access any taint sources they want.

We tested nine applications in total, as shown in Table 1. 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. From the results, we can see that only a tiny portion of objects contain 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. Thus 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 tainting 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, 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 a pointer, as illustrated in Figure 2. 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 region for handling stack overflow errors). 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 hybrid tainting.

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 section 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 tainting for the stack and lazy tainting for the heap. Since the stack is heavily used for computation and the heap for storage, we can minimize performance overhead in the stack and minimize memory overhead in the heap at the same time by forming a *memory hierarchy*. However, 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 [27, 28, 29], 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 [27] or prohibit computation-heavy instrumentation when taints haven't been introduced into the system [29]. 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 2. 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 access 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 allocated 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.

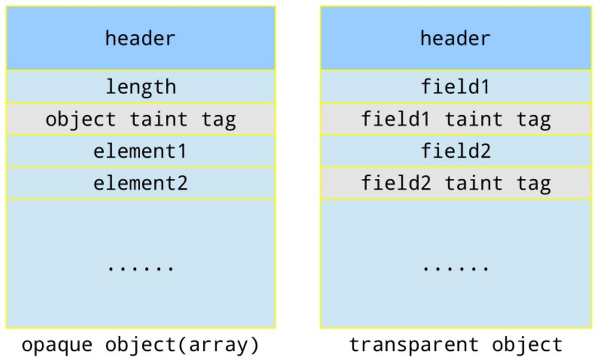

Figure 3: TaintDroid object layout.

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 a 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 during garbage collection.

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, since all fields are taint tags, the garbage collector should just mark the object and return to the parent object. To let the garbage collector differentiate between real and shadow objects, we set the space reserved for the shadow pointer in a 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 arises because LazyTainter uses dynamic taint allocation in 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.

4. IMPLEMENTATION

We implemented LazyTainter on Android `4.1.1_r6`, which is officially supported by TaintDroid. Since we want to have

```
1  struct Object {
2      ClassObject*     clazz;
3      u4               lock;
4      //  Added in LazyTainter.
5      Taint            taint;
6  };
7
8  struct ClassObject : Object {
9      u4 instanceData[CLASS_FIELD_SLOTS];
10     const char*      descriptor;
11     char*            descriptorAlloc;
12     ...
13 };
14
15 struct ArrayObject : Object {
16     u4               length;
17     u8               contents[1];
18 };
19
20 struct DataObject : Object {
21     u4               instanceData[1];
22 };
```

Listing 1: Object implementation in Dalvik.

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. Third-party native libraries are not supported. Native system libraries are supported but native methods are tracked at method level. In addition, only explicit data flow is tracked to avoid taint explosion. LazyTainter is implemented on the portable interpreter and we describe the reason for this decision in Section 6.

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 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 one taint tag for each field, while an array object maintains 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 3 gives the layout of opaque and transparent objects in TaintDroid.

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 2: Object size with varying numbers of fields.

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 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 2.

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

Listing 2: Field reuse with `union Taint`.

Then we add a field of type `Taint` in `Object` as shown in Line 5 of Listing 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 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 2 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 so that all kinds of objects can still be correctly recycled in face of the heap diversity. Specifically, we want a shadow object to be 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 garbage collector (GC). We must modify the GC so that the following properties are satisfied:

1. *No dangling pointers.* A shadow object must be alive if the real object is alive.
2. *No memory leaks.* 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 to deal 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 access 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 one. We apply the same logic in other places where 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 conversion between an `int` and an `Integer`. We fixed these issues by manually shifting the offset by 4 bytes before accessing the field.

5. EVALUATION

We evaluated LazyTainter on Galaxy Nexus (maguro). We chose this device because it's sufficiently popular, relatively low-end and officially supported by TaintDroid. Galaxy Nexus has a dual-core 1.2GHz ARM Cortex-A9 processor with 1GB RAM and runs well on Android `4.1.1_r6`, the version on which TaintDroid and LazyTainter are implemented.

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.

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 taint storage overhead on the stack is also limited to 256 KB. Therefore, we focus on the taint storage overhead on the heap, which dominates the memory overhead of taint tracking in Dalvik.

As illustrated in Table 2, 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 (i.e. $\Theta(1)$). In contrast, the taint storage overhead of TaintDroid will linearly increase with the number of fields (i.e. $\Theta(n)$). 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 3. *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 2. Since there are non-data objects on the heap, Heap Alloc is larger than Data Object. A full discussion of memory measurement methods used in this section is given in [7]. From these results we can see that TaintDroid incurs a 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 `int` 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 can become larger because the per-object memory overhead of TaintDroid grows with the object size while the per-object overhead of LazyTainter is constant for untainted objects.

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

Table 3: Heap memory usage (synthetic workload).

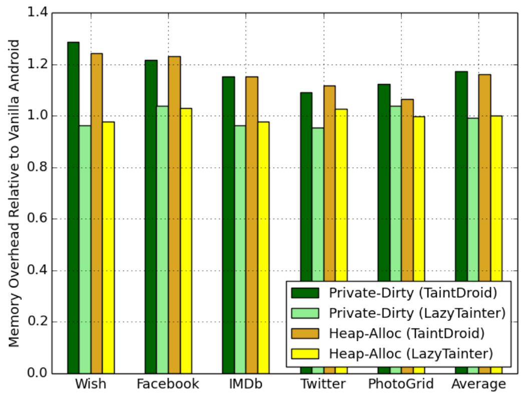

Figure 4: Memory overhead of TaintDroid and LazyTainter (relative to vanilla Android).

We then proceed to evaluate the memory savings of LazyTainter on real applications. We created a corpus of Android applications based on popularity and non-trivial use of private information. Then we ran the applications on three ROMs: vanilla Android, TaintDroid and LazyTainter and measured the amount of memory used by each application. Since memory usage varies with application usage, we need a methodology that can provide similar application usage across all three ROMs.

To do this, we use the `monkeyrunner` testing tool, which is part of the Android SDK, to write a script that mechanically interacts with each application for two minutes. The use of `monkeyrunner` ensures that UI events will be delivered to applications consistently and uniformly. Care was taken to ensure that the script caused a realistic amount of tainted, sensitive data to be read by the application. After the two minutes, the script triggers a garbage collection of the Dalvik heap to deallocate unused objects and then reads 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 and low-latency university network. 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 concurrent applications. However, if applications are measured sequentially, 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 Figure 4. In this figure, each bar represents the memory overhead of TaintDroid or LazyTainter relative to 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 maintains a small memory footprint by using small objects and recycling unused objects as soon as possible, then the memory overhead of TaintDroid is usually small as well (less than 10%). On the contrary, if an application heavily uses the heap, then TaintDroid can give a larger memory overhead (more than 20%). In terms of private dirty memory, the measured memory overhead of TaintDroid varies between 9-29%, while that of LazyTainter varies between 0-4%. In terms of heap usage, the measured memory overhead of TaintDroid varies between 6-24% while that of LazyTainter varies between 0-3%. And LazyTainter always uses less memory than TaintDroid. In the best instance, LazyTainter reduced heap usage by as much as 26.5% when compared to TaintDroid. In fact, the memory overhead of LazyTainter fluctuates around that of vanilla Android within a range given by repeated runs of vanilla Android itself. This conforms to the theoretical results calculated in Table 2.

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 make 100 million accesses to fields in an array of 1000 objects. We measure performance overhead for both reads (`IGET`) and writes (`IPUT`), as well as measure the overhead when the objects are clean or tainted (i.e. whether shadow objects are allocated for the objects or not). We note that the objects in the array are all initially untainted, so the run times measured include the cost of allocating the shadow objects the first time the real objects are tainted.

We performed five measurements of each benchmark on each platform and tabulate the ratio and standard deviation of the ratio of the application execution time on LazyTainter to the application execution time on TaintDroid in Table 4. From this result, we can see that the performance overhead of LazyTainter over TaintDroid on clean objects is negligible while `IGET`s and `IPUT`s both have about 6% overhead on tainted objects.

Object \ Ratio \ Op	IGET	IPUT
Clean	1.000±0.006	0.991±0.006
Tainted	1.055±0.033	1.058±0.032

Table 4: Execution time ratio of LazyTainter to TaintDroid (rounded to 0.001).

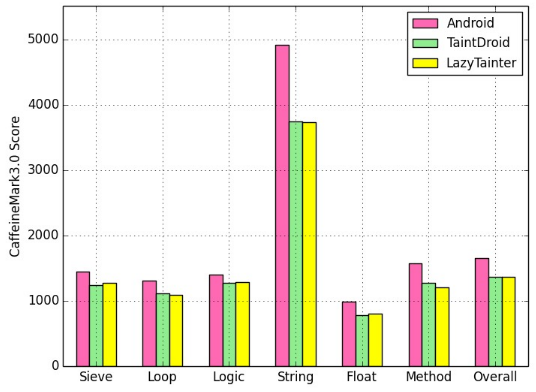

Figure 5: CaffeineMark3 benchmark result.

In practice, the majority of objects in an application are clean and `IPUT` and `IGET` instructions take a smaller portion in all executed instructions, so we expect the actual overhead to be lower in real workloads. To get a better idea, we use CaffeineMark3 [5], a popular benchmark tool measuring the speed of Java programs, for our evaluation. We run each workload in CaffeineMark3 five times and take the average. We enable all standard taints so that data objects acquire taints when they access sensitive Android APIs. The results are presented in Figure 5. The X-axis represents the test performed by CaffeineMark3. The Y-axis represents the CaffeineMark3 score, where higher bars represent faster execution. The overall scores are 1656, 1370 and 1366 for vanilla Android, TaintDroid and LazyTainter, respectively. This shows that in practice, the runtime overhead between TaintDroid and LazyTainter is generally within 1%.

5.3 Taint Propagation Logic

Recall that to ensure the validity of our memory and performance overhead measurements, we designed LazyTainter to have the same taint propagation logic 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.

Both TaintDroid and LazyTainter reported exactly the same privacy leaks. We also manually confirmed some of them by looking at the text sent through the network, as summarized in Table 6. This confirms the effectiveness of both TaintDroid and LazyTainter.

6. DISCUSSION

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 devices than the JIT. In fact, Google's official 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 vari-

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: 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 6: Confirmed privacy leaks.

ous instructions like IGET and IPUT where LazyTainter adds overhead is fairly constant as the instructions are translated the same way each time.

Both the portable interpreter and the JIT 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 IGETs and IPUTs. These instructions only make up a small percentage of total instructions executed, so we expect the overhead in a JIT implementation to be similar to that of the portable interpreter implementation.

Finally, the current measurements of memory savings do not fully represent the capability of LazyTainter because of TaintDroid's decision to track taints for array objects as a whole. The savings for these objects are nominal in both TaintDroid and LazyTainter as the taint storage overhead is only one taint tag, so there isn't much opportunity to decrease the taint tracking cost of arrays. However, users who want fewer false positives and more precision may decide to turn on per-element taint tracking for arrays, at which time lazy tainting will be able to provide even greater memory savings.

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. [28] assume a similar shadow memory is implemented in hardware. Zhu et al. [29] 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. [27] intercept segmentation fault signals and allocate a 16KB memory chunk spanning the faulting address if it's within the expected range. When compared with these projects, LazyTainter gives a unique solution to manage taint storage 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 [8, 25]. 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.

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. 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 other projects that use TaintDroid would also work well with LazyTainter, and we are pleased to enrich the mobile taint tracking toolset to help a broader range of users secure their devices.

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 memory overhead observed in eager-tainting systems 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.

Acknowledgments

We would like to thank Zhen Huang, Michelle Wong, Ashvin Goel and Michael Stumm for their insightful conversations and suggestions. We thank William Enck and Jaeyeon Jung for the development of TaintDroid and detailed explanations of its internals. We also thank the anonymous reviewers for their constructive comments. This research was supported by an Ontario Research Fund – Research Excellence Grant and an NSERC Discovery Grant.

References

[1] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. 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] R. Balebako, J. Jung, W. Lu, L. F. Cranor, and C. Nguyen. "Little brothers watching you": Raising awareness of data leaks on smartphones. In L. Bauer, K. Beznosov, and L. F. Cranor, editors, *SOUPS*, page 12. ACM, 2013.

[3] D. Bornstein. Dalvik VM Internals. `https://sites.google.com/site/io/dalvik-vm-internals`, 2008.

[4] D. 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] M. Dam, G. Le Guernic, and A. 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] V. Djeric and A. 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] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *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] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. 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] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't the Droids You're Looking for: Retrofitting Android to Protect Data from Imperious Applications. In *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] D. Lea. A Memory Allocator. `http://g.oswego.edu/dl/html/malloc.html`, 2000.

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

[17] B. Livshits and J. 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] J. 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] A. Nguyen-tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *In 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[21] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. 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] P. Security Engineering Research Group, Institute of Management Sciences Peshawar. Analysis of Dalvik Virtual Machine and Class Path Library, 2009.

[23] G. E. Suh, J. W. Lee, D. Zhang, and S. 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] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. 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] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*. The Internet Society, 2007.

[26] T. Wang, K. Lu, L. Lu, S. Chung, and W. 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.

[27] W. Xu, S. 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.

[28] H. Yin, D. Song, M. Egele, C. Kruegel, and E. 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.

[29] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: Protecting Sensitive Data Leaks Using Application-level Taint Tracking. *SIGOPS Oper. Syst. Rev.*, 45(1):142–154, Feb. 2011.