PIFT: Predictive Information-Flow Tracking

Man-Ki Yoon
University of Illinois at Urbana-Champaign
mkyoon@illinois.edu

Negin Salajegheh Yin Chen
Mihai Christodorescu
Qualcomm Research
{msalajeg,yinc,mihai}@qti.qualcomm.com

Abstract

Phones today carry sensitive information and have a great number of ways to communicate that data. As a result, malware that steal money, information, or simply disable functionality have hit the app stores. Current security solutions for preventing undesirable data leaks are mostly high-overhead and have not been practical enough for smartphones. In this paper, we show that simply monitoring just some instructions (only memory loads and stores) it is possible to achieve low overhead, highly accurate information flow tracking. Our method achieves 98% accuracy (0% false positive and 2% false negative) over DroidBench and was able to successfully catch seven real-world malware instances that steal phone number, location, and device ID using SMS messages and HTTP connections.

Categories and Subject Descriptors K.6.5 [*Management of Computing and Information Systems*]: Security and Protection

Keywords Security; information flow tracking

1. Introduction

Information-flow control (IFC) is a fundamental technique in computer security and privacy, allowing fine-grained control over information as it is routed between running programs and processed by them. Information-flow tracking [9, 14, 16] is the underlying mechanism that enables IFC to make decisions. With a robust tracking system, IFC can be used to prevent privacy leaks, ensure the integrity of information by verifying its provenance, and minimize the attack surface of a system by restricting the data accessible to any one program in that system. An information-flow tracking system that supports these goals has to be accurate, scalable, and efficient.

ASPLOS '16 April 2–6, 2016, Atlanta, Georgia, USA.

DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2872362.2872403

Today there is a large number of information-flow tracking techniques, from static [5] to dynamic [9] and from hardware-based [16] to hybrid [19] to software-only [14]. Static analysis of program code is often constrained to particular languages and particular runtime environments, thus making it inadequate in a real system consisting of programs written in diverse languages. Dynamic information-flow tracking modifies the program, the runtime environment, or the hardware to associate supporting information (called taint state or taint flags) to relevant program data and to propagate this supporting information as the program copies, manipulates, and transforms the data. The dynamic approach is broadly applicable to many programming languages and runtimes, making it attractive in practice, but comes with the implicit overhead of doing additional work per program instruction. Rewriting the program or the runtime to add dynamic information-flow tracking introduces large overheads, which translate to increased power consumption, and is thus impractical for all but strictest security- and privacy-conscious uses. Hardware solutions usually introduce less than 1% overhead, but require re-architecting the system to supplement each storage element (e.g., registers, cache blocks at all levels, main memory) with the storage for the associate taint state. Such re-architecting is undesirable from an engineering perspective.

We approach the challenges of information-flow tracking from the position of simplifying the mechanics of tracking while carefully trading off perfect accuracy for sufficient accuracy. In other words, the question we try to answer is how much performance and simplicity of implementation we can gain by reducing tracking accuracy without endangering the task at hand (qualitative information-flow control). We make the observation that *information flows have distinguishable characteristics in terms of their structure* (e.g., length, duration, instruction distribution). This fact (which we support with empirical evidence and analytical reasoning later in this paper) allows us to discard the requirement that every instruction be analyzed (statically or dynamically). Furthermore, this allows us to design the tracking mechanism in terms of inexpensive syntactic elements of the program, instead of expensive semantic ones.

We discuss the use of statistics over the instruction stream, in particular with respect to *memory loads* and *mem-*

ory stores, to determine whether a data value from a source reaches a sink. While in general it might be possible to predict whether a memory operation that reaches a sink (e.g., a memory store to a parameter for a sink function call) operates on data read much earlier from a source, the accuracy of the resulting information-flow would be low if all intermediate execution between source and sink are ignored. We take a more refined view, where an information flow from a source to a sink is divided into segments consisting of a memory-load operation (where sensitive data is loaded into the CPU), a number of manipulations of the data in CPU registers, and one or more memory-store operations (where the derived data, still sensitive, is sent from CPU to memory). The structural characteristics of information flows can predict which memory stores in an instruction stream are related via a CPU-only flow to which memory loads. In other words, we treat the CPU as a black box, where sensitive data goes in via *loads*, and then we can tell which of the outgoing data is also sensitive based on the characteristics of the *stores*. Repeating this prediction process creates a chain of load–store operations that deal with sensitive data, eventually establishing whether an information flow from a source to a sink exists.

The benefits of this approach is to reduce the amount of data to be processed (load–store operations are at least an order of magnitude less frequent than arbitrary CPU operations) and to allow for a non-intrusive design (as monitoring memory loads and stores is easier than monitoring the CPU pipeline). Beyond performance improvements, the reduction in the amount of data means it is possible to move information-flow tracking off the critical path in the architecture, such that the load–store stream is buffered for delayed processing at a more convenient time (while trading prevention for detection, of course).

To evaluate our observation on the structural characteristics of information flows and our design for a predictive tracking mechanism, we consider mobile operating environments such as Google Android on smartphone devices. Mobile devices, though becoming more powerful every year, are significantly constrained in terms of performance and power. Each component either in hardware or software is carefully analyzed for impact on performance and on battery use. In our discussions with engineers from companies in the mobile ecosystem, any overhead over 5% (together with its associated battery impact) is considered unacceptable. Mobile devices commonly suffer from privacy-intruding apps, which collect sensitive data about the user and ship it over the network to a remote service. This combination of factors (strict performance constraints, prevalent well-defined privacy challenges) led us to focus on mobile devices for our evaluation.

In this paper we make the following contributions:

- We observe that information flows in programs have structural characteristics that makes them amenable to analysis and detection using *predictive heuristics* over the instruction stream. We propose the use of such information in a new information-flow tracking mechanism that can be easily deployed in high-performance hardware.
- We design and implement a lightweight and effective heuristic (called PIFT) for information-flow tracking, which relies only on the memory loads and stores issued by the CPU during program execution.
- We present a detailed evaluation on an ARM SoC running Google Android, showing 98% accuracy with limited impact on performance and on added CPU complexity.

2. Overview

The problem of information-flow tracking consists of determining whether sensitive data, once read at program points called *sources*, reaches program points called *sinks*. In the simplest case, a source is a program point where a sensitive data is introduced into the program memory (e.g., a password is read from user input) and a sink is a program point where some data (possibly derived from the sensitive data originated at the source) is sent out of the system.

Threat Model: Since information-flow tracking is useful in security contexts, we define here the type of attacker we consider. Our assumption is that the attacker creates a program to steal sensitive data from a user's device, and convinces the user to run this program on the device. The attacker can design and develop the program in any way they wish, using any programming language, and any supporting runtime libraries they deem suitable. The program is installed on the user's device using standard means for that environment. We make no assumptions about the exact time during execution to access sensitive data or to communicate it. The flow of data from source to sink is of the direct kind, without any indirect, conditional, or control flow-based information flows.

We are interested in mobile environments, and we consider a RISC-style load/store architecture (e.g., ARM), running a software stack designed for interactivity with a single user (e.g., Android OS). It is possible that the techniques in this paper would be applicable to web server, cloud, or scientific workloads, but we have not investigated these options.

Example: Let us consider the following program fragment:

```
String msgX = "type=sms";
...
msgY = msgX + "&imei=" + telMan.getDeviceId();
...
msgZ = msgY + "&dummy";
...
sms.sendTextMessage(phNum, null, msgZ, ...);
```

This code sends out an SMS message that contains the IMEI (International Mobile Equipment Identity) number, a sensitive ID of the mobile device. If we are interested in `getDeviceId()` as a source and `sendTextMessage()` as

a sink, the problem is how to determine most efficiently that msgZ is sensitive because it was derived from the return value of getDeviceId().

Approach: All information-flow tracking techniques applied to the above example rely on establishing the following relationships:

$$\texttt{getDeviceId()} \longrightarrow \texttt{msgY}$$
$$\texttt{msgY} \longrightarrow \texttt{msgZ}$$

where $a \longrightarrow b$ denotes an information flow from memory location a to memory location b. Together with the information about sources and sinks, these relationships are sufficient to determine that there is an information flow from getDeviceId() to sendTextMessage(). Typically the problem is solved through expensive statement- or instruction-level tracking, where an additional bit of information (called *taint*) is associated with memory locations and registers to indicate the presence of sensitive data in those locations and where execution of any instruction is supplemented to propagate these bits from the source operands to destination operands.

We explore an alternate design where *not all instructions are tracked for taint propagation*. In particular we are interested in computing taints for memory locations, so we consider the feasibility of *propagating taints from load instructions to store instructions, irrespective of the interceding data manipulation done via registers*. Our intuition behind this goal of avoiding full register-level tracking of previous work [8, 14, 16] is that code for Android-on-ARM platforms exhibits data flows of predictable lengths, not arbitrarily long, and thus the starting and ending memory operations of these data flows likely have *temporal locality*. This temporal locality is induced by the choice of Java as programming language (e.g., prevalence of heap-allocated data), by the Java virtual machine (e.g., register pressure), and by the architectural characteristics of ARM processors (e.g., load–store instruction set architecture).

To test our hypothesis of temporal locality for load–store data flows, we analyzed the native code corresponding to our Java example. During execution, the strings are copied between the character arrays of the String instances. For example, suppose the application has just executed msgX + "&imei=" and mem[$addr_1$,$addr_2$] denotes the corresponding memory address range. The application then executes telMan.getDeviceId() which returns the IMEI string located at, say, mem[$addr_3$,$addr_4$]. Now, the concatenation of the two strings is carried out by appending the character array of the second string (i.e., the IMEI string) to the end of the array of the first string (which is now storing "type=sms&imei="). The final string is then stored at mem[$addr_1$,$addr_2$+L] where L=$addr_4$-$addr_3$+1 is the length of the IMEI string.[1] Assuming that it is sensitive data and its corresponding memory location (mem[$addr_3$,$addr_4$]) has already been tainted, the goal is to propagate the taint to the corresponding part of the new string, i.e., mem[$addr_2$+1, $addr_2$+L]. To achieve this goal, we need to know *when* the characters are stored into the new location.

[1] In fact, L is the twice of the length of the IMEI string because in Java, each character consumes two bytes.

```
0x4004c114: ldrh r6,[r1, r4] //r6<-mem[addr_s,addr_s+1]
0x4004c116: adds r3, r3, #1
0x4004c118: strh r6,[r0, r4] //r6->mem[addr_d,addr_d+1]
004004c11a: adds r4, r4, #4
0x4004c11c: cmps r3, r5
0x4004c11e: b
0x4004c114: ldrh r6,[r1, r4] //r6<-mem[addr_s,addr_s+1]
0x4004c116: adds r3, r3, #1
0x4004c118: strh r6,[r0, r4] //r6->mem[addr_d,addr_d+1]
004004c11a: adds r4, r4, #4
0x4004c11c: cmps r3, r5
0x4004c11e: b
...
```

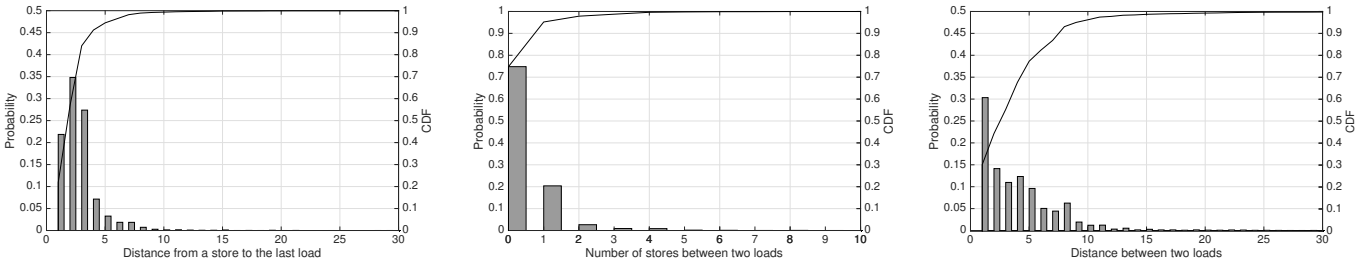
Figure 1: The native code for Java string copy. Each character is loaded into a register and then stored to its destination.

Figure 1 shows the stream of ARM instructions that appends a Java string to another one by copying two bytes at a time. Each character of the string is first loaded from the source address into a register (r6 in this example), and then later it is stored to the destination address. The full-tracking techniques would propagate the taint associated with the source address to register r6 and then to the destination address. Albeit the taint tracking accuracy is high, this register-level tracking requires a significant modification of the processor hardware architecture, not to mention high performance overhead due to the tracking operation for (almost) every instruction (such as add, mov, etc.).

Empirical Study of Load–Store Distances: To better understand how loads and stores are distributed in the instruction stream of a program's execution, we instrumented the gem5 simulator system [2, 6] to obtain instruction level execution traces of Android apps running on an ARM processor. While it is possible for loads and stores to appear anywhere in an instruction stream, we are interested in understanding their distribution in regular Android apps, running in the Android environment. Here, we highlight the statistics for a real-world malicious Android app (known as LGRoot), however, we also analyzed a number of app executions (Section 5).

The first metric we consider is the distance between loads and their subsequent stores. In Figure 2a load–store distances from a trace with 2.2 million memory loads and 768 thousand memory stores are plotted as a probability distribution. It is noticeable that the bulk of load–store distance values cluster in the range 0–5, meaning that most stores follow closely after a load. The distribution tapers off past a distance of 10, such that the range 0–10 captures 99% of all loads and stores. This indicates that it is feasible to track loads and related stores with a small window size.

The second metric is the number of stores following a load, i.e., the number of stores that could potentially be



(a) Distance from most recent load to store (b) Number of stores between consecutive loads (c) Distance between consecutive loads

Figure 2: Memory operations exhibit predictable patterns, as shown by the probability and cumulative distributions for various metrics over memory loads and memory stores in an LGRoot malware execution trace. (distance = number of instructions)

tainted from the previous load. If such a number is high, then a heuristic that considers all of these stores as candidates for tainting would result in over-tainting and a large false-positive rate. As Figure 2b shows, the number of stores between consecutive loads is small and thus naturally limits the taint propagation.

The third metric is the distance between consecutive loads. Figure 2c indicates that loads are fairly uniformly spread throughout the program execution, and thus more amenable to tracking over time.

Together Figure 2 show that most computations in an Android program on ARM consists of short-lived, possibly interleaved, *load–process–store* sequences, where the *process* step is strictly local to the CPU and has no memory operations.

Designing a Load–Store Taint Tracker: Given this structural characteristic of information flows, the tracking mechanism would identify stores that are within the correct distance of a load of sensitive data and taint the destination addresses as sensitive. It is important to note that our approach has to handle a certain amount of imprecision due to the fact that it ignores the actual instructions between loads and stores (i.e., process step).

3. PIFT Architecture

In this section, we discuss the system architecture, PIFT algorithm, and hardware design for PIFT. We especially consider a hardware-based approach because of the needs for fast processing of low-level events and for an OS/application-independent solution.

3.1 System Architecture

PIFT architecture (Figure 3) consists of three main components in different levels of Android software hierarchy.

1. PIFT Manager: We instrument each type of sensitive data source (such as `LocationManager`) so that the data being fetched by an application are registered with our tracking. Similarly, at a sink (such as `SmsManager`) the data being sent out is passed down to the lower level to check if any part of it has been tainted. The registration and check of data are handled by PIFT Manager in the Android Framework level as in [9].
2. PIFT Native: The data is passed down to PIFT Native in the Android runtime layer for address translation. For an object-type data such as IMEI string, it simply obtains the pointer to the data using Java Native Interface (JNI). For a primitive data type such as the GPS location, PIFT Manager passes the object instance that owns the field in addition to the field's name, and then PIFT Native finds the byte offset of the field in the object instance, which is the address of the primitive-type data.
3. PIFT Module: The address is passed down to PIFT Module in the Linux kernel layer. It interacts with the PIFT Hardware Module to register sensitive data's address ranges and make taint queries for check requests. Upon detecting any taint associated with the given address range, it may generate an event to the upper layer to inform of the potential leakage.

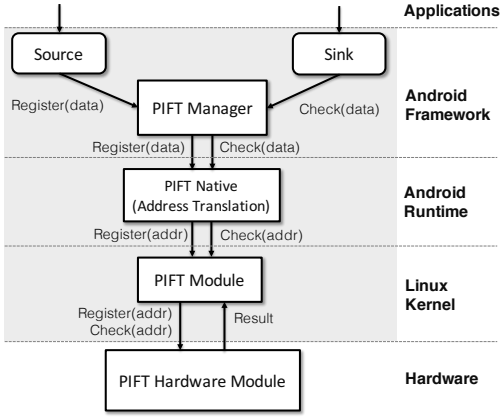


Figure 3: PIFT architecture allows for registering sources and sinks and monitoring data flows.

3.2 PIFT Taint-Propagation Algorithm

Our taint propagation heuristic works based on memory load and store events. Conceptually speaking, upon loading of a piece of sensitive data from a tainted address range, the

Algorithm 1 TAINT PROPAGATION HEURISTIC

1: $\{\mathtt{r_i} = [\mathtt{s_i}, \mathtt{e_i}]$: address range$\}$
2: $\{\mathtt{R} = \{\mathtt{r_i} | 1 \leq \mathtt{i} \leq \mathtt{n}\}$: tainted address ranges$\}$
3: $\{\mathsf{Inst_k}$: the k^{th} CPU instruction$\}$
4: $\{\mathtt{N_I}$: tainting window (TW) size$\}$
5: $\{\mathtt{N_T}$: the maximum number of propagation per TW$\}$
6: $\{\mathtt{LTLT}$: the last tainted-load time (in CPU inst. seq)$\}$
7: $\{\mathtt{n_t}$: the number of taint propagations$\}$

8: $\mathtt{LTLT} \leftarrow -\infty, \mathtt{n_t} \leftarrow 0$
9: **for each** CPU instruction $\mathsf{Inst_k}$ **do**
10: **if** $\mathsf{Inst_k}$ is a memory *load* from $\mathtt{r_L}$ **then**
11: **if** $\exists \mathtt{r_i} \in \mathtt{R}$ that overlaps $\mathtt{r_L}$ **then**
12: {Starts Tainting Window}
13: $\mathtt{LTLT} \leftarrow k$
14: $\mathtt{n_t} \leftarrow 0$
15: **end if**
16: **else if** $\mathsf{Inst_k}$ is a memory *store* to $\mathtt{r_S}$ **then**
17: **if** $k \leq \mathtt{LTLT} + \mathtt{N_I}$ **and** $\mathtt{n_t} < \mathtt{N_T}$ **then**
18: Taint: Add $\mathtt{r_S}$ to $\mathtt{R}$
19: $\mathtt{n_t} \leftarrow \mathtt{n_t} + 1$
20: **else**
21: Untaint (if enabled): Remove $\mathtt{r_S}$ from $\mathtt{R}$
22: **end if**
23: **end if**
24: **end for**

algorithm starts a window of instructions called *Tainting Window* (TW). Then, it taints the target addresses of the next few (but upper-bounded) store instructions in that window.

We formalize our taint propagation heuristic algorithm as follows: Let $\mathtt{R}$ be the set of memory address ranges that have been tainted, $\mathtt{R} = \{\mathtt{r_1}, \mathtt{r_2}, \ldots, \mathtt{r_n}\}$. Each range is defined by $\mathtt{r_i} = [\mathtt{s_i}, \mathtt{e_i}]$ where $\mathtt{s_i}$ and $\mathtt{e_i}$ are the start and end addresses of the range. Given $\mathtt{R}$, the algorithm works as follows as the CPU executes instructions (see Algorithm 1 and an example in Figure 4):

1. [LINE 10–15] For a memory *load* instruction (e.g., `ldr, ldrd, ldmia`) that loads a data from $\mathtt{r_L} = [\mathtt{s_L}, \mathtt{e_L}]$, the algorithm queries if there exists any range $\mathtt{r_i} \in \mathtt{R}$ such that
$$\max(s_i, s_L) \leq \min(e_i, e_L).$$
That is, $\mathtt{r_i}$ overlaps $\mathtt{r_L}$ iff this condition is true. If there exists such an $\mathtt{r_i}$, the algorithm starts (or starts over) the Tainting Window (TW). Its size is measured in the number of instructions from the last tainted-load instruction.
2. [LINE 16–23] For a memory *store* instruction (e.g., `str, strh, stmdb`) that stores a data to $\mathtt{r_S} = [\mathtt{s_S}, \mathtt{e_S}]$,
   (a) [LINE 17–19] If in the Tainting Window and have not exceeded the maximum number of propagations of the current TW, taint $\mathtt{r_S}$ by adding it to $\mathtt{R}$.
   (b) [LINE 20–22] Otherwise, do nothing or, if enabled, untaint $\mathtt{r_S}$ by removing it from $\mathtt{R}$.

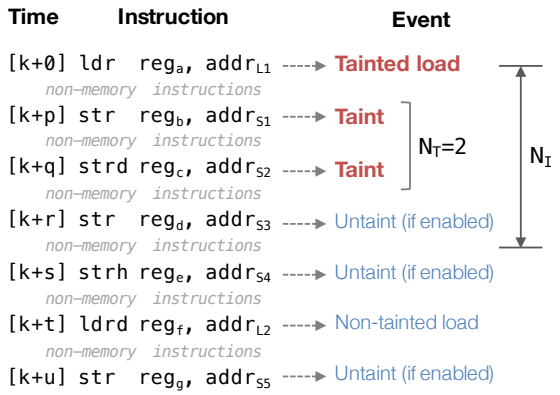


Figure 4: Upon reading of a tainted load (the first line), the Taint Window (TW) of size $\mathtt{N_I}$ starts. Within this window, the target address ranges ($1, 2, 4, > 4$ byte-long, depending on the specific store instruction) of the next $\mathtt{N_T}$ store instructions become tainted. The store instruction at $[\mathtt{k} + \mathtt{r}]$ is not tainted because the maximum number of propagations have completed. The store instructions at $[\mathtt{k} + \mathtt{s}]$ and $[\mathtt{k} + \mathtt{u}]$ are not tainted because they are outside of the TW. If $\mathtt{N_I} > \mathtt{t}$ and if the load instruction at $[\mathtt{k} + \mathtt{t}]$ was a tainted load, then the Tainting Window starts over at $[\mathtt{k} + \mathtt{t}]$.

This window-based taint propagation tries to achieve both accuracy and efficiency. Since we avoid register-level tracking, we do not know which store instruction(s) write back the processed data to the memory, or even whether this would ever happen or not. For example, in Figure 4, a sensitive data loaded to $\mathtt{reg_a}$ could be written to (after some processing) $\mathtt{addr_{S1}}$ or $\mathtt{addr_{S2}}$ or neither of them. Hence, the propagation algorithm taints multiple ranges (i.e., *overtaints*) and increase the chances of tracking sensitive data flow. However, we limit the number of propagations in a TW in order to prevent the tainted regions in the memory from exploding. We further reduce the tainted regions by *untainting* the target addresses of the store instructions, which have been tainted, that do not fall within a TW because they are likely overwritten by a non-sensitive data. It is, however, possible that they actually contain sensitive data, in which case we might create a false negative. However, our experimental results in Section 5 indicate that untaintings do not degrade the detection accuracy while significantly reducing the tainted regions.

3.3 PIFT Hardware Module

The high-level architecture that illustrates how the on-chip PIFT hardware module (PIFT HW) interacts with the CPU is shown in Figure 5. The PIFT front-end logic, added in the CPU, performs two operations: it 1) tracks the instructions executed by the CPU's instruction unit and 2) generates events to PIFT HW upon observing memory access instructions. It also maintains the instruction counter for each process (indexed by a process-specific ID such as Process

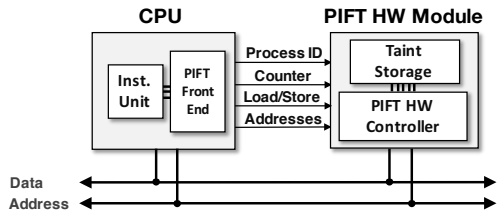


Figure 5: PIFT hardware architecture.

ID (PID) or Translation Table Base Register (TTBR)). For each memory access instruction, PIFT HW receives from the front-end logic 1) the process-specific ID, 2) the process-specific instruction counter, 3) memory access type (load or store), and 4) the read or write address range. Then the PIFT HW controller performs the taint propagation heuristic using the given information and the attached taint storage while the memory sub-system is handling the memory access. For a software-level registration at source (i.e., tainting a new address range), a query at sink (i.e., checking a range's taint), and configuration (parameter setting $\mathtt{N_T}$ and $\mathtt{N_I}$), PIFT software module sends commands and receives responses through an array of memory-mapped ports of PIFT HW. Note that the SW module does not interact with the HW module most of the time; taint lookup and propagation operations are transparent to the software side.

The most frequent operation of the taint tracking is the taint lookup operation, which is carried out on every memory load instruction. Thus, the taint storage is desired to have a very fast lookup-time. One possible option is a cache-like on-chip memory as the taint storage as used in [16, 17, 19]. In our architecture, we use a cache of ranges in which each entry stores arbitrary-length tainted ranges, similar to [17]. Figure 6 shows the structure of our taint storage and illustrates the lookup procedure. Each entry holds the start and end addresses of each tainted range, the associated process-specific ID, and a valid bit. A lookup is a parallel operation; it is hit if any entry 1) has the same process ID, 2) is valid, and 3) contains a range that overlaps (irrespective of being fully/partially) with the one being queried. If hit, the PIFT HW controller starts a new tainting window. The same procedure is carried out for software-level query.

Because of this cache-like taint storage, a query can be very fast and thus take a constant time, provided that the storage is large enough. As will be shown in Section 5.2, the number of ranges created during tracking is sufficiently small so that an on-chip memory with a reasonable size can handle tainted ranges without a secondary storage at, for example, a part of the main memory. To be more specific, each range takes up 12 bytes (excluding the valid bit) - 4 byte for start and end addresses, respectively, and additional 4 bytes for PID value. Accordingly, a small on-chip memory, for example, of 32KB can accommodate approximately 2730 ranges at the same time. If a secondary storage is allocated on the main memory and the entire range entries

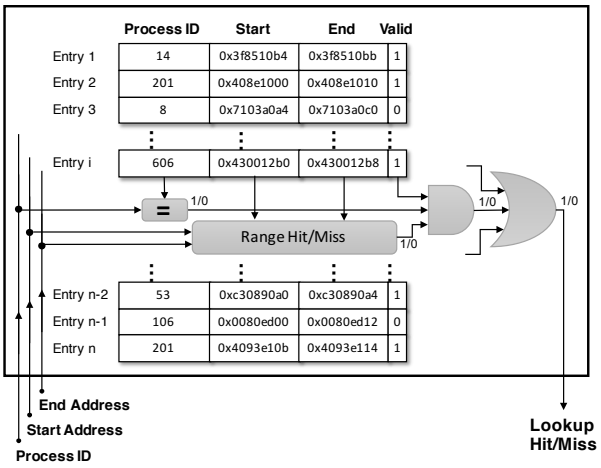


Figure 6: Taint storage and lookup operation.

are written back when a context switch occurs, we can remove the process-specific identification for each entry and thus can store 4096 entries in the 32KB memory. If, however, the storage is not sufficient, one may evict some of the existing entries to the main memory using a replacement algorithm such as LRU (Least Recently Used) as in [17] or may simply drop it. The former case resembles the ordinary data/instruction caches, and hence it may experience delays when there is a 'cache miss'. The latter case does not exhibit a performance overhead, however it may increase the possibility of false negative because it may lose some sensitive data flow.

Another possible option is to taint addresses at a fixed granularity such as a word. That is, instead of keeping track of ranges of arbitrary lengths, we can taint a block as a whole if any part of the block is being tainted. This can be done by storing the $(32 - r)$ most significant bits of the tainted addresses if the granularity is 2^r bytes. This can reduce the size per entry of the taint memory to 4 bytes (or 8 bytes if each entry is tagged with process-specific ID), and also simplify the hardware logic and make queries faster due to the fewer comparisons. However, this may lead to overtaintings. The consequence could be more propagations, hence possibly more entries might be needed than was the case with arbitrary range sizes, and also false positives might be introduced. Witchel *et al.* [20] presents a multi-level address space partitioning method that can associate an arbitrary range with a tag by a series of power-of-two sized ranges.

4. Technical Analysis

The existence of a predictable distance between related loads and stores, empirically measured in Section 2, allowed us to design an end-to-end system as described in Section 3. We have yet to explain analytically why this predictable distance exists for Java applications running on top of the Android-on-ARM platform. This analysis is the purpose of this section.

```
void foo() {                                          Java
    …
    int key = 123;  //v1
    String msg = "2*key+456="  + bar(key,456);
    …
}
```

```
...                                                   Bytecode
move v5, v1
const/16 v6, #int 456
invoke-direct {v4, v5, v6}, MainActivity;.bar
move-result v4
invoke-virtual {v3, v4}, StringBuilder;.append
move-result-object v3
invoke-virtual {v3}, StringBuilder;.toString
move-result-object v3
...
```

```
int bar(int x /*v1*/, int y /*v2*/) {                 Java
    return 2*x + y;
}
```

```
const/4 v3, #int 2                                    Bytecode
move v4, v1
mul-int/2addr v3, v4
move v4, v2
add-int/2addr v3, v4
move v0, v3
return v0
```

Figure 7: Java code and corresponding Dalvik bytecode.

```
/* mul-int/2addr vA, vB */

1: mov     r3, rINST, lsr #12     @ r3<- B
2: ubfx    r9, rINST, #8, #4      @ r9<- A
3: GET_VREG(r1, r3)               @ r1<- vB
4: GET_VREG(r0, r9)               @ r0<- vA
5: FETCH_ADVANCE_INST(1)          @ advance rPC, load rINST
6: mul     r0, r1, r0             @ r0 <- op, r0-r3 changed
7: GET_INST_OPCODE(ip)            @ extract opcode from rINST
8: SET_VREG(r0, r9)               @ vAA<- r0
9: GOTO_OPCODE(ip)                @ jump to next instruction
```

Figure 8: Dalvik bytecode `mul-int/2addr` to native code translation.

4.1 Load–Store Distances in Native Code

Android applications are executed by Dalvik virtual machine (VM) that translates each Dalvik bytecode to a sequence of native instructions [3]. Two examples of Java code and their corresponding bytecodes are shown in Figure 7 and the translation rule of bytecode `mul-int/2addr` is shown in Figure 8.

Dalvik is a register-based VM in which the operands of bytecode are *virtual registers*. For example, '`mul-int/2addr v3, v4`' multiplies the value in virtual register `v3` by the one in `v4` and then writes the result back to `v3`. Our taint tracking heuristic method takes advantage of the property that *virtual registers reside on the memory*. Hence, each bytecode involves loading operands from the memory and storing the resultant to the memory, if produced. In Figure 8, these correspond to '`GET/SET_VREG(_reg, _vreg)`' which are macros defined as

```
ldr _reg, [rFP, _vreg, lsl #2],
str _reg, [rFP, _vreg, lsl #2],
```

```
0x407c7c40: mov   r3, r7, LSR #12
0x407c7c44: ubfx  r9, r7, #8, #4
0x407c7c48: ldr   r1, [r5, r3 LSL #2]     r1←v4   Tainted Load
0x407c7c4c: ldr   r0, [r5, r9 LSL #2]     r0←v3   Non-tainted Load
0x407c7c50: ldrh  r7, [r4, #2]!
0x407c7c54: mul   r0, r1, r0
0x407c7c58: and   r12, r7, #255
0x407c7c5c: str   r0, [r5, r9 LSL #2]     r0→v3   Tainting
0x407c7c60: add   pc, r8, r12, LSL #6

0x407c5000: mov   r1, r7, LSR #12
0x407c5004: ubfx  r0, r7, #8, #4
0x407c5008: ldrh  r7, [r4, #2]!
0x407c500c: ldr   r2, [r5, r1 LSL #2]     r2←v2   Non-tainted Load
0x407c5010: and   r12, r7, #255
0x407c5014: str   r2, [r5, r0 LSL #2]     r2→v4   Untainting
0x407c5018: add   pc, r8, r12, LSL #6

0x407c7bc0: mov   r3, r7, LSR #12
0x407c7bc4: ubfx  r9, r7, #8, #11
0x407c7bc8: ldr   r1, [r5, r3 LSL #2]     r1←v4   Non-tainted Load
0x407c7bcc: ldr   r0, [r5, r9 LSL #2]     r0←v3   Tainted Load
0x407c7bd0: ldrh  r7, [r4, #2]!
0x407c7bd4: add   r0, r0, r1
0x407c7bd8: and   r12, r7, #255
0x407c7bdc: str   r0, [r5, r9 LSL #2]     r0→v3   Tainting
0x407c7be0: add   pc, r8, r12, LSL #6
```

Figure 9: Native codes translated from `mul-int/2addr`, `move`, and `add-int/2addr` in Figure 7.

respectively. Thus, if there is a data movement (with or without processing) from a virtual register to another within a bytecode, we can find a pair (or more) of load and store instructions associated with them. More importantly, because of the pre-defined translation rules, the distance between the loads and stores *cannot be arbitrary*. For instance, if one of the operands of '`mul-int/2addr v3, v4`', loaded at line 3 and 4 in Figure 8, were tainted, a tainting window of size 5 could propagate the taint correctly to the resultant (line 8).

Accordingly, the data movement over a sequence of bytecodes can be tracked by matching such load and store instructions. Figure 9 shows a gem5 trace of native codes that are translated from the sequence of bytecodes `mul-int/2addr`, `move`, and `add-int/2addr` used in Figure 7. Suppose the sensitive data, `key`, has been tainted. It has been placed in virtual register `v4` (which is copied from `v1`, the first argument of function `bar`, by `move` bytecode). By our heuristic, the taint of `v4` is propagated to `v3` if the TW size was set to at least 5. The next bytecode, `move`, overwrites `v4` with the value in `v2` which in fact is the argument `y`. This store instruction is outside of the TW started by the load of `v4`, and thus becomes untainted. In the last block of native codes (translation of `add-int/2addr`), the load of `v3` starts a new TW which then propagates the taint to itself.

The proper window size varies with bytecode as we can see from the example above; for `add-int/2addr`, the TW size should be at least 5 while `move` needs a size of 2. To examine the adequate sizes for Dalvik bytecodes, we first categorized the total 256 bytecodes into the ones that can move data around in memory and the others (e.g., method invocations, constants, and if-statements). The latter includes 74 bytecodes, of which 10 are unused (based on armv7-a architecture). Then, we examined each of the bytecode in the former group, and measured the longest distance between

Load-Store Distance	Cnt	Example Bytecodes
1	3	`return, return-wide, return-object`
2	26	`move-result, move/16, aget, aput, sput,iput-quick`
3	19	`move-object, sget-object, long-to-int, sget`
4	11	`iput-iput, neg-double, iget-quick, sget-volatile`
5	46	`iget, iget-object, int-to-long, add-int/lit8`
6	21	`int-to-char, sub-long, shl-int/lit8, iget-volatile`
9–12	9	`mul-long/2addr, aput-object, mul-long, shr-long`
Unknown	47	`double-to-int, rem-float, div-int/lit16`

Table 1: Native load & store distances within Dalvik bytecodes.

the loads of actual data and the store instruction. The results summarized in Table 1 show that most of the bytecodes have a short load-store distance. There exist 47 bytecodes of which load-store distances were not measured. Those are the ones for floating-point arithmetic and call ARM's runtime ABI (Application Binary Interface) helper-functions such as `__aeabi_fadd` [1]. Thus, their required window sizes are unknown.

We have also measured how frequently each bytecode is used by analyzing the dex files of the Android system libraries (Core, Framework, and Services) and those of the stock applications (Browser, Contacts, Email, KeyChain, Music, Calculator, Gallery, Phone, Calendar, Dialer, HTMLViewer, Mms, SpeechRecorder). The tables in Figure 10 show the top 30 bytecodes in the number of appearances in applications as well as the system libraries. Most of the frequently appearing bytecodes have a short load-store distance. One exception is `aput-object` bytecode, which puts an object reference into an array of object, that appears more frequently in applications. The relatively long load-store distance is due to type checking. Most of the bytecodes that have long or unknown load-store distances take a very small percentage of the examined lines (less than 0.1% or 0.01%) or even did not appear in the collected dex files. These results indicate the effectiveness of our taint propagation heuristic specifically in Android.

Impact of Dalvik JIT and ART AOT Compilation: While we have run all of our experiments with the default Dalvik optimization settings and the optimization has not impacted our algorithm accuracy, we are interested to investigate the effect of more aggressive optimization on PIFT accuracy. More aggressive optimizations might eliminate or reorder some of the load and store instructions and potentially make it more difficult to predict data tracking. Our initial testing of running apps with and without JIT (Just-In-Time) optimization has shown little impact on the distribution of load and

Dalvik Bytecode	%	L-S Distance
invoke-virtual	11.06%	
move-result-object	8.98%	2
iget-object	7.10%	5
const/4	5.19%	
const-string	4.85%	
invoke-static	4.45%	
move-result	4.42%	2
invoke-direct	4.31%	
return-void	3.19%	
goto	3.10%	
invoke-interface	3.04%	
const/16	2.82%	
if-eqz	2.82%	
return-object	2.79%	1
aput-object	2.50%	10
new-instance	2.36%	
iput-object	1.97%	5
move-object/from16	1.84%	2
return	1.68%	1
iget	1.46%	5
if-nez	1.40%	
check-cast	1.31%	
sget-object	1.09%	3
add-int/lit8	0.80%	5
iput	0.74%	4
move	0.68%	3
move/from16	0.65%	2
throw	0.64%	
const	0.60%	
move-object	0.53%	3

(a) Applications (1.2M lines)

Dalvik Bytecode	%	L-S Distance
invoke-virtual	12.57%	
iget-object	7.51%	5
move-result-object	7.46%	2
const/4	5.64%	
invoke-direct	4.57%	
move-result	4.16%	2
const-string	3.84%	
invoke-static	3.59%	
goto	3.30%	
if-eqz	3.26%	
move-object/from16	3.22%	2
return-void	2.83%	
iget	2.60%	5
new-instance	2.57%	
iput-object	1.76%	5
if-nez	1.61%	
invoke-interface	1.57%	
const/16	1.50%	
return-object	1.44%	1
throw	1.30%	
iput	1.27%	4
return	1.17%	1
move/from16	1.13%	2
move-exception	1.12%	
add-int/lit8	0.96%	5
check-cast	0.95%	
sget-object	0.91%	3
monitor-exit	0.82%	
invoke-virtual/range	0.74%	
move	0.74%	3

(b) System libraries (1.3M lines)

Figure 10: The distribution of the top 30 Dalvik bytecodes in the number of appearances in Google stock applications and Android libraries dex files. The bytecodes in the highlighted cells are the ones that can move data, irrespective of being a real data or a reference to it. The third column of each table is the load-store distance of such bytecodes.

store distances. For example, we profiled the memory operation profile as in Figure 2 without JIT, but the patterns were identical.

ART (Android RunTime) is an ahead-of-time compiler for Dalvik bytecode and has become the preferred platform for the most recent Android versions. ART compiles the bytecode of individual Java methods into native versions and provides the linking environment to route calls between the compiled methods. The optimizations applied by ART are a limited subset of Dalvik's JIT optimizations and thus ART does not impact the accuracy of our taint-propagation algorithm.

4.2 Limitations

Native code obfuscation: If an attacker can insert an arbitrarily long, dummy block of native codes between a load of sensitive data and a store using JNI (Java Native Interface), PIFT can be circumvented. For example, one may pass the IMEI `String` instance to the JNI level, obtain the pointer to the character array, load each character onto a register, using `ldr` instruction, do a series of dummy computations, then store the register (i.e., an IMEI character) to the memory. If the extra dummy code is not optimized out by the native code compiler, this could copy the IMEI string to another without any taint propagation.

Implicit flows: A sensitive information flow can be created through control flows [12], which PIFT does not directly address. However, some types of implicit flows are detected by PIFT due to their temporal locality. As an example, a detected case in one of the tested DroidBench applications, `ImplicitFlows_ImplicitFlow1` [4] works by obfuscating the IMEI and then leaking it:

```
for(char c : imei.toCharArray()){
   switch(c){
      case '0' : result += 'a'; break;
      case '1' : result += 'b'; break;
      case '2' : result += 'c'; break;
      ...
```

The `imei` string is already tainted, hence each character load starts a tainting window. Then, since a writing to the `result` string (i.e., each `case` statement) is close enough to the tainted load (in terms of the number of native instructions), the obfuscated string, `result`, gets tainted.

5. Evaluation

The two major goals of a data leak detection algorithm are 1) maximum accuracy and 2) minimum overhead, especially if the system is resource-constrained (e.g., mobile devices). For the evaluation of PIFT, we used DroidBench 1.1 [4] as well as seven real-world malware apps. The test set is representative of real-world apps and their challenges, as it moves data through arrays, lists, callbacks, exceptions, intents, and obfuscates control flow through method overriding, reflection, and object inheritance. In our experience of manually analyzing hundreds of malicious and benign Android apps, the vast majority of information flows are *direct*. Thus we focus on optimizing the common case, and we consciously choose to leave implicit flows and obfuscated apps for future work.

We used the DroidBench apps (a total of 41 leaky and 16 benign apps that ran on gem5 simulator [2]) as-is except for minor modifications for the purposes of automation (so that they can run without user interaction). The sources used in DroidBench are device ID (IMEI), serial number, phone number, and location. The sinks are SMS messages, HTTP queries, and logging functions. We used Android 4.2 Jelly Bean (the highest Android version supported by gem5 simulator), with the Dalvik JIT enabled. In our experiments, the PIFT Native (Figure 3) just prints out the address ranges of source and sink, which then are fed into the PIFT analysis code along with the CPU instruction stream trace obtained by gem5 simulator.

5.1 Accuracy Evaluation

We tested the 57 DroidBench apps and investigated the impact of window size on accuracy. We tried 200 combinations of $\mathtt{N_I} = [1, 20]$ and $\mathtt{N_T} = [1, 10]$. Figure 11 shows the accuracy heatmap, i.e., (true positive + true negative)/total, of DroidBench apps. When $\mathtt{N_I} < 10$, PIFT was not able to detect an app that sends out GPS location, which requires floating point-to-string transformation through an ARM runtime ABI. $\mathtt{N_I}$ had to be at least 10 for PIFT to detect such a case. Using a window size of $\mathtt{N_I} = 13$ and $\mathtt{N_T} = 3$, PIFT was able to achieve 98% accuracy, 0% False Positive (0 out of 16) and 2% False Negative (1 out of 41). The one app we did not detect has an implicit data flow which makes it difficult (but not impossible) to detect. To achieve a 100% accuracy, the windows size should be set to $\mathtt{N_I} = 18$ and $\mathtt{N_T} = 3$. We believe there is a proper upper-bound on the window size for each leakage type, which could be found from a future large-scale experiment.

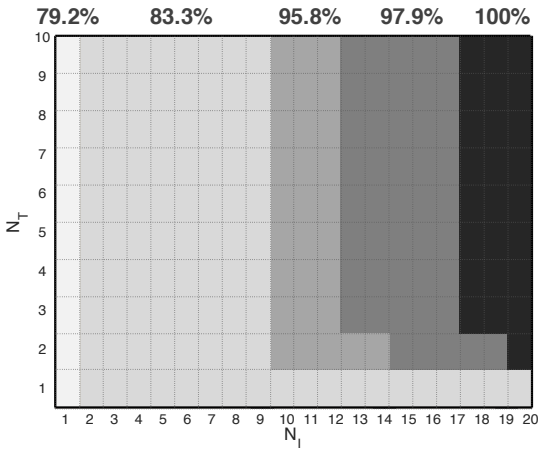


Figure 11: Accuracy of a subset of DroidBench apps for all possible window sizes of $\mathtt{N_I} = [1, 20]$ and $\mathtt{N_T} = [1, 10]$. For the window size of $\mathtt{N_I} = 13$ and $\mathtt{N_T} = 3$ the accuracy reaches 97.9%.

In all experiments, no false positive occurred. False positive is not likely to occur because the tainting window should be so large that it can span across multiple Java statements. Note that one Java statement is expanded into multiple bytecodes and each bytecode is translated into multiple native instructions, therefore the distance between Java statements will be large in terms of the number of native instructions and with high probability out of the window size. Furthermore, a mis-tainting does not necessarily lead to a false positive because a mis-tainted range should be sent out through a sink to cause a false positive. This, however, is not likely to happen.

We have also tested seven real-life malware samples, which target Android mobile devices and send out various data, such as phone number, location, and device ID, over HTTP connections and SMS messages. The malware samples include four rootkits and three CPU over-clocking apps from a Chinese app store. PIFT successfully detected all the malware given a very small window size of $\mathtt{N_I} = 3$ and $\mathtt{N_T} = 2$. This result shows our algorithm is very effective for most of current data leaking apps. Here we present the micro-benchmark results for one of the tested rootkits to give insight into the detection rate of PIFT.

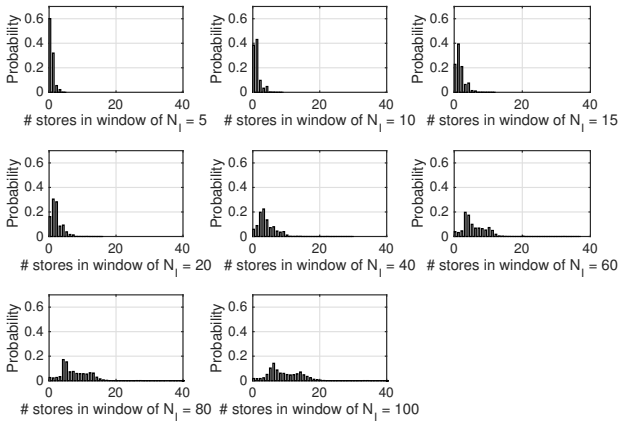


Figure 12: Probability distributions of number of stores in window of size $\mathtt{N_I} = 5, 10, 15, 20, 40, 60, 80, 100$ (LGRoot trace).

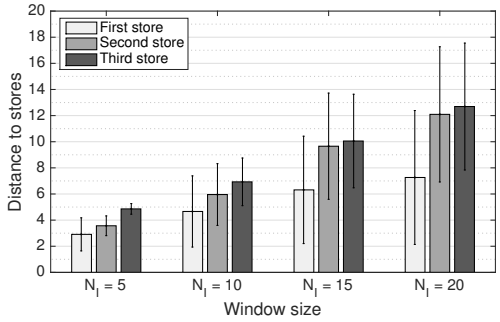


Figure 13: The average distance to the 1^{st}, 2^{nd}, and 3^{rd} stores within an window of size $\mathtt{N_I} = 5, 10, 15, 20$ (LGRoot trace).

Load–store distance: For every load instruction, we counted the number of stores in the next $\mathtt{N_I}$ instruction, and calculated the probability distribution for different window sizes. As Figure 12 indicates, small window size is acceptable because of the diminishing returns; increasing the window size above 10 or 15 does not capture more stores.

Load–store extended distances: We extended the previous experiment by measuring the distances to the first, the second, and the third stores in the window. Figure 13 shows that the stores are in close proximity of loads, and as a result, we can taint all the three stores after a load without taint explosion.

5.2 Overhead Evaluation

We have analyzed the runtime overhead of a real-world malware—known as LGRoot—with respect to the tainting window parameters and the effects of untainting. Due to the intrinsic limitation of cycle-inaccurate simulation, we

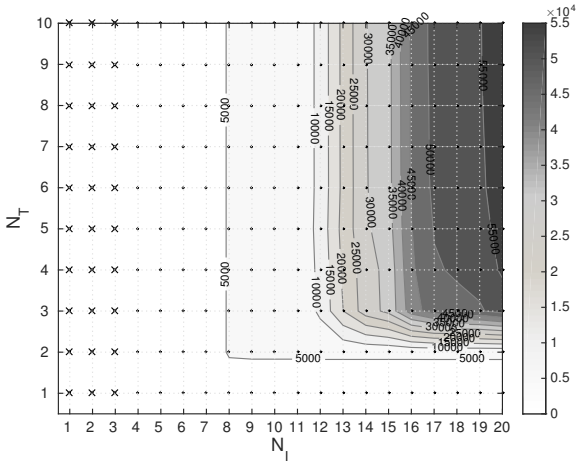


Figure 14: Maximum size of tainted addresses for different tainting window size parameters ($\mathtt{N_I}$ and $\mathtt{N_T}$).

performed detailed analyses on factors affecting runtime overhead, e.g., the number of operations, and the size and number of tainted regions.

Figure 14 shows the maximum size (in bytes) of tainted addresses over time with varying size of tainting window defined by $\mathtt{N_I}$ and $\mathtt{N_T}$. The heatmap clearly shows the increasing trend of tainted regions with tainting window parameters. Also, $\mathtt{N_T}$ (the maximum number of propagations in a window) outweighs $\mathtt{N_I}$ (the tainting window size) in its effect on the tainted region size. This is because a larger $\mathtt{N_I}$ increases the chance of a propagation, whereas a larger $\mathtt{N_T}$ increases that of overtainting. When the window size is small, $\mathtt{N_T}$ does not affect much since we would likely not see the additional store instructions in such a small window. Due to the similar reason, $\mathtt{N_T}$ becomes a critical factor for long windows.

Figure 15 shows how the tainted region grows as time progresses, for different window parameters ($\mathtt{N_I} = \{5, 10, 15, 20\}$, $\mathtt{N_T} = \{1, 2, 3\}$). The sensitive data (i.e., IMEI) is

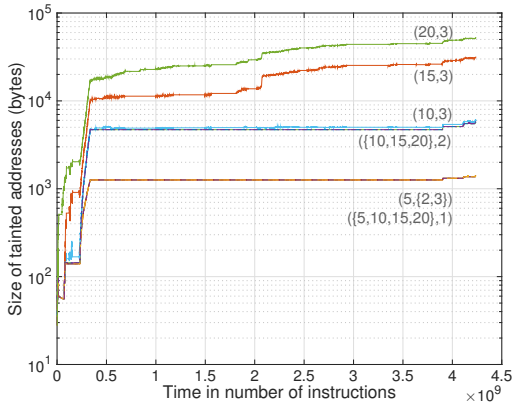


Figure 15: Size of tainted addresses for different tainting window size parameters ($\mathtt{N_I}$ and $\mathtt{N_T}$) as time progresses.

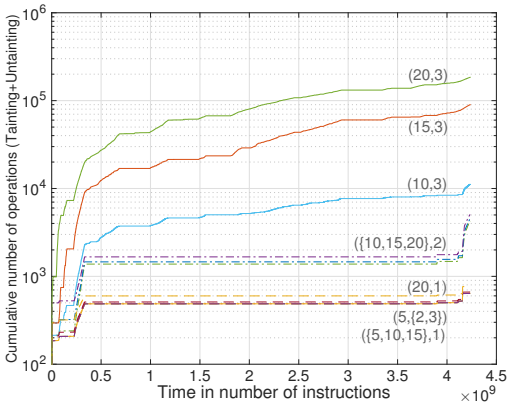


Figure 16: Number of operations (tainting + untainting) for different tainting window size parameters ($\mathtt{N_I}$ and $\mathtt{N_T}$) as time progresses.

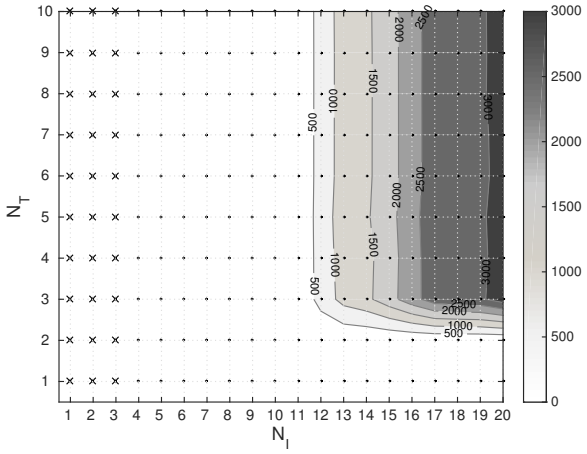


Figure 17: Number of distinct ranges for different tainting window size parameters ($\mathtt{N_I}$ and $\mathtt{N_T}$).

fetched at the beginning and then used to compose a message. It is sent out at the end of the trace. Here, notice the flat lines corresponding to $(\mathtt{N_I}, \mathtt{N_T}) = (\{5, 10, 15, 20\}, \{1, 2\})$ and $(5, 3)$ between them. It is a period when the taints do not propagate because of inactivity on the sensitive data. This can be checked from Figure 16 that shows the number of tainting and untainting operations over time.

The case of $(10, 3)$ is a bit different. The tainted regions do not grow during the period, whereas the tainting/untainting operations keep occurring. This can be explained by situations where some small regions are repeatedly mistainted and then untainted or retainted. If the level of overtaintings exceeds a certain level, those produce a far more overtaintings, which eventually expands the tainted regions exponentially as evidenced by the cases of $(15, 3)$ and $(20, 3)$ in Figure 15. If the parameters are selected properly,

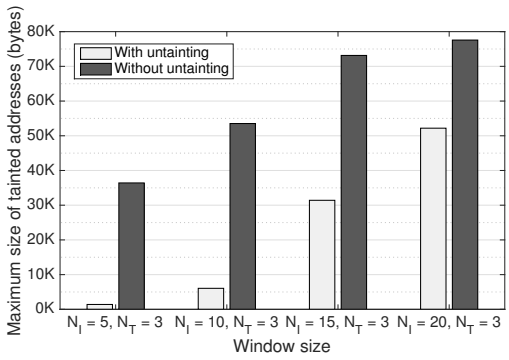


Figure 18: Effect of untainting on the maximum size of tainted addresses.

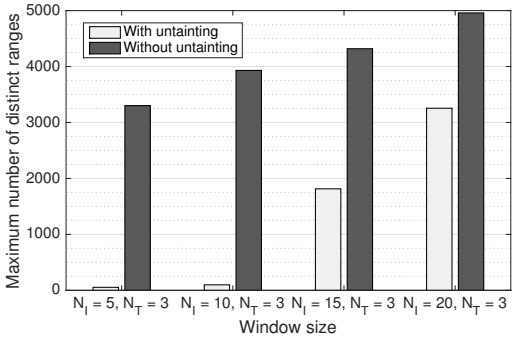


Figure 19: Effect of untainting on the number of distinct ranges.

the tainted regions can be kept reasonably small while not missing sensitive data flows as previously seen.

The number of distinct ranges maintained by PIFT is another important metric to measure the performance overheads since the operations (i.e., query, tainting, untainting) are performed on address ranges. Hence we measured the maximum number of distinct ranges over time for different tainting window size parameters as shown in Figure 17. The trend is similar to that in Figure 14.

For window sizes not larger than $\mathtt{N_I} = 10$, there were less than 100 distinct ranges at any time instant over the trace. This result implies that the memory required to store the ranges can be kept very small, which in turn make it possible to make taint queries very fast by removing the need for a secondary storage.

Effects of untainting: The untainting technique can remove possibly-mistainted, non-sensitive data from the tainted ranges and thus can reduce the possibility of tainted region explosion which may introduce significant performance overheads. Hence we evaluate how much untainting can help reduce the performance-related metrics.

Figure 18 compares the maximum sizes of tainted regions when untainting is enabled or disabled. The results show sig-

nificant reductions in the size of tainted regions due to the untaintings; for the case of $\mathtt{N_I} = 5$ and $\mathtt{N_T} = 3$, untainting resulted in 26 times smaller tainted regions. Without untainting, the varying window size does not make a considerable difference, whereas with untainting enabled, shorter window could significantly reduce the tainted regions.

The reduction in tainted regions led to fewer number of distinct ranges as shown in Figure 19 which compares the maximum number of distinct address ranges when untainting is enabled or disabled. For the case of $\mathtt{N_I} = 5$ and $\mathtt{N_T} = 3$, more than 60 times fewer number of distinct ranges were kept when untainting was enabled.

6. Related Work

Information-flow tracking techniques can be *dynamic*, where a program's execution is monitored and analyzed, or *static*, where a program's code is analyzed ahead of execution.

Hardware-based instrumentation: Hardware-based techniques extend the processor architectures to facilitate taint propagation in a fine-grained, software-transparent manner. Suh *et al.* [16] introduced a hardware mechanism in which data coming through network I/O channels are tainted and the CPU is prevented from executing such tainted addresses. The taint propagation is carried out at every register- and memory byte-level by adding taint tags to caches, TLBs, and register blocks. Raksha [8] follows a similar hardware mechanism with additional flexibility by making the propagation and check rules programmable via multi-bit tags that represent different security policies. Flexitaint [19] provides full flexibility in specifying the taint propagation and check rules by allowing software to handle instructions with different rules. Tiwari *et al.* [18] proposed a processor architectural technique that can track complete information flow at the logic gate-level.

Software-based instrumentation: Software-based techniques such as TaintCheck [14] and Panorama [22] use emulation environments or virtualization techniques to enable program instrumentation, where every CPU instruction is checked and therefore incurs a huge overhead. A line of work has focused on *efficient* taint tracking. LIFT [15], built on top of a binary translator, eliminates unnecessary flow tracking when programs executes computations that involve safe data. It also reduces the number of checks by considering spatial and temporal localities of memory references. Ho *et al.* [11] reduces the overhead associated with hardware emulation by dynamically switching between emulated execution (by Qemu) and virtualized execution. TaintEraser [23] performs an application-level taint tracking to reduce overhead due to system-wide instrumentation. It tracks object-level taints at the kernel level and in certain highly utilized functions and turns off tracking to avoid huge overhead. TaintDroid [9] detects data leakage of mobile malware by instrumenting the Dalvik VM interpreter at variable granularity. The Dalvik bytecode translations rules are patched to include taint propagation logic. For native code, TaintDroid does not track taints but apply a heuristic that propagates the taint of input arguments to that of the return value of functions in native code. DroidScope [21] modifies the Android emulator to instrument native and Dalvik instruction traces, API calls, and also to track taints.

Static analysis: AndroidLeaks [10] uses a call graph extracted from an application under test. It applies a reachability analysis to find if there is any path from a sensitive data source to a network sink. Since the analysis is restricted at the Java code-level, flows that involve native code executions cannot be tracked. FlowDroid [5] uses not only the source code but also auxiliary information obtained from application manifest and layout files. Such information are used to identify the application's lifecycle, sensitive data source and sink, which helps reduce both false positives and negatives. FlowDroid does not track native code either. AAPL [13] improves the accuracy by identifying conditional sources and sinks which depend on the input arguments to them, and joint data flow which could potentially be a sensitive data flow when multiple, non-sensitive flows become connected during run-time. EdgeMiner [7] deals with imprecision due to implicit sensitive data flows (especially by callback functions) by analyzing the Android framework and identifying all implicit flows (that is, pairs of callback and registration methods) within the framework.

7. Conclusions

We hypothesized and empirically verified that information flows in programs have stable structural characteristics. Based on these characteristics we introduced in this paper a new lightweight and effective information-flow tracking system called PIFT. PIFT monitors memory accesses (loads and stores) and propagates taint tags across them in a predictive fashion, achieving 98% accuracy on a simulated ARM processor running Google Android, with limited overhead and limited added hardware complexity.

As discussed in Section 4.2, PIFT could be circumvented if the load-store distances are controlled using native-level instructions. A compiler support for PIFT could address such attacks. For example, the compiler could eliminate dummy code inserted between related load/store instructions and could relocate such instructions to be closer to each other (although in general the problem of statically identifying dummy code is undecidable). This necessitates a follow-up study.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions. This work was conducted during Man-Ki Yoon's internship at Qualcomm Research.

References

[1] Run-time ABI for the ARM architecture. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0043d/IHI0043D_rtabi.pdf.

[2] Bbench-gem5. http://www.m5sim.org/BBench-gem5.

[3] Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html.

[4] DroidBench Version 1.1. http://sseblog.ec-spride.de/tools/droidbench/.

[5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2014.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[7] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *Proceedings of the 22nd Network and Distributed System Security Symposium*, NDSS, 2015.

[8] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA, 2007.

[9] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2010.

[10] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST, 2012.

[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*, EuroSys, 2006.

[12] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the 18th Network and Distributed System Security Symposium*, NDSS, 2011.

[13] K. Lu, Z. Li, V. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *Proceedings of the 22nd Network and Distributed System Security Symposium*, NDSS, 2015.

[14] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Network and Distributed System Security Symposium*, NDSS, 2005.

[15] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2006.

[16] G. E. Suh, J. Lee, 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, 2004.

[17] M. Tiwari, B. Agrawal, S. Mysore, J. Valamehr, and T. Sherwood. A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2008.

[18] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2009.

[19] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Proceedings of the 14th IEEE International Symposium on High Performance Computer Architecture*, HPCA, 2008.

[20] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2002.

[21] L. K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security, 2012.

[22] 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, 2007.

[23] 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, 2011.