JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 33, 1237-1254 (2017)
DOI: 10.6688/JISE.2017.33.5.8

DROIT+: Taint Tracking for Storage Access on Android*

CHIA-WEI HSU, CHIA-HUEI CHANG, CHI-WEI WANG+ AND SHIUHPYNG SHIEH
Department of Computer Science
National Chiao Tung University
Hsinchu, 300 Taiwan
Email: {hsucw; changjh; ssp}@cs.nctu.edu.tw; cwwangabc@gmail.com+

The leakage of sensitive data has been a major concern in Android ecosystem. Analysts therefore propose dynamical taint tracking to effectively track the data flow of accessed data. However, the off-the-shelf taint tracking systems lack byte-granularity support for storage tracking. In this paper, we propose DROIT+ which uses the fine-grained storage tracking technique to track data flow among Android storages. DROIT+ is able to reveal the composition of data flows. Storage tracking on Android is difficult since data flows of apps may span heterogeneous media including memory, SD cards, NAND Flash, and network adapters. To capture a whole picture of data flows in storage, we formally define data flow and propose our method from both logical and physical perspectives. The method has also been implemented as an extension to the proposed tracking system, DROIT. Two case studies and two benchmark tools are used for the evaluation in terms of storage tracking ability, network tracking ability, and efficiency, respectively. The result shows that DROIT+ provides a better coverage using byte-granularity taint tracking.

Keywords: mobile security, information flow, Android, file system, taint tracking

1. INTRODUCTION

Android is the most popular mobile system occupying over 82% market share. At the same time, it is exposed to malware attacks, mostly the data theft. Instead of breaking the system, the personal data is more valuable to adversary [19]. Most of malicious behavior can be associated with storage access. Take *Backdoor.AndroidOS.Obad.a* (found by Kaspersky in June 2013) as an example. It stealthily sends user contacts to remote servers, and then downloads malware to infect neighbor devices via Bluetooth. These stolen data or downloaded malware are usually encrypted or obfuscated to evade elementary detections such as pattern matching and packets sniffing. Therefore, analysts use the dynamic information flow analysis [9, 17, 18], called taint tracking, to monitor the data of interest. These taint tracking systems effectively uncover the malware behaviors to analysts.

Based on object-level tracking, TaintDroid [3] tracks runtime data flow with high-level semantics on Dalvik virtual machine. Tracking at the Java object level can be circumvented by launching the malicious behaviors in native code execution. To address this issue, DroidScope [16] provides machine-level taint tracking for better code coverage. By running on top of an ARM emulator, it can analyze not only Java code but also native code thereby extending the scope of observation. However, the machine-level

Received December 26, 2015; revised October 31, 2016; accepted November 11, 2016.
Communicated by Hung-Min Sun.
* This work is supported in part by Taiwan's Ministry of Science and Technology, the National Security Council of Taiwan, the Telecom Technology Center, Taiwan's Ministry of Justice Investigation Bureau, Chunghwa Telecomm, the Taiwan Information Security Center at NCTU, and Taiwan's Ministry of Education.

taint tracking conducts semantics gaps such as process status, kernel objects, and network packets, and to resolve this information from machine level, containing only the values of registers and memory, costs much computing power and implementation effort. To maintain better coverage and the high-level semantics simultaneously, DROIT [14] proposes a dual-level taint tracking that dynamically switches the tracking layers. It can use either object-level or machine-level tracking according to the executing instruction.

Conventional work aims to design taint propagation rules corresponding to the executed Dex bytecode or ARM instructions. They barely discuss the tracking on peripheral devices, and therefore do not support taint tracking on mobile storage. Storage access, mostly the file modification, is a common program behavior for inter-process communication and data keeping. Conventional work perceives only the files being accessed but missing the data sources of the modified bytes. Without the file composition, analysts cannot distinguish the bytes of sensitive data from that of harmless data if they are in the same file. This will cause a false positive in their system when bytes of harmless data are leaked.

The goal of this paper is to uncover the data composition of the accessed files. Some implementation challenges lie ahead of this work. First of all, the sematic gap between memory, and the secondary storage (SD cards, NAND FLASH) should be addressed [7, 8]. The tracking system does not know the correlation between the physical objects (the bytes) and the logical objects (the files). Thus, we modify a file system parser, *unyaffs* [20] that converts byte offsets into files to resolve the gaps. Moreover, the taint sources should be dynamically selected because of the large number of files in storage. Conventional tracking systems need to decide data of interest beforehand. This requirement hinders themselves from performing either accurately or efficiently tracking when the selected data types are insufficient or excessive, respectively. To eliminate this restriction, we propose *dynamic taint source selection* that marks files when they are accessed by the subject through the out-of-box hooking technique [13]. Similar to traditional system hooks, the out-of-box hooking can intercept some specific functions through matching up constant code patterns of these functions. For example, if analysts want to hooks the function *fread*(), the code comparison with the pre-defined code pattern of *fread*() will be triggered every time code cache update, thereby stopping to execute the injected code. The injected code can acquire some runtime information extracted from the parameters of the hooked program points.

This paper presents DROIT+, a fine-grain tracking system that is able to learn the information about the data composition. DROIT+ is based on the proposed taint tracking system [14, 15] for reducing implementation effort on taint tracking. We propose a byte-granularity storage tracking method, and integrate the method into DROIT. The contributions of this paper are three-folds.

- We propose a byte-granularity storage tracking which dynamically chooses taint sources at runtime for efficiency and accuracy. The out-of-box hooking intercepts the execution of processes, kernel, and emulated devices to obtain high-level semantics of files. Once the data of interest are marked, we leave the taint propagation to our previous work.
- We make attempt to prove both the soundness and completeness of our method. The file access on Android has been formally modeled in this paper.

- We implement our method and integrate into our previous work to provide the taint tracking among storage resources, including memory, SD cards, NAND flash, and network. DROIT+ gives a boarder profiling scope than the previous. The file related behaviors are valuable to address security issues privacy leakage, for example.

The rest of this paper is organized as follows. Section 2 shows the methodology and gives the formalism proofs. Section 3 represents our system design and implementation. Section 4 performs the real world sample analysis, and evaluates DROIT+ by benchmarks. Section 5 briefly introduces the related work. The last section concludes our work.

2. METHODOLOGY

The background knowledge relevant to file access will be formally described in this section. Our formal models can illustrate the process of data access from function call, for example *fread*(), to hard drive I/O operations. Each hard drive I/O operation may trigger the propagation of the *taint tags*. Each tag is of distinct types of data source. A byte with a non-zero value taint tag is *tainted*. The taint tags are kept in a bitmap in order corresponding to the addresses in storage. In this paper, DROIT+ aims to reveal the file composition, so each bit stands for an individual file.

Most of taint systems are built on top of emulators, mostly QEMU [1]. The program execution on emulator may make a series of storage state transitions. The transition occurs when a *store* or a *load* instruction is executed. Such instructions cause state transitions of storage. Storage in the emulator includes memory, NAND Flash, and SD cards. A state transition stands for a variation of a basic data unit. A basic data unit is indexed by a physical address, and is usually part of file, from a logical perspective, residing in secondary storage. Furthermore, file systems have various management algorithms appropriate to the heterogeneous hardware material. The data units of a logical object are usually stored discretely. For looking up a position of a file, a conversion between a logical address (in program) into a physical address (in hard drives) is necessary while only hardware information is observable at machine level. In the following, we will model the emulator, the address conversion, and the file operations. The model will be used to prove the soundness and completeness of our design.

Android emulator Android emulator M splits an execution environment into two conceptual systems: *guest system* and *host system*. The guest system is emulated, and is restricted in the emulated resources. The entire guest system is just a process running upon the host system. With the pure software emulated hardware, the host system can emulate various *instruction set architecture* (ISA) such as x86, x64, and ARM. Emulator-based virtual machines (VMs) have the advantage over virtualization-based VMs in terms of architecture compatibility. Currently, the Android emulators are based on QEMU, an emulation-based VM for ARM system on x86 machines.

An emulator M can be formally described as follows. M contains emulated storage devices including the emulated *memory*, *internal storage*, and *external storage*, and it also contains an *emulated processor*, which loads instructions from the emulated memory. For emulating the execution, M sets its program counter to a memory address,

which fulfills the code for emulating an instruction. When finished, *M* returns to load the next instruction, and repeats the above procedure.

Data flow within Android emulator The execution on *M* is a sequential state transition mainly based on the data of the storage *S* (data flow) and executed instructions *I* (control flow). The logical data flow df_i corresponds the executed instruction i_i can be generally denoted as Eq. (1).

$$df_i = \begin{cases} SRC^* \overset{len}{\Rightarrow} DST^*, & i_i \text{ has a data flow} \\ \varnothing, & \text{otherwise} \end{cases} \tag{1}$$

The *SRC* is the source object, and the *DST* is the destination object. Both of them symbolize the storage regions from the logical perspective. The double arrow appends a data length *len*, and it shows the direction of the data flow. The star notion * indicates that the objects is logical. We consider the logical items such as files and memory buffers are accessible by pointers. A byte in such items can be indexed by an offset from the start address. The logical objects are eventually transformed into physical objects, which is of the storage units of hard drives.

The storage *S* includes primary storage *PS* (memory), internal storage *IS* (NAND Flash), and external storage *ES* (SD cards). During emulation, each executed instruction creates a data flow df_i. df_i can be classified into two types: *mem-mem* and *mem-disk*. The former is memory to memory data flow, and the latter is memory to disk data flow. Note that the term *disk* is not only meaning the magnetic disk. In Linux kernel, *disk* is a general term for secondary storage, thereby including *IS* and *ES*. The mem-mem tracking has been studied for years, so our paper will focus on the latter data flow. The byte-granularity tracking across storage devices has not yet been formally discussed. Before proceeding into the modeling, we briefly model the mem-mem taint tracking system DROIT.

Mem-mem taint tracking system DROIT initially determines which data should be taint source. Next, the tracking system propagates taint tags to show the tainted data relevant to the taint source. A general taint tracking system can be formally depicted as follows.

1. Given an Android emulator *M* containing memory, NAND Flash, and SD cards that represent primary storage (*PS*), internal storage (*IS*), and external storage (*ES*), respectively.
2. During the execution of *M*, *M* makes storage transitions that depend on the control flow consisting of ARM instructions from the boot loader, the kernel, and applications.
3. A taint engine *T* based on the emulator is a 5-tuple $T = (M, BM, \delta^T, C, K)$, where *M* is the emulator; *BM* is a bitmap for taint tracking; δ^T is a set of rule for taint propagation; *C* is a set of taint sources; and *K* is a set of taint sinks.
4. *BM* indicates whether a byte is *tainted* or *untainted* by 1 or 0, respectively, with a taint tag.

5. The size of the taint tag is adjustable. The more bits are used, the more information for its composition is kept. For example, using three bits as file identifiers, we can figure out which files of the three are involved with the creation through examining the corresponding bit status. A 32-bits taint tag is applied to maintain at most 32 distinct data sources in the tracking system.
6. The δ^T defines the rules how the tainted bytes propagate the taint tags to other bytes after execution. In general, the δ^T depends on the ISA, so each type of ISAs would have its own propagation rules.
7. The δ^T, a propagation rule on DROIT, can be classified into three propagation types *assign*, *append*, and *hybrid*.
8. The *assign* type is used for data movement such as *mov* that taints the output if the input is tainted.
9. The *append* type is used for bitwise operations such as *or* that taints the output if one of the input operands is tainted.
10. The *hybrid* type is used for arithmetic operations such as *mul* that combines above two situations to a byte-granularity taint tracking. Each byte is handled separately.
11. The set of taint sources C marks the data regions in memory as the propagation origins. These data as the input operands will further *taint* other memory regions depending on δ^T.
12. The taint sink K is a set of the terminal points of tainted data. Analysts can make decision when the tainted data flow to the taint sink. A taint sink may be a memory region. Once the tainted data flows to K, the tracking system will halt to perform actions for specific purposes such as privacy protection (by erasing data), sensitive leakage detection (by informing users), and malware analysis (by recording the system variation).

The tracking system T inspects the data flow instruction-by-instruction. It constructs the byte-to-byte correlation among registers and memory, so the mem-mem tracking can be achieved.

Lemma 1: Let δ^T is a set of propagation rules. It can be determined according to the given instruction set.

Proof: Each architecture has its own instruction specification that shows which bytes will vary after execution. Much research has proposed their own propagation rules by modeling the instruction sets.

Lemma 2: Given a single instruction, each byte of output can be either "*assign*" or "*append*" type propagated from the bytes of the operands.

Proof: The "assign" type is more straightforward. While the value of a byte is assigned to the one of another byte, the taint tag of the former shall be consistent to the latter. If the value of a byte is assigned to an immediate value, its taint status will be cleared since it is no longer relevant to the data source. The "append" type is used when the value of the output byte comes from the input operands. The taint status of the output will be derived from the inclusive OR between the input operands. Like the bitwise "OR" operator,

the output should be tainted if one of the inputs is tainted.

Lemma 3: Each data flow of an instruction i, namely df_i, the taint type can be one of the three types: *assign*, *append*, and *hybrid*.

Proof: Each register contains four bytes regarded as a basic unit for computing. Considering three operand architecture, two as input and one as output, the taint type of an output byte is determined by its input bytes where can be either "*assign*" or "*append*." If the output register simultaneously has both type, we call this instruction is a "*hybrid*" operation that decide the taint propagation according to the op-code. The propagation rule among bytes can be further defined in δ^T.

Lemma 4: The information flow df_i is byte-granularity addressable.

Proof: After an instruction execution, every output byte reveals its data sources by its taint status. The taint status is maintained by the propagation rule δ^T.

Lemma 5: Let $T = (M, BM, \delta^T, C, K)$ is a 5-tuple tracking system creating a series of data flow df_i, and T can track information flow within memory with byte-granularity.

Proof: When the M is running, it continuously generates df_i. The taint tags of every byte are kept in BM. After execution, T maintains the data sources of each tainted bytes.

Theorem 6: Given a proper δ^T, the taint sink K is tainted if and only if there is a data flow that moves data from taint source C into taint sink K.

Proof: Note that the C and K are memory regions. We firstly prove the "if" statement. If a program exists a data flow from C to K in program expressions, the program further be translated into a sequence of instructions, producing data flows that realize the dependency between inputs and outputs. Thus, the memory region K will be eventually tainted after the execution of the program. For the "only if" part, C is the only place marked as tainted, and these taint tags adhere to the propagation rules. If C has no data dependency with K, there is no means to taint K.

The above theorems depict the ability of tracking system T, and the soundness and completeness of its byte-granularity tracking ability. Next, we will further discuss the file operation, and simplify operations of the logical objects.

File operations In general, users manipulate regular files by *list*, *create*, *read*, *write*, and *delete*. The *list*, *create* and *delete* can be deemed as the *read* or *write* operations to directories, which a directory is also a file. Thus, only two operations, *read* and *write*, shall be considered. Due to the POSIX file system implementation, Linux provides a uniform means to access file, and the function prototypes are declared as follows.

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Both functions return a type *size_t* value that counts how many bytes are processed, and it is a negative number when errors occur. The *fd* is a file descriptor that contains the information of the opening files and the current positions of the file pointers. The *buf* is a pointer of a memory region for data saving. The third parameter, *count*, is the expected data size of operations. The two functions of file access can be represented by the data flow as follows.

$$\begin{cases} read : buf^* \overset{size}{\Rightarrow} FILE(fid,\ offset)^* \\ write : buf^* \overset{size}{\Rightarrow} FILE(fid,\ offset)^* \end{cases} \tag{2}$$

Given *fid* is a unique identity in OS. Usually, it is an *i* node number that can be looked up in the directory trees maintained by file systems. The *offset* indicates the logical position for operations. Both *fid* and *offset* are kept in file descriptors. The file descriptors are created when the file opens. The *buf* is a memory pointer, pointing to an available memory region from logical perspective. So far, these data objects are logical, not yet transformed to the physical objects in hard drives. Next, we will show the address transformation between the logical symbols and the physical bytes.

Theorem 7: An in-memory file modification can be tracked with byte-granularity if the memory regions of the file content are marked as a unique taint source.

***Proof*:** The kernel modifies file content by substituting the copies in memory. Memory modification is performed by instructions. If all file pieces within memory are marked with *C*, the file composition of a byte can be uncovered by examining the taint tag. The byte-granularity property for file processing is guaranteed by Lemma 4, and the tracking ability is guaranteed by Theorem 6.

Currently, the modeling shows that file operations is traceable within memory. The relation between the emulated storage devices and memory is not discussed yet. In the next part, the formalism is given to represent the address conversion between the logical and the physical objects.

File systems Android devices are mainly based on the two file-systems, YAFFS2 for internal storage and FAT for external storage. File systems divide files into many blocks as well as the sectors in hard disks. We use *blocks* and *sectors* for distinguishing the *logical* and the *physical* storage units, respectively. A block is a logically addressable, and it can be programmatically modified. A sector is a physically addressable, and the medium of sectors actually save data. For performance concern, the device updates a sector instead of a byte once at a time. Since a sector is the basic unit of hard drives, a block of file systems must contain a multiple of sector size. The kernel and file systems commit I/O operation consisting of blocks. The data order inside a single block is as the same as the one inside the storage devices.

There are three kinds of storage including primary storage *PS* (memory), internal storage *IS* (NAND Flash), and external storage *ES* (SD cards), are accessible in Android emulator *M*. The *i*th byte of these storage are denoted as ps_i, is_i, and es_i, respectively,

where i is physical address on hardware. A sector $s_i^{\{IS|ES\}}$ points to the specific address is_i or es_i of sectors, where $i = j \times SIZE(s_i^{\{IS|ES\}})$, $i, j \in Z^+$. The function $SIZE(\cdot)$ returns the size of objects, and the superscript $\{IS|ES\}$ shows which storage type of the data. The storage types can be simplified by a symbol t, and we omit the description $t = \{IS|ES\}$ in the rest of the paper. The sectors $<s_j^t, s_{j+1}^t, \ldots, s_{j+k}^t>$ can form a block blk_i^{PS}, where $k = \left(\frac{SIZE(blk_i^{PS})}{SIZE(s_j^t)} - 1\right)$. The data flow between them can be represented by the following form.

$$\begin{cases} \text{read a block from disk} : blk_i^{PS} \overset{size(blk_i^{PS})}{\Leftarrow} < s_j^t, s_{j+1}^t, ..., s_{j+k}^t \\ \text{write a block into disk} : blk_i^{PS} \overset{size(blk_i^{PS})}{\Rightarrow} < s_j^t, s_{j+1}^t, ..., s_{j+k}^t \end{cases} \tag{3}$$

Suppose $PA(\cdot)$ returns the physical address of an pointer, the physical address of a block blk_i^{PS} is written as $PA(blk_i^{PS})$. If $w = PA(blk_i^{PS})$, the block begins at the wth byte of PS. In the right hand side of Eq. (3), the start address of $<s_j^t, s_{j+1}^t, \ldots, s_{j+k}^t>$ on hard drives is the same as that of s_j^t. The following equation holds.

$$PA(<s_j^t, s_{j+1}^t, \ldots, s_{j+k}^t>) = PA(s_j^t) = PA(\{es_u | is_u\}) = u \tag{4}$$

In Eq. (4), it shows a sequence of successive sectors is byte addressable by the first byte. To use byte-granularity representation, we refine Eq. (3) by Eq. (4), which replaces sectors and blocks by bytes. The two operations $read^t$ and $write^t$ are denoted as follows.

$$\begin{cases} \text{read}^t : blk_i^{PS} \overset{SIZE(blk_i^{PS})}{\Leftarrow} s_j^t \vdash ps_w \overset{SIZE(blk_i^{PS})}{\Leftarrow} \{es_u \mid is_u\} \\ \text{write}^t : blk_i^{PS} \overset{SIZE(blk_i^{PS})}{\Leftarrow} s_j^t \vdash ps_w \overset{SIZE(blk_i^{PS})}{\Rightarrow} \{es_u \mid is_u\} \end{cases} \tag{5}$$

The kernel enqueues the I/O request after modifying files in memory. The I/O requests contain the information of the physical addresses of the sectors. The physical addresses are managed by file systems. Given a file x, all blocks of x is $B_x = \{blk_{x,0}, blk_{x,2}, \ldots, blk_{x,m-1}\}$, where m is the total number of the occupied blocks. Through dividing the file offset by the block size, file systems can know which block is under modification. The physical address of a logical block $blk_{x,v}$ can be calculated by the following function.

$$\begin{aligned} PA(blk_{x,v}) &= PA(< s_j^t, s_{j+1}^t, ..., s_{j+k}^t >) = PA(\{es_u \mid is_u\}), \\ &\text{where } v = \left\lceil \frac{offset}{(k+1) * SIZE(s_j^t)} \right\rceil, j = FS(f_x, offset), \\ &u = j * SIZE(s_j^t). \end{aligned} \tag{6}$$

To permanently store data, the kernel eventually writes the block $blk_{x,v}$ into the sectors $<s_j^t, \ldots, s_{j+k}^t>$ through file systems. These blocks can be converted into physical addresse $\{es_u|is_u\}$ on hard drives. The $FS(\cdot)$ function lookups the sectors by parameters f_x and $offset$, where the f_x indicates the file system type, and the $offset$ indicates the sectors responsible for data keeping. Eq. (6) is the key address conversion between logical ob-

jects and physical objects. The created data flow df_i of each file operation is byte-addressable on both internal and external storage since all the variables are known, including f_x, *offset*, and the size of sectors and blocks. The proposed taint tracking system can map the tainted bytes into memory.

Note that, from Eqs. (5) and (6), objects of both sides are physically addressable, and can be converted to each other. The byte-addressable conversion allows the tracking system to provide a byte-granularity tracking among storage.

Theorem 8: All dataflow between $\{ES|IS\}$ and memory is traceable in tracking system $\boldsymbol{T}$ if the algorithm of $FS(\cdot)$ and the location of I/O buffers are known.

Proof: If a dataflow df_i is related to storage access, by Eqs. (3)-(6), the taint status of the memory blk_i^{PS} can be synchronized (*assign* type) with the storage unit $\{es_u|is_u\}$, where the index u can be computed by Eq. (6). The byte indexing of FAT and YAFFS2 are existing in file system tools, so the physical address of $\{es_u|is_u\}$ can be revealed along with the file name and offset. For each read/write operation to disk, we can hook the functions of file operations for retrieving the addresses of I/O buffers, and hook the functions of device I/O execution for retrieving the addresses of the addresses of sectors. These addresses construct the byte correlation between the memory and secondary storage. Consequently, such data flow between memory and disks is with byte-granularity. By Theorem 7, mem-mem tracking is with byte-granularity, so the storage tracking will be achieved by synchronizing the taint status of the data units.

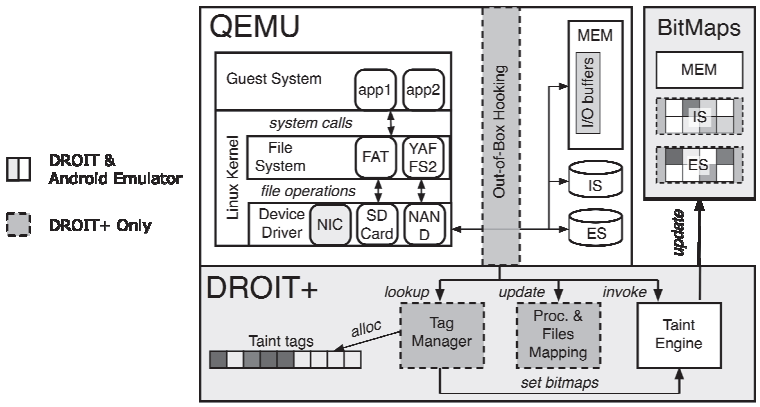


Fig. 1. The architecture of DROIT+. The light gray and white blocks belong to the previous work. We add some features to achieve the storage tracking. Two additional bitmaps for *IS* and *ES* are joined into the tracking system.

Once the correlation between physical and logical addresses is known, the taint status of secondary storage can be faithfully delivered into memory for further access. In the other hand, the buffers in memory update the taint status of disks when the files are flushed into disks. The data in disks will be tainted *if and only if* the creation of content involves tainted bytes.

In this section, we have formally introduced the emulation-based taint tracking sys-

tems and our methodology. The mem-disk tracking is similar to the mem-mem taint tracking, but with the address conversion. The ability of storage tracking is proven by Theorem 8. Next, we will discuss the implementation of DROIT+, showing how to extract the location of I/O buffers.

3. IMPLEMENTATION

The goal of our work is to perceive the composition of tracked data that the subject application ever accessed. We observed all data modification must be performed via memory operation and device I/O. It contains mem-mem and mem-disk data movements. The mem-mem tracking is achieved by the previous work, DROIT [14], with byte-granularity. The byte-granularity mem-disk tracking is not yet proposed. In this paper, the mem-disk tracking has been implemented to extend the tracking scope of DROIT.

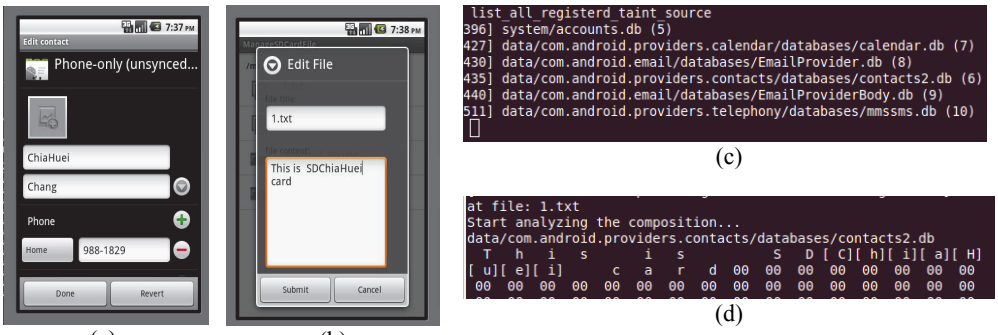
Maintaining the bitmaps of heterogeneous storage is challenging. First, we need to know how Android stores files. If a file is opened by the subject, the sectors of the file shall be tainted. DROIT+ enumerates the sectors of a file to setup taint tags. For this sake, we herein leverage the existing file system tools, *unyaffs* [20]. The rest of storage tracking is introduced in the following. Second, the size of taint tags is fixed, usually restricted in 32-bit taint tags. An intuitive means for storage tracking is to mark all the files in storage with unique identifiers at prior. A 32-bit tag is insufficient to distinguish hundreds of files in hard drives. Consequently, the proposed system should make good use of the limited taint source. To address this, we draw support from out-of-box hooking [13] at file operation functions, thereby acquiring opened files information and accomplishing dynamic taint tag allocation. The details are further described. Third, the network packets use port I/O, so it is hard to obtain the memory regions of packets. We also employ out-of-box hooking technique for retrieving the memory address for network traffics.

Dynamic taint tag allocation Due to the limit of taint tag size, DROIT+ allocates a taint bit as identifiers of individual files when they are opened. File opening starts from the system call, `sys_open()`, for checking the file presence. This system call further invokes the `do_filp_open()` in `/fs/namei.c` to create the `struct file` data, which contains more hardware information. The code pattern of `do_filp_open()` can be used for setting breakpoints. Since the binary code of the kernel does not vary, during emulation, we can halt to retrieve the file information in the stack when the code pattern is recognized. The procedure above mentioned is called “out-of-box hooking.”

A mapping between taint tags and files is necessary for dynamically marking the files as taint source when they are opened by the subject. DROIT+ always matches the current process name with the package name of the subject. If matched, DROIT+ regards the opened file as the data of interest, thereby marking it as a taint source. Moreover, the system databases including accounts, contacts, calendar, SMS, and browsing history contain much sensitive data, but the data of these databases cannot be obtained by file operations. The data can be accessed only via inter-component communication (ICC). Thus, hooking on `sys_open()` will cause misses for these sensitive data. To address the missing flow, DROIT+ marks the sectors of these databases in advance.

Taint propagation between the memory and the secondary storage In Android, persistent data are stored in either *IS* or *ES* with different access characteristics. Each installed application has its own directory in *IS*, and the directory cannot be accessed by other applications. *IS* stores system and application configurations in various data format, such as XML and SQLite databases. These files are only accessible by the owner and the *root* user. On the contrary, external storage can be arbitrarily accessed by the applications granted with the permissions `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE`. Mostly, applications access storage by file operations, and files are conceptual objects. At machine level, identify the accessed files is hard since the device I/Os do not contain the name of files, instead, only the sector numbers. We employ the out-of-box hooking to gather the file information.

By the out-of-box hooking technique, DROIT+ can know the addresses of the I/O buffers in memory. The kernel invokes the `do_sync_read()` and `do_sync_write()` for manipulating the I/O requests. When the subject reads a file, DROIT+ firstly allocate a tag for identifying the file. Next, DROIT+ use *unyaffs*, a file system tool, to locates all the sectors of the file, namely the function *FS*(.) in Eq. (6), and then marks them with the tag in the bitmap. To intercepting the device I/O of *IS*, DROIT+ hooks the `nand_dev_read_file()` and `nand_dev_write_file()` for extracting the addresses of the I/O buffers. The code is in `/external/qemu/hw/goldfish_nand.c`. For *ES*, the emulation code is written in `/external/qemu/hw/goldfish_mmc.c`. While the addresses of I/O buffers are known, the taint tags in bitmaps can be synchronized between memory and secondary storage. Because the mem-mem taint tracking is done by the previous work, the taint status of the memory buffer for writing to the secondary storage shall be correctly set. With the hooks in emulated devices, DROIT+ propagates the taint tags into *IS* and *ES* through updating the bitmaps of the secondary storage. The tags are permanently stored, so they will be successively propagated by the processes accessing the tainted data. DROIT+ therefore achieves a whole-system storage tracking through correlating the system-level and machine-level semantics.



(a) (b) (c) (d)

Fig. 2. The result of storage tracking; (a) is a pop-up window of contacts; (b) is the GUI of the simple file manager. We pasted the data of contacts and then saved the file as 1.txt; (c) shows the contacts are tainted with #6; (d) shows the tainted data in 1.txt is with byte-granularity property.

Network communication tainting In Linux, device I/Os are regarded as file operations, except the network communication. The emulated network interface controller (NIC) receives and sends packets via port I/O. Port I/O directly transmits data to hardware, and therefore no need to use memory buffer. Thus, the memory addresses of packets are hard to know. Observe that the network communication still invokes the system call `sys_read()`. DROIT+ intercepts all `sys_read()` for searching for the `SOCKET` data types. Once the current pid is matched with the one of the subject, DROIT+ will assign the taint tag of network into the bitmaps of memory.

4. EVALUATION

In this section, we will evaluate DROIT+'s the ability of storage access tracking, the ability of network communication tracking, and its performance. The evaluations are performed on a PC equipping an Intel i7-4770 3.4 GHz Quad-Core Processor and 8GB DDR RAM. The apps are tested on Android SDK 2.3 and 4.4. The first experiment is a proof of concept of byte-level tracking among storage. The crafted file manager can create, delete, and modify files in both IS and ES. Secondly, two representative malware are analyzed for evaluating the ability of network tracking. Finally, the performance experiment shows DROIT+ has around 30~40% performance downgrade for storage tracking.

Storage Tracking Most of system data and configurations are stored in *IS* in the directory "`/data/data/*`". At system booting stage, Android system initially opened the contacts databases for further use. DROIT+ detected the file opening and marked these data as tainted with tag #6, shown in Fig. 2 (c). After the data were tainted, we copied a piece of contacts and pasted them into the file in *ES*. This operation caused two data flows: *IS*-to-memory and memory-to-*ES*, shown in Figs. 2 (a) and (b), respectively. This experiment requires the tracking ability across the two file systems, FAT and YAFFS2. In Fig. 2 (d), the result showed the characters from contacts were correctly tainted with byte-granularity. The file composition of `1.txt` included the `contacts2.db`.

Malware Analysis Two malware are analyzed in this experiment. Both of them need to connect to web servers for proceeding malicious behaviors. Unfortunately, the servers were no longer existing so the malware would do nothing unless it connected to the servers. To pretend the servers were alive, we employed DNS query redirection and crafted a HTTP server to interact with the malware. The redirection intercepted all query packets from the malware, and returned the IP of the crafted HTTP server if the domain name was non-existent. When connecting, the HTTP server sent a random page for triggering the malware behaviors.

In the Table 1, the result is listed along with the captured sensitive data. The upper three columns are the package name of Android apps, requested the permissions, and sent data types. The row right after the description records the leaked data with byte-granularity. Both of them leaked IMEI, which is protected by the permission READ_PHONE_STATE, to the Internet. The IMEI number is 260310IMEI00000, and it can be found in the packets.

Table 1. The performance evaluation result.

	AuTuTu (points)							
	CPUI		CPUF		RAM		DB I/O	
	μ	σ	μ	σ	μ	σ	μ	σ
(a)	96.66	1.969	7.58	0.51	46.16	0.93	64.75	1.91
(b)	140.41	2.34	9.83	0.71	77.16	1.80	101.00	4.11
(a)/(b)	68.8%		77.1%		59.8%		64.1%	

	0xBench (points)											
	FFT		Comp.		MCI		SMM		LUMF		VMGC (ms)	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
(a)	0.75	0.025	1.17	0.037	0.341	0.01	1.393	0.67	1.393	0.05	28,275	961
(b)	1.14	0.043	1.72	0.038	0.576	0.02	1.990	0.72	2.072	0.09	17,206	671
(a)/(b)	65.8%		68.0%		59.2%		70.0%		67.2%		1.64 X	

Note: Two benchmark apps are applied in this experiment. It is estimated that DROIT+ maintains 60%~77% performance of the previous work.

Table 2. The analysis result of the malware. Owing to the DNS redirection, the leaked data can be shown. The IMEI is tagged with #27.

Package name	Requested Permission	Sending
com.crazyapps.angry.birds.rio.unlocker	**INTERNET,** ACCESS_WIFI_STATE, READ_CONTACTS, **READ_PHONE_STATE**, READ_LOGS	IMEI(27)
[TAINTED PCACKET]: "applicationDetails":{"applicationId":"325842966#752469853","build":{"brand":"generic","device": "generic","manufacturer":"unknown","model":"generic","versionRelease":"2.2.3","versionSDKInt ":8},"deviceId":" **2(27) 6(27) 0(27) 3(27) 1(27) 0(27) I(27) M(27) E(27) I(27) 0(27) 0(27) 0(27) 0(27) 0(27)** ","displayMetrics":{"density":1.0,"densityDpi":160,"heightPixels":480,"scaledDensity": …		
com.allen.txthej	**INTERNET**, **READ_PHONE_STATE**, ACCESS_NETWORK_STATE, WRITE_EXTERNAL_STORAGE, …	IMSI(26) IMEI(27)
[TAINTED PCACKET]: GET /activate/v1/?bd=&ei=**2(27) 6(27) 0(27) 3(27) 1(27) 0(27) I(27) M(27) E(27) I(27) 0(27) 0(27) 0(27) 0(27) 0(27) &si= 3(26) 1(26) 0(26) 2(26) 6(26) 0(26) I(26) M(26) S(26) I(26) 0(26) 0(26) 0(26) 0(26) 0(26)** &pack=com.galeapp.ebookshop &vs=1.2&vn=4&chn=1&src=3&sig= **e(26, 27) 2(26, 27) 3(26, 27) 7(26, 27) 6(26, 27) 5(26, 27)** d4 **4(26, 27) f(26, 27) d(26, 27) b(26, 27) 9(26, 27) b(26, 27) b 3 8(26, 27) 4(26, 27) b(26, 27) 7(26, 27) 6(26, 27) d(26, 27) b 5 2(26, 27) e(26, 27) 4(26, 27) 4(26, 27) a(26, 27) e(26, 27)** 54 HTTP/1.1 User-Agent: Dalvik/1.2.0 (Linux; U; Android 2.2.3; generic Build/MASTER) 0x0d 0x0a Host: effects.youmi.net 0x0d 0x0a Connection: Keep-Alive 0x0d 0x0a		

Performance Evaluation We applied two benchmark tools for performance evaluation. The first one is AnTuTu Benchmark which tests the performance of integer computation, floating point computation, memory access, and database I/O speed. In the Table 1, we use CPUI, CPUF, RAM, and DBI/O for short, respectively. Each test is repeated 20 times to acquire the average value μ and the standard variation σ. The results are the performance scores of the tracking systems; the higher, the better. Another tool, *0xbench*, integrates some well-known algorithm such as Fast Fourier Transform (FFT), Composite (Comp.), Monte Carlo Integration (MCI), Spares Matrix Multiply (SMM), LU Matrix Factoring (LUMF), and Dalvik VM Garbage Collection (VMGC). The unit of these tests is Mflops (Millions of floating point operations per second), except the VMGC. The result of VMGC is counted by milliseconds. The 1.64X overhead roughly equals to 62.5%, in other words, 37.5% performance downgrade.

Configuration (b) sets the baseline for the comparison. The result of our system is

measured in (a), which can track the dataflow among storage and memory. Maintenance of taint tags when data movement is significant overhead for emulation. The sixth row shows that our system maintains 59%~70% performance of the DROIT. The performance downgrade is due to the additional computation for storage tainting. As aforementioned, we use out-of-box hooking at file operations such as `nand_dev_read_file()` and `nand_dev_write_file()`. For every file accessing, the injected code will be executed whichever file is data of interests. The extra computation causes about 30~40% overhead according to the number of file operations.

5. RELATED WORK

Previous work [2, 4, 5, 18, 19] focus on the information flow within memory for program analysis. Static information flow tracking (SIFT) analyzed DEX byte-codes [5] or native binaries [6, 9, 11] to find potential privacy leakage. On the contrary, dynamic information flow tracking (DIFT) [3, 14, 16] can reveal more complex behaviors of an application without relying on the specific patterns of the source code of the subjects. It is meaningful to understand the run-time behaviors of apps [2, 12, 18], especially the obfuscated and the encrypted malware. Taint tracking provides abundant data flow information so that they can monitor data related to security issues.

Taint tracking at object level provides more program semantics, but can be circumvented if the malware is composed in native code. Meanwhile, tracking at machine level avoids the evasion of analysis, but it leads to obscure semantics reconstruction. DROIT [14] exploits the advantage of the two designs to provide a dual-level taint tracking. By inspecting the executed code, it can dynamically switch to the two levels on demand. Although DROIT can effectively and efficiently show the data flow of whole system, without the byte-granularity file tracking, it leads to the false positives that an entire file is tainted even if only a few tainted bytes are written into the file. To reduce these false positives, we improve the DROIT for a better storage tracking. The summary comparison of related work is shown in Table 3.

6. CONCLUSION

Taint tracking with byte-granularity is useful to extract the whole picture of a malware program, thereby uncovering its malicious behavior. In this paper, we present DROIT+, a taint tracking system to reveal data flow due to file access and network communication. File modification normally involves user-level, kernel-level, and hardware-level operations. To track data flow, we first model the file access and show the methodology of file tracking. Secondly, the dynamic tag allocation and out-of-box hooking are applied. Dynamic tag allocation facilitates the automation of setting taint tags. The out-of-box hooking is used to gather the system-level information of a subject where the hooking on the emulated hard drives can inform DROIT+ to synchronize the tags between memory and secondary storage as the device I/O is proceeding. On the other hand, using DNS redirection triggers network communication even if target hosts are unavailable. The redirection is effective to stimulate a quarantined malware program

Table 3. The summary of related work.

	Tracking Type		Granularity			Platform		Features					
Name	Static Information Taint	Dynamic Information Taint	Byte-granularity Support	Object-granularity Support	Symbol-granularity Support	x86 / x64	Android	Malware Detection	Leakage Detection	Decouple Taint Tracking	Storage Taint Tracking	Dynamic Taint Source Selection	Semantic Gap Resolving
Scandroid [4]	✓				✓		✓	✓		N/A	N/A	N/A	N/A
AndroidLeaks [5]	✓				✓		✓		✓	N/A	N/A	N/A	N/A
DroidSafe [6]	✓				✓		✓			N/A	N/A	N/A	N/A
DidFail [9]	✓			✓			✓		✓	N/A	N/A	N/A	N/A
Epicc [11]	✓			✓			✓		✓	N/A	N/A	N/A	N/A
Crowdroid [2]		✓		✓			✓	✓		N/A	N/A	N/A	✓
Paranoid [12]		✓		✓			✓			N/A	N/A	N/A	✓
TISSA [18]		✓		✓			✓		✓	N/A	N/A	N/A	✓
TaintDroid [3]		✓		✓			✓		✓		△		✓
TaintEraser [19]		✓	✓			✓			✓		✓		✓
DroidScope [16]		✓	✓			✓	✓		✓		✓		✓
DROIT [14]		✓	✓			✓	✓		✓	✓	△		△
SWIFT [15]		✓	✓			✓			✓	✓	✓		✓
DROIT+		✓	✓			✓	✓		✓	✓	✓	✓	✓

Note: The first column group identifies the type of analysis technique. The second column group shows the unit of analysis target. The third column group shows the target platforms. Finally, the forth column group identifies more specific features of each system. The four rightmost features are only applicable in taint tracking technique. The triangle mark stands for "partial." TaintDroid cannot provide byte-granularity for storage taint tracking, and DROIT needs declaring taint source in advance.

invoking a connection to a remote server. DROIT+ has been evaluated with formalism, case studies, and performance benchmarks. In our experiment, DROIT+ is able to uncover the file composition with 40% performance downgrade. This overhead may seem significant, but is reasonable for the dynamic testing of a malicious software program. Our work extends the scope of taint tracking, and provides more accurate program analysis.

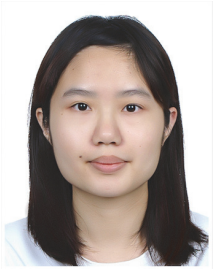
REFERENCES

1. F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of Annual Conference on USENIX Annual Technical Conference*, 2005, p. 41.
2. I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2011, pp. 15-26.
3. W. Enck, P. Gilbert, S. Han, V. Tendulkar, 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 USENIX Conference on Operating Systems Design and Implementation*, 2010, pp. 393-407.
4. A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android applications," *Manuscript*, University of Maryland, http://www.cs.umd.edu/avik/projects/scandroidascaa, Vol. 2, 2009.
5. 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*, 2012, pp. 291-307.
6. M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information-flow analysis of Android applications in DroidSafe," in *Proceedings of Network and Distributed System Security Symposium*, 2015.
7. X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007, pp. 128-138.
8. X. Jiang and X. Wang, "'Out-of-the-box' Monitoring of VM-based high-interaction honeypots," in *Proceedings of Recent Advances in Intrusion Detection*, 2007, pp. 198-218.
9. W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, 2014, pp. 1-6.
10. C. Mann and A. Starostin, "A framework for static detection of privacy leaks in Android applications," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, 2012, pp. 1457-1462.
11. D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. 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*, 2013, pp. 543-558.
12. G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid Android: versatile protection for smartphones," in *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010, pp. 347-356.
13. C. W. Wang, C. K. Chen, C. W. Wang, and S. W. Shieh, "MrKIP: Rootkit recognition with kernel function invocation pattern," *Journal of Information Science and Engineering*, Vol. 31, 2015, pp. 455-473.
14. C. W. Wang and S. W. Shieh, "DROIT: Dynamic alternation of dual-level tainting for malware analysis," *Journal of Information Science and Engineering*, Vol. 31, 2015, pp. 111-129.
15. C. W. Wang and S. W. Shieh, "SWIFT: Decoupled system-wide information flow tracking and its optimizations," *Journal of Information Science and Engineering*, Vol. 31, 2015, pp. 1413-1429.
16. 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*, 2012, p. 29.
17. 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*, 2007, pp. 116-127.

18. Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on Android)," in *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, 2011, pp. 93-107.
19. D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "TaintEraser: Protecting sensitive data leaks using application-level taint tracking," *ACM SIGOPS Operating Systems Review*, Vol. 45, 2011, pp. 142-154.
20. "ehlers/unyaffs," *GitHub*, https://github.com/ehlers/unyaffs, 2015.

Chia-Wei Hsu (許家維) is a Ph.D. student of Department of Computer Science, National Chiao Tung Univeristy, Hsinchu, Taiwan. He has experience in virtual machine, malware analysis, cloud computing, system security, mobile security, wargame contest, and network security. His recent research interests include mobile security, Android, and operating system. Contact him at hsucw@cs.nctu.edu.tw.



Chia-Huei Chang (張佳惠) is backend IT engineer with Taiwan Semiconductor Manufacturing Company (TSMC), Taiwan. She received an MS in Computer Science from National Chiao Tung University, Hsinchu, Taiwan.

Chi-Wei Wang (王繼偉) received his Ph.D. in the Department of Computer Science and Information Engineering, National Chiao Tung University, Taiwan. He has been very active in the malicious software analysis community, and has received many awards. Recently, he led his team and achieved 1st place in the Wargame Contest held by Hacks in Taiwan Conference 2010 and 2011. He also received the 1st place in the Microsoft Cross-Strait Innovation Contest in 2007. His research interests include network security, software security, and operating systems.

Shiuhpyng Winston Shieh (謝續平) is a Distinguished Professor and the past Chair of the Department of Computer Science, National Chiao Tung University (NCTU). He is actively involved in IEEE and has served as the Reliability Society VP Tech and EIC of IEEE Reliability. Shieh received his Ph.D. in Electrical and Computer Engineering from the University of Maryland, College Park, and invented (along with Virgil Gligor of CMU) the first US patent in the intrusion detection field. He is an IEEE Fellow and ACM Distinguished Scientist. His research interests include intrusion detection, network and system security, and malware behavior analysis. Contact him at ssp@cs.nctu.edu.tw.