ViaLin: Path-Aware Dynamic Taint Analysis for Android

Khaled Ahmed, Yingying Wang, Mieszko Lis, Julia Rubin
The University of British Columbia, Canada
{khaledea,wyingying,mieszko,mjulia}@ece.ubc.ca

ABSTRACT

Dynamic taint analysis – a program analysis technique that checks whether information flows between particular source and sink locations in the program, has numerous applications in security, program comprehension, and software testing. Specifically, in mobile software, taint analysis is often used to determine whether mobile apps contain stealthy behaviors that leak user-sensitive information to unauthorized third-party servers. While a number of dynamic taint analysis techniques for Android software have been recently proposed, none of them are able to report the complete information propagation path, only reporting flow endpoints, i.e., sources and sinks of the detected information flows. This design optimizes for runtime performance and allows the techniques to run efficiently on a mobile device. Yet, it impedes the applicability and usefulness of the techniques: an analyst using the tool would need to manually identify information propagation paths, e.g., to determine whether information was properly handled before being released, which is a challenging task in large real-world applications.

In this paper, we address this problem by proposing a dynamic taint analysis technique that reports accurate taint propagation paths. We implement it in a tool, ViaLin, and evaluate it on a set of existing benchmark applications and on 16 large Android applications from the Google Play store. Our evaluation shows that ViaLin accurately detects taint flow paths while running on a mobile device with a reasonable time and memory overhead.

CCS CONCEPTS

• **Software and its engineering**; • **Theory of computation → Program analysis**;

KEYWORDS

Dynamic taint analysis, path tracking, Android

ACM Reference Format:
Khaled Ahmed, Yingying Wang, Mieszko Lis, Julia Rubin. 2023. ViaLin: Path-Aware Dynamic Taint Analysis for Android. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616330

1 INTRODUCTION

Taint analysis [42, 43] is a type of information flow analysis technique that reasons about the propagation of sensitive data through the program execution. Taint analysis has many applications in security, e.g., for identifying information leakages, injection flaws, and cross-site scripting vulnerabilities [10, 17, 19, 23, 30, 46]. It is also applied in program understanding, software testing, and debugging activities [14, 29, 32].

Due to the increased popularity of mobile software, numerous static and dynamic taint analysis techniques for mobile applications (or apps, for short) were recently proposed. These techniques are used to facilitate different types of application analysis, including identification of injection vulnerabilities and information leakages [15, 24, 37]. Static taint analysis techniques [10, 21, 47] consider all possible paths that data can flow through without running the apps. Yet, recent studies [39, 53] show that existing static taint analysis tools do not scale to applications of realistic size and complexity. They also have difficulties analyzing several generic and Android-specific language constructs, such as reflection, dynamic code loading, and Android callbacks and framework methods.

Dynamic taint analysis techniques have access to such runtime information and thus became a practical alternative to static analysis, despite their own disadvantages, such as the inability to reason about behaviors that were not triggered at runtime and the need to keep a low runtime overhead (Android simply kills slow apps). TaintDroid [19] is probably the first dynamic taint analysis technique implemented for Android; it modified the Dalvik Virtual Machine to accurately track information flows while keeping a low execution overhead. Subsequent work, such as Taint-ART [45], further improves the efficiency of the analysis, while also re-implementing it for the new ART runtime.

To keep the runtime overhead low, dynamic taint analysis techniques typically assume a one-bit taint mark for each tracked information *source*, such as device id, location, and sensitive input data. They propagate the marks through the execution of the program, raising an alert when marked data reaches a sensitive *sink*, such as an API that sends data out from the device. Marks are typically held in a 32-bit vector tag (i.e., one integer) attached to each program variable. Such design ensures that taint marks do not occupy too many processor registers, which would slow the execution down.

One limitation of this design is that it allows at most 32 different taint marks, with a non-linear performance degradation when increasing the number of tracked taints. An even more important limitation is that this design only allows the techniques to report flow endpoints, i.e., the sources and sinks involved in a flow. The techniques do not capture and cannot report the *taint path*, i.e., *how* the information propagates from sources to sinks. Finding such paths manually is a challenging yet necessary task, e.g., to check whether the information was properly handled before it was released. Our experiments show that taint paths extracted from 16 real

```
 1 class AppActivity {
 2   String info;
 3   String getUserInfo() {
 4     String input1 = getUserInput();  //source1
 5     String input2 = getUserInput();  //source2
 6     Location loc = getLocation();    //source3
 7     String size = input1.getLength();
 8     String lang = loc.getLanguage();
 9     String data = size + lang;
10     setInfo(data);
11     sendInfo();
12     Log.d(input2);
13  }
14   void setInfo(String str) {
15     this.info = str;
16   }
17   String getInfo() {
18     String val = this.info;
19     return val;
20   }
21   void sendInfo() {
22     String data = getInfo();
23     sendToInternet(data);               //sink
24 }}
25 void start(){
26   AppActivity a = new AppActivity();
27   a.getUserInfo();
28   a.<more code...>;
29 }
```

Figure 1: An Example Application.

Android applications contain more than 57 code-level statements, on average, and span more than 14 methods, on average.

Consider, for example, the simplified (pseudo-)code snippet in Figure 1. The method `getUserInfo()` (lines 3-13) is triggered when the app execution starts (lines 26-27), to collect information about the user and store it in a field called `info` (line 2). Specifically, the method reads two input values provided by the user (lines 4-5), e.g., the username and password specified in different textual input boxes, and the user location (line 6). However, instead of storing these values directly, it only extracts the size for the first input (line 7) and the language spoken at the current user location (line 8). It concatenates and stores these values in the `info` field (lines 9-10, 14-16) and sends the value of this field out to the internet (lines 11, 17-24). Furthermore, the method writes the value of the second user input, `input2`, into a log file stored on the device (line 12).

While existing dynamic taint analysis techniques will correctly report that there is a flow of sensitive information from the user and location data into the sink in line 23, they do not report the exact flow path, making it difficult for an analyst to inspect how the information flows between sources and sinks, e.g., to understand that only the size of the user input and the language spoken in a particular location were released and decide whether that constitutes a violation of user privacy. Moreover, to keep the size of the taint tag small, the techniques only track the type of information being released and do not distinguish between code statements in which the information is obtained. For example, they cannot determine which of the user-provided information is leaked, username or password, making it even harder to analyze the identified flows and/or build other types of analysis on top of these tools.

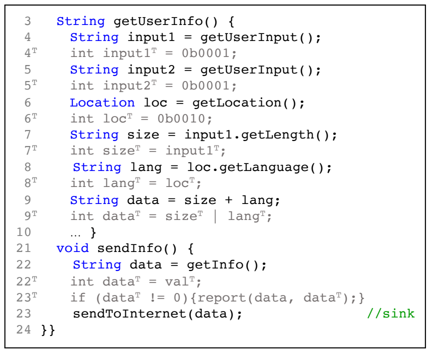
Other variations of dynamic information flow analysis, e.g., approaches based on dynamic forward slicing [35, 44], could, in principle, report path-level information. Yet, these existing techniques suffer from low efficiency when applied to large software, cannot effectively separate paths when simultaneously tracking information from multiple different sources, and are not designed to consider specifics of the Android platform (see Section 8 for more details).

To fill this gap, we propose the first dynamic taint analysis approach for Android that accurately tracks and reports detailed, statement-level taint path information without inducing a substantial application slowdown. We refer to our approach as ViaLin (for routing and linking source to sink information). Instead of attaching a taint mark to each information source and propagating these marks through data flows, as done by the classical taint analysis approaches, ViaLin keeps a lightweight data structure (up to five integer variables in most cases) that points to elements that hold information about previous propagation steps. That is, ViaLin treats taint propagation as a sequence of elements representing taint propagation steps. At a sink, it traverses this sequence to report both the propagation path and the source(s) involved in the paths. This design leads to a more efficient management of memory, allowing the operating system to garbage-collect taint marks that can no longer lead to any sink. It also allows ViaLin to track any number of taint sources without any implementation changes. Section 2 further illustrates of how ViaLin operates and how it compares with existing dynamic information flow analysis approaches.

We implemented ViaLin by extending the Android Runtime and evaluated it on a set of benchmarks and on 16 large Android applications from the Google Play app store. We compare the paths extracted by ViaLin to those extracted statically by FlowDroid for the benchmark apps and those extracted by a human expert for the Google Play apps. To evaluate the overhead of the path collection functionality introduced by ViaLin, we compare it to our own re-implementation of the "classical" dynamic taint analysis approach that allocates a one-bit mark for each tracked taint source, as was done in TaintDroid. We had to re-implement TaintDroid on top of our taint analysis infrastructure as the original TaintDroid implementation does not support the current Android architecture. We refer to our re-implementation as TD+.

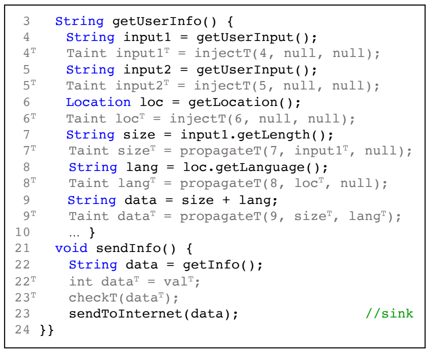
Our evaluation of ViaLin shows that it can accurately identify taint flows and their corresponding paths, with only a few false positive and false negative results. As with other taint analysis approaches, the false-positives are due to array, file, and parcel over-tainting while false-negatives are due to the lack of support for native code and implicit flows. Yet, our evaluation shows that paths reported by ViaLin can help reveal security vulnerabilities and malicious handling of user-sensitive information in large Android apps. The runtime and memory overhead induced by the tool in a typical use is 39.8% and 7.3%, respectively, which we believe is acceptable, given the extra functionality that it provides.

Contributions. This paper makes the following contributions:
1. It proposes a novel context-, field-, flow-, and path-sensitive dynamic taint analysis approach that captures the entire taint propagation path, not only its endpoints.
2. It implements the approach in a tool named ViaLin and empirically evaluates its accuracy and overhead on a set of benchmarks and on large Android applications from the Google Play store.
3. It shows the practical usefulness of ViaLin in identifying security vulnerabilities and malicious behaviors in large Android apps.
4. It makes our implementation of ViaLin, the "classical" dynamic taint analysis approach TD+ implemented on top of the same infrastructure, and all our experimental data publicly available to support future work in this area [8].



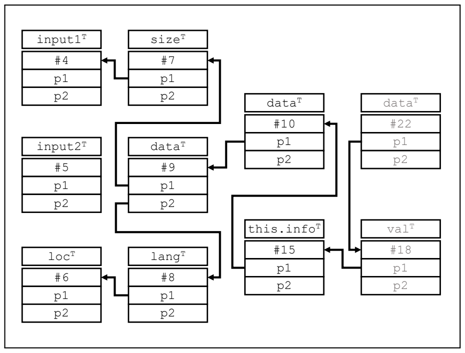
```
3    String getUserInfo() {
4      String input1 = getUserInput();
4^T    int input1^T = 0b0001;
5      String input2 = getUserInput();
5^T    int input2^T = 0b0001;
6      Location loc = getLocation();
6^T    int loc^T = 0b0010;
7      String size = input1.getLength();
7^T    int size^T = input1^T;
8      String lang = loc.getLanguage();
8^T    int lang^T = loc^T;
9      String data = size + lang;
9^T    int data^T = size^T | lang^T;
10     ... }
21   void sendInfo() {
22     String data = getInfo();
22^T   int data^T = val^T;
23^T   if (data^T != 0){report(data, data^T);}
23     sendToInternet(data);                 //sink
24 }}
```

(a) TaintDroid



```
3    String getUserInfo() {
4      String input1 = getUserInput();
4^T    Taint input1^T = injectT(4, null, null);
5      String input2 = getUserInput();
5^T    Taint input2^T = injectT(5, null, null);
6      Location loc = getLocation();
6^T    Taint loc^T = injectT(6, null, null);
7      String size = input1.getLength();
7^T    Taint size^T = propagateT(7, input1^T, null);
8      String lang = loc.getLanguage();
8^T    Taint lang^T = propagateT(8, loc^T, null);
9      String data = size + lang;
9^T    Taint data^T = propagateT(9, size^T, lang^T);
10     ... }
21   void sendInfo() {
22     String data = getInfo();
22^T   int data^T = val^T;
23^T   checkT(data^T);
23     sendToInternet(data);                 //sink
24 }}
```

(b) ViaLin



(c) Taint Structure Generated by ViaLin.

Figure 2: Taint Tracking for the Example in Figure 1.

2 VIALIN BY EXAMPLE

In this section, we use the example in Figure 1 to illustrate how ViaLin operates and how it compares with existing, “classical” dynamic analysis approaches. We leave a detailed discussion about the design choices made, our memory management optimization, and our handling of Android-specific constructs to later sections.

Figure 2a shows the implementation of the `getUserInfo()` and `sendInfo()` methods, augmented with the sketch of the code that tracks taint tags in the “classic” way, as implemented by TaintDroid. There are two types of taint sources in this example, which are tracked by TaintDroid using two taint marks: `0b0001` for the information coming from the user input and `0b0010` for the location data. To ease presentation, we show the marks in a 4-bit rather than 32-bit taint tag vector.

The taint tracking code is injected between the original code lines, i.e., line i^T (in grey) for each original line i. For example, in line 4^T, a new taint tag `input1`$^{\mathrm{T}}$ is created to track the taints of the variable `input1`. The tag is associated with its corresponding variable by name; it is initialized with the taint mark for the accessed information source – `0b0001`, in this case.

For each code instruction, taint marks are propagated from the used to the defined variables. For example, in line 9^T, the taint tag `data`$^{\mathrm{T}}$ tracking the taints of the variable `data` is assigned all taints of the variables `size` and `lang` by unifying their corresponding taint tags. At the sink, the taint tag of the released variables is checked and, if not empty, the tool raises an alert reporting the taint marks that reached the sink, as done in line 23^T. In this case, the tool just reports two taint sources: user input and location data.

Figure 2b shows the alternative instrumentation performed by ViaLin. Instead of tracking taint tags as bit vectors, it uses an object `Taint`, which contains three fields: an id of the code instruction that *defines* the tracked variable and two pointers, `p1` and `p2`, which point to the `Taint` objects tracking taints of variables *used* in the instruction. Figure 2c shows a schematic representation of the created graph structure, which allows ViaLin to report both the sources of the taint and the propagation path. Two pointers are sufficient in most cases as Java bytecode uses three-address instruction commands; exceptions to this format are discussed later in the paper.

More specifically, our instrumentation uses three operations: `injectT`, `propogateT`, and `checkT`. The `injectT` operation creates a new `Taint` object each time a taint source is accessed and associates it with the corresponding defined variable (see lines 4^T, 5^T, 6^T). The `propogateT` operation creates a new `Taint` object for the defined variable and propagates the taint from the used variables by setting the pointers `p1` and `p2` accordingly. For the example in line 9^T, a new `Taint` object `data`$^{\mathrm{T}}$ is created and the pointers `p1` and `p2` are set to point to `size`$^{\mathrm{T}}$ and `lang`$^{\mathrm{T}}$, which track the taints of `size` and `lang`.

At a sink, the `checkT` operation traverses the data structure starting from the released variables, to identify and report the sources involved in the information release, if any, and the propagation path (see line 23^T). The graph structure in Figure 2c illustrates this process: it shows that the taint for the `data` variable accessed in line 23 arrives from both the user input `input1` in line #4 (via the path 4⟶7⟶9⟶10⟶15⟶18⟶22⟶23) and the location source in line #6 (via the path 6⟶8⟶9⟶10⟶15⟶18⟶22⟶23). The graph also shows that `input2` in line #5 does not flow to any sink.

In the remainder of the paper, we give the necessary background information and describe our implementation of ViaLin. We then discuss our experimental setup and evaluation results.

3 BACKGROUND

The Android System. The Android operating system is a software stack architecture built on top of the Linux kernel. Android apps execute in the topmost architecture layer. They call APIs from the Java framework layer, which provides an abstraction for interfacing with the device hardware and sensors. The Android Runtime (ART) layer compiles apps and the Java framework code into native code using the dex2oat compiler; the native code is then executed by the ART. Apps and the Java framework code can also call native code directly through the Java Native Interface (JNI).

Each app is run in its isolated sandbox. Apps interact with one another through *parcels* – a message-passing construct that is serialized by the system and transported between apps. In-app communication, between different app processes, can also rely on parcels. Apps may also store files on the persistent storage of the system.

Android Memory Management. Any memory an app modifies, e.g., by allocating new objects, is managed by the ART in RAM. An app can release memory by releasing object references that it holds, making the memory available to the garbage collector. Garbage collection events are triggered by the ART and work in two phases.

The first phase marks all objects that can be accessed by the app in the future. It starts by collecting the roots: all in-scope variables, local variables on the stack, static variables, and active threads. It then traverses all reachable objects from the roots and marks them as visited. In the second phase, the garbage collector frees non-marked objects from memory, reclaiming the resources used by those objects. In Figure 1, if garbage collection is executed after line 27, it will use the variable `a` as the root. It will mark this variable and its field, `a.info`, as visited and release all other variables, e.g., `input1` and `input2` defined in method `getUserInfo()`, as they are no longer reachable in or after line 28.

Dalvik Bytecode. Android-native apps are written in either Java or Kotlin; they are compiled into Dalvik executable bytecode (dex), stored in an `apk` package. Each Dalvik bytecode method has a fixed-size stack frame that consists of a particular number of registers (specified by the method) as well as any adjunct data needed to execute the method, e.g., program counter. When used for bit values, such as integers and floating point numbers, registers are considered 32 bits wide. Adjacent register pairs are used for 64-bit values. When used for object references, registers hold exactly one such reference.

Dalvik employs three-address instructions; each instruction has a single opcode, e.g., addition, and up to three operand registers. An instruction may assign a value to at most one operand register and use the two other registers as sources. For example, r_3=r_1+r_2 is encoded as `add-int` r_3, r_1, r_2. Some instructions do not assign any operands, e.g., `if-eqz A, Offset`, which checks if the value in register `A` is zero and branches accordingly to the `Offset`.

An exception to the three-address instruction format is the `invoke` instruction for method calls: it uses up to five operands representing method parameters. For methods with more than five parameters, the `invoke/range` instruction uses two registers, r_1 and r_n, and treats all registers between r_1 and r_n as parameters, inclusive.

Even though the example in Figure 2b is written as Java-style code, it is actually close to the bytecode format and uses three-address instructions; the analysis described in this paper is performed on bytecode rather than Java source.

Statements and Statement Instances. For simplicity of presentation, we refer to a bytecode statement in line i as s_i i.e., saying that `input1` is defined in statement s_4. Each bytecode statement can be triggered multiple times during the app execution, e.g., in multiple iterations of a loop or in different calls to a method; we refer to each individual execution of a statement as a *statement instance* and denote the j^{th} execution of a statement s_i as s_i^j. For example, the first execution of s_4 is s_4^1.

Data-flow Analysis and Taint Paths. Data-flow analysis is the process of collecting information about the way variables are defined and used in the program. For each program statement, we say that the assigned operand is *defined* in the statement and the source operands, if any, are *used* in the statement. We say that there exists a *data flow* from the statement instance s_i^j to the s_k^m through variable v if and only if s_i^j assigns a value to v and s_k^m uses this value. For example, there is a data flow from s_4^1 to s_7^1 through the variable `input1`.

A *data flow graph* is a graph whose nodes are statement instances and edges are data flows between statement instances (not necessarily over the same variables). There might be multiple data flow paths (which we refer to as *routes*) between a pair of statement instances, e.g., due to dependencies induced by multiple variables.

In taint analysis, statements defining sensitive data, e.g., user input, location, are marked as information *sources*; protected statements, where the information should not propagate, e.g., sent to the internet, are defined as information *sinks*. A taint analysis technique is then applied to identify data flow graphs between pairs of taint source and sink instances.

In our example, $s_4^1 \rightarrow s_7^1 \rightarrow s_9^1 \rightarrow s_{10}^1 \rightarrow s_{15}^1 \rightarrow s_{18}^1 \rightarrow s_{22}^1 \rightarrow s_{23}^1$ is a taint flow graph between s_4^1 and s_{23}^1. In this case, it only contains a single route. For simplicity and usability, it can be reported at a statement level, without including information about statement instances, as in the example in Section 2.

4 APPROACH

At a high-level, ViaLin is implemented as an instrumentation pass in the Android Open Source Project, which is conducted at app installation time. ViaLin obtains as input application bytecode and a list of source and sink APIs. It then injects the taint tracking instrumentation, which:
1. Identifies statements that use any of the source APIs and injects taints at these statements (e.g., line 4^T in Figure 2b);
2. Injects the taint propagation logic to track taints in assignments, method calls, native method calls, files, and parcels (e.g., line 9^T);
3. Identifies statements that use any of the sink APIs and injects taint checking and reporting logic (e.g., line 23^T).

The instrumented code is further compiled by the `dex2oat` compiler and executed in ART. Unlike other approaches that inject the instrumentation in the dex2oat compiler optimization phases [13, 45], our approach of instrumenting ahead of dex2oat allows the compiler to optimize the inserted taint propagation logic in tandem with the original app code. Furthermore, we apply taint propagation instrumentation to the Java framework bytecode, which is compiled and optimized when building the system, not at runtime.

When a statement instance containing a sink is about to execute, ViaLin checks if tainted information reached the sink from at least one of the sources, and if so, reports taint propagation paths from all sources reaching the sink.

We now discuss each of these steps in more detail. We start by describing our approach for storing taint tags in memory.

4.1 Taint Tags

We allocate a taint variable (a.k.a. taint tag) to track taint for each source statement. This taint variable is an instance of the `Taint` class, described in Section 2. Then, for each tracked Java construct, such as a local variable, field, instance field, method parameter and return variable, file, and parcel, we allocate another `Taint` instance, if required by the taint propagation rules discussed below. We define a taint map function T, which maps the tracked Java construct to its corresponding taint tag.

The scope of the allocated taint tag is the same as the scope of the tracked object. That is, for **local variables** the taint variable is allocated in the scope of the method, doubling the number of registers allocated in the method stack frame. For the example in Figure 2b, `loc`^{T} is the taint tag associated with the `loc` variable (line 6); its scope is the same as the scope of the tracked object, i.e., the `getUserInfo` method.

While techniques such as TaintDroid [19] interleave the taint tags with the original registers for spatial locality, this requires a modification to the Dalvik virtual machine instructions. For example, an instruction that reads a long variable from register r_1 expects the second part to be in r_2. Interleaving taint tag references with the original variables means that such instruction has to be modified to read the second part from r_3 instead of r_2. Instead, we opted to add all the taint tag reference registers after all the original registers in the stack frame. We rely on the dex2oat compiler and optimizer to correctly allocate original and taint tag Dalvik registers to reduce register spilling and expensive memory accesses. That is, a taint tag of a local variable allocated at register r_n within a method with a total of m registers, is r_{n+m}. The left-hand side of Figure 3 shows the taint tag allocation for the method `setInfo` of Figure 1: the first two registers are allocated for the receiver `this` and the variable passed as a parameter `str`. The next two registers are allocated for the taint tag references of `this` and `str`, respectively.

Taint variables for **fields** are defined as additional fields of the class and have the same scope as the tracked field. Likewise, static fields, which are shared by all instances of the class, are allocated an extra static field in the class' metadata to store a reference to the taint tag of the field, and, thus, it is also shared by all instances of the class. For example, the right-hand side of Figure 3 shows the allocation of the $\texttt{info}^T$ taint tag mapped to the `info` field in an instance of `AppActivity`.

For scalability, we allocate a taint tag for the entire **array**, rather than tainting each entry of the array separately. This choice may cause false positives but offers lower space and performance overhead [19]. An array's taint tag is, again, defined as the same scope as the original array.

Parameter passing in Dalvik is done through registers; when the stack frame of the called method is created, the parameters are copied into registers in the newly created stack frame. One approach to pass taint tags with parameters is to modify all method signatures to include the taint tags as parameters and modify the Dalvik calling convention to insert copies of taint tag references into the stack frame. However, this approach breaks reflection [52] since method signatures for a reflective call target may not be available at the time of instrumentation. Thus, we follow the approach by TaintMan [52], adding a global taint tag array, stored as an instance field in the `Thread` class (thus, each thread gets its own separate array). At the caller, taint tags of method parameters are placed in order in the array; the callee copies the taint tag references from the array and places them in local registers. Similarly, the taint tag reference of the returned variable is placed in the array by the callee and is copied by the caller, making the analysis performed by ViaLin context-sensitive.

The center part of Figure 3 shows the taint tag array for the thread executing the code in Figure 1. The first cell in the array is reserved for the method return or thrown exception and the following cells store passed parameters in their order.

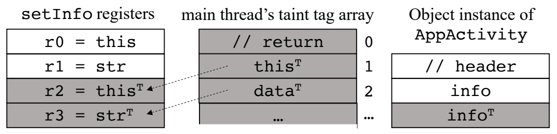


Figure 3: Taint tag storage (shaded) for local variables (left), parameter passing and returns (center), and fields (right).

Files are stored in the system's persistent storage rather than within an app. To track taints for files, we create a taint tag file with the same path as the tainted file but with an extra extension (.taint). The taint tag file stores a serialized version of the entire taint data flow graph arriving at the file. We modified the file copying Java APIs to copy the (.taint) file when propagating taints. However, our implementation does not support cases when files are copied using shell commands or native code.

Parcels are serialized prior to transmission. Thus, a parcel's scope may change after it is serialized, e.g., from the scope of one activity class to another. We modified the parcel class in the framework to allocate an extra field for the taint tag of the parcel, which gets serialized with the parcel data and de-serialized at the new scope of the parcel.

4.2 Taint Injection and Propagation

The taint injection function, `injectT`, simply allocates a new taint tag for each variable defined at a statement instance that invokes a source API. To propagate taints, the `propagateT` function first checks if at least one of the used registers in a statement instance is tainted, i.e., its corresponding taint tag is not null. If so, the function creates a new taint tag and points to the taint tags of the used registers.

Detailed taint propagation rules for assignments, method calls, returns, and exceptions are defined in our online appendix [8]. For example, for statements that **assign** a constant to a register, `propagateT` clears the taint tag of the register (by assigning it the null value). For assignments with one used register, it checks whether the taint tag of the used register is not null and, if so, creates a new taint tag with `p1` pointing to the taint tag of the used register and `p2` being null. Assignments with two used variables, returns, and exceptions, are handled similarly. Method calls are treated as a sequence of parameter copies, mapping each callee formal parameter to the corresponding variable in the caller function.

Field assignments do not propagate the taint to the entire object that contains the field, thus ensuring our analysis is field-sensitive. However, instructions that read instance fields propagate the taint tag not only from the field itself but also from its enclosing object since the object could be tainted but its fields may not be, e.g., by being assigned the return of a source statement. In this case, all sub-fields of the object are considered tainted.

For inserting variables into **arrays**, we propagate the taint from two registers: the register of the inserted variable and the array itself as it could already be tainted by a previous insertion. When retrieving variables from an array, we propagate the taint from the array to the variable's register.

Each **parcel** and **file** can be tainted by numerous taints: a parcel can carry several tainted variables within its internal array and multiple taint sources can be written to a file. As our propagation rules can propagate taints from a maximum of two variables, we treat each write to a file or a parcel in a similar way we treat arrays: we propagate both the prior taint of the entire file/parcel and the taint of the written variable, accumulating the taints of all variables written to the file or parcel.

ViaLin injects the taint propagation code into the Dalvik bytecode and thus does not track taint propagation in **native code**. Instead, we model framework native calls using FlowDroid's taint wrappers [10]: a set of rules that define how taint is propagated within a method. For example, for the method call `ret=a.f1(b,c)`, a rule will indicate that the taint only propagates from the input parameter `c` to the return value, but not to the receiver object `a`. We thus treat this call in a similar way to treat an assignment `ret=c`. We inject the taint propagation logic at the call site of each modeled native call. For methods not covered by FlowDroid's taint wrappers, e.g., custom methods defined in the app, we use the *generation* taint wrapper strategy, conservatively propagating the taint from all arguments and the receiver to the receiver and the return values. For example, for the method call `ret=a.f2(b,c)`, is treated as the set of assignments `a=b`, `a=c`, `ret=a`.

4.3 Taint Checking

The taint check function, `checkT`, is injected at every statement instance that invokes a sink API. It obtains as input taint tags of all parameters of the sink. For each non-null tag, i.e., when the corresponding parameter is tainted, `checkT` uses depth-first search to traverse all reachable taint tags, building the data flow graph between the source and sink statement instance pair.

As discussed in Section 3 and also by Arzt [9], there may be more than one route between a given source and sink statement instance pair in a data flow graph. Reporting such a graph in full, at a statement instance level, does not add any additional value for the human analyst and might even overwhelm the analyst due to an exponential number of paths caused by loops, repeated methods calls, etc. Therefore, similarly to FlowDroid [9, 10], we chose not to report the full data flow graph in ViaLin, though it can be explicitly enabled, if needed. Instead, we report all statements of the graph in the order of their execution. We refer to this report as a *taint path*. The evaluation section contains a more detailed discussion on the practical implication of this decision.

4.4 Memory Management

ViaLin allocates taint tags in the scope of the tracked variable. However, these tags are not freed by the garbage collector when the variable gets out of scope, as long as they are reachable from taint tags of other, in-scope variables. This ensures that taint propagation paths are reachable even if variables on the path are garbage collected. The data structure is automatically pruned once the tracked taint propagation path can never reach a sink.

For the example in Figure 1, if a garbage collection event is triggered after line 27, all local variables of methods other than `start` are garbage collected. However, the taint tags of all but the variables in lines 18 and 22 remain in memory. That is because the variable `a` and its `info` field are still live objects and thus, the taint tag of the `info` field is still in scope. This tag points to the tag of an object in line 10, which points to the tag of an object in line 9, etc., as shown in Figure 2c. The remaining tags are garbage collected after the execution of the `start` method terminates.

5 EVALUATION METHODOLOGY

In our evaluation, we aim to investigate the accuracy and performance of ViaLin by answering the following research questions:

RQ1 (Accuracy) How accurate is ViaLin in identifying
RQ1.1 (Detection) taint flows?
RQ1.2 (Path Construction) path information?

RQ2 (Performance) What is the impact of ViaLin on
RQ2.1 (Time Overhead) app execution time?
RQ2.2 (Memory Overhead) consumed memory?

RQ3 (Utility) How useful is ViaLin in practice?

5.1 Subject Applications

We evaluated ViaLin on three sets of applications. The full list of applications that we used, information about their size, the list of expected and identified paths, and other relevant information are available in our online appendix [8].

DroidICCBench. To answer **RQ1**, we used popular benchmarks created specifically for evaluating the effectiveness of taint-analysis tools for Android: DroidBench v3.0 [2, 10] and ICC-Bench [3, 47]. These benchmarks, combined, consist of 217 apps. Even though the benchmarks were developed mostly to challenge static taint analysis tools, many of the cases are also relevant to dynamic analysis, e.g., array over-tainting, sensitivities, and more. We had to exclude eight apps for which we cannot reliably trigger the flows in an automated way, e.g., applications for which a taint flow is triggered only when the phone is running out of memory (from the `onLowMemory` callback). Moreover, as the benchmark apps were developed for an older Android API (level 19), where permissions to run any sensitive API are given at installation time, we modified the apps to request permissions using the approach of the newer Android versions. In the end, we used 209 apps in our evaluation, and we refer to them as *DroidICCBench*.

Sources and sinks. A set of sources and sinks for each application in DroidICCBench is defined in the application header file. In total, there are 5 sources and 11 sinks used across different benchmarks. To increase automation, we configured ViaLin to use this combined list of sources and sinks for all apps, as was done in prior work [53]. We also used the updated set of expected flows, which corresponds to this combined set of sources and sinks. Finally, we manually analyzed each benchmark application to extract the path, at statement level, corresponding to each flow.

GPBench. To further evaluate ViaLin in a more realistic setup, and answer both **RQ1** and **RQ2**, we used a benchmark of Google Play applications from Zhang et al. [53]. To construct this benchmark, the authors systematically collected 19 large popular apps with login functionality from the Google Play store. They deem the benchmark representative as the average dex size of apps in their benchmark is 15 MB while their reported average size of 5,500+ top apps in Google Play is 11.6 MB. The authors manually identified at least one login-related flow in each app, and reported sources, sinks, and paths of these flows. They further used the benchmark to evaluate the accuracy of static taint analysis tools, manually analyzed all flows reported by the tools, and augmented the expected results with the identified true-positive flows. Yet, this set of manually marked flows is still partial: there could be additional flows not found manually or by the static analysis tools.

We excluded from our study three out of the 19 apps, as their backend servers were non-functional at the time of writing and we thus could not execute them dynamically. We used the remaining set of 16 apps, which we refer to as *GPBench*, for our evaluation. As manually identifying all expected flows in large real apps of GPBench is a very challenging task, we used the set of flows marked by the authors of the benchmark, together with their established expected paths, as a *partial* expected result for ViaLin.

The only other publicly available benchmark with real apps and marked flow paths we are aware of is TaintBench [6]. However, we could not use it for our evaluation as the benchmark contains old malware samples whose servers were taken down and is thus not suitable for dynamic analysis.

Sources and sinks. We configured ViaLin to use two sets of sources and sinks for the GPBench apps. The first, which we refer to as the *short list*, was defined by Zhang et al. [53] for each individual app, e.g., `EditText.getText()` for obtaining the password and `HttpURLConnection.getResponseCode()` for sending it to the internet. We use this configuration to assess the accuracy of the tool.

Then, to check scalability and further stress the tool, we configured ViaLin to run on the same set of apps using a larger set of taint sources. Specifically, we augmented our short list of sources with an additional list of taint sources used by FlowDroid [10]. The combined list includes 97 sources, out of which 23 are executed in at least one app and we adopted them for our analysis. We refer to this list as the *long list* of sources and sinks.

Fake WhatsApp client. To help answer **RQ3**, we additionally browsed blog posts of major security companies, identifying posts that (a) describe information-stealing Android malware and (b) provide clear characteristics allowing us to identify the described malicious apks. As a result of this search, we identified a mod WhatsApp client called YoWhatsApp [5]: a repackaged WhatsApp client app that steals a user authentication key.

Sources and sinks. To identify the stealthy behavior of the app, we configured ViaLin to use the key generation API, `Curve25519Provider.generatePrivateKey`, as a source and all APIs sending information to the internet as sinks.

5.2 Evaluation Methodology and Baseline Tools

To answer **RQ1.1 (Detection Accuracy)**, we run ViaLin on: (1) DroidICCBench applications, to compare the reported to the expected results. (2) GPBench-SL, denoting GPBench applications run with the short list of sources and sinks. We use the results of these runs to check whether the tool can detect the expected flows which were manually marked by the GPBench authors. We also inspect the number of reports produced by the tools, the number of distinct paths it reports, and the properties of these paths, such as their length and the number of different routes between source and sink statements. (3) GPBench-LL, denoting GPBench applications ran with the long list of sources and sinks. We use the results of these runs to further inspect the quality of the produced paths.

To answer **RQ1.2 (Path Construction)**, for the apps in DroidICCBench, we compare paths produced by a static taint analysis tool FlowDroid to paths produced by ViaLin. As FlowDroid does not scale to analyze apps in the GPBench dataset [53], for the GPBench-SL app, we compare the paths reported by ViaLin to those manually identified by the authors of GPBench.

For **RQ2.1 (Time Overhead)** and **RQ2.2 (Memory Overhead)**, we run ViaLin on GPBench applications: first under the GPBench-SL configuration that represents the "typical" use and, to further stress it, on the GPBench-LL configuration. Furthermore, we used our taint analysis infrastructure to re-implement the "classic" one-bit mark taint analysis approach, as discussed in Section 2, which we refer to as TD+. To fairly compare the results of both tools, we run TD+ with the same configuration setup and execution script. While the overhead of ViaLin is expected to be higher than that of TD+ (as TD+ does not collect/report any path info), this baseline comparison is performed to estimate the magnitude of the increase. Due to possible differences in execution time and memory footprint between different runs, we execute each experiment ten times on each system and report the average reported scores of all runs.

We do not compare ViaLin with other dynamic taint analysis tools for Android, e.g., TaintART, because our proposed idea is orthogonal to that of other tools, which focus on improving the efficiency of one-bit taint mark tracking, not on reporting taint propagation paths. Moreover, the implementations of these tools are either unavailable or major parts are missing. We also do not compare ViaLin with dynamic taint analysis techniques outside of the Android domain as we are not aware of any technique that provides full statement-level path tracking.

For **RQ3**, we manually inspected the paths of the GPBench apps that we analyzed, identifying vulnerabilities related to handling of user credentials. In addition, we recruited two third-party developers with more than two years of experience with security-related tasks each. This experience includes reverse-engineering apps, security protocols, and cryptographic protocols; penetration testing; and participation in national and international cybersecurity competitions. We provided both experts with the malicious YoWhatsApp apk and the blog post describing its malicious behavior; we asked them to identify and describe the key stealing code of the app and to report on the time it took to perform the task. While expert #1 did not receive any additional information, we provided the taint path describing the information stealing flow to expert #2. As a follow up on the study, we also sent a questionnaire to expert #2, to investigate (a) whether they believe the provided taint path helped in completing the task; (b) the reason for the provided answer; and (c) any suggestions for improvement of the tool.

Execution Scripts and Environment. As ViaLin is a dynamic approach, we executed all apps in DroidICCBench and GPBench. We manually traversed each app to collect its execution traces. Flows in most of DroidICCBench apps are trivially invoked when an app starts; for GPBench, we ensured to cover the login functionality and traversed as much of the apps as possible afterwards. To ensure our analysis is fair and repeatable, we recorded an execution script for each app using the android-touch-record-replay tool [16] and replayed these scripts in all stages of our evaluation, to run the apps automatically. We perform all our experiments on the same Pixel 2 XL (taimen) device running Android API level 28, version 8.0.0. The analysis of YoWhatsApp was performed by the security experts manually, by reverse-engineering and inspecting the app.

6 RESULTS

6.1 RQ1.1 (Detection Accuracy)

Table 1 shows the results of ViaLin runs on the DroidICCBench applications. Due to space limitations, we grouped all apps from the same benchmark category into one row. For example, the first

Table 1: RQ1.1: Detection Accuracy on DroidICCBench.

ID	Category	# Apps	Expected	Reported	TP	FP	Precision	Recall	F-Measure
1	Aliasing	4	1	1	1	0	100	100	100
2	Android Specific	14	10	10	10	0	100	100	100
3	Arrays And Lists	10	4	9	4	5	44.4	100	61.5
4	Callbacks	9	9	9	9	0	100	100	100
5	Dynamic Loading	5	3	3	3	0	100	100	100
6	Emulator Detection	15	16	13	13	0	100	81.3	89.7
7	Field And Object Sens.	7	2	2	2	0	100	100	100
8	General Java	25	20	20	20	0	100	100	100
9	Implicit Flow	6	6	0	0	0	100	0	0
10	Inter-App Comm.	11	25	25	25	0	100	100	100
11	Inter-Comp. Comm.	18	26	26	26	0	100	100	100
12	Lifecycle	23	22	22	22	0	100	100	100
13	Native	5	5	0	0	0	100	0	0
14	Reflection	9	9	9	9	0	100	100	100
15	Reflection ICC	10	21	21	21	0	100	100	100
16	Self Modification	4	3	0	0	0	100	0	0
17	Threading	6	6	6	6	0	100	100	100
18	Unreachable Code	4	0	0	0	0	100	100	100
19	ICC Handling	12	13	13	13	0	100	100	100
20	ICC Target Finding	7	15	15	15	0	100	100	100
21	Mixed	1	2	2	2	0	100	100	100
22	RPC Handling	4	4	4	4	0	100	100	100
Total		209	222	210	205	5	97.6	92.3	94.9

row includes four apps from the "Aliasing" category. The number of expected flows in these four apps is 1 (column "Expected"), indicating that some apps are rather designed to "confuse" the analysis into producing false positive results.

As any dynamic approach, ViaLin generates a report every time tainted data reaches a sink statement instance (column "Reported"). We further count the number of unique statement-level paths that the tool reports (there could be multiple paths with different statements and/or statement instances between the same source and sink statement pair). As apps in DroidICCBench are simple by design, in this benchmark, each sink statement is executed only once and there is only one path between each source and sink pair. Thus, the number of paths ViaLin reports is equal to the overall number reports (in column "Reported").

The "TP" and "FP" columns of the table show the breakdown of reported flows to true positive and false positive results, respectively. ViaLin has a very low false positive rate (and, thus, high precision): out of the 210 reported flows, only five are false-positives, all in the "Arrays and Lists" category, due to the over-tainting of arrays, i.e., tainting the entire array instead of its individual cells – a common limitation of many taint analysis techniques.

There are also 17 false-negatives, 9 of which are due to the design choice not to support implicit flows (categories #9 and #16). Another 5 false-negatives are in the "Native" category. While ViaLin models some framework-related native calls, the DroidICCBench apps have either the taint source or the entire path implemented in custom native code, which ViaLin does not support. Finally, 3 false-negatives, in the "Emulator Detection" category, are due to the evaluation limitation: these apps check for the presence of Google Play services on the device, which we cannot install on top of the Android Open Source build used for our prototype.

Overall, ViaLin achieves a high recall of 92.3% and an overall F-Measure (a harmonic mean that balances precision and recall) of 94.9% on the DroidICCBench apps. This result is largely expected, given that the benchmark apps were mostly designed to confuse static rather than dynamic analysis techniques.

Table 2 shows similar information for GPBench-SL and GPBench-LL runs of the tool, i.e., runs with the short and long list of sources and sinks on the GPBench benchmark. As discussed in Section 5.1, we do not have the complete ground truth for these large apps, due to the sheer number of sources, sinks, and possible paths. Yet, our experiment confirms that ViaLin can detect all expected flows manually marked by the benchmark authors (columns "Marked Flows" and "Marked Flows Detected").

For the GPBench-SL, ViaLin produces additional 56 reports, bringing the number of total reports to 86 in this configuration. As in the case of DroidICCBench, all but two reports correspond to a unique reported path – information of the highest interest to the human analysis. In apps #3 and #12, the same sink statement is triggered twice, resulting in two identical statement-level paths.

To gather insights about the correctness of the reports, two of the paper authors independently sampled and inspected a subset of the reported paths. They were able to confirm the majority of the reports, observing only two false positive results, both in app #11, due to over-tainting of a hashmap – a data structure backed by an array. Such false positives were also observed in our DroidICCBench evaluation and are shared by most taint analysis tools.

Interestingly, some of the reported paths "connect" the same source and sink statements via different sequences of statements. Column "Source-Sink Pairs" reports on the number of *unique* pairs of source and sink statements, among all reported paths. For example, in app #8, there are only two such unique pairs. However, there are six types of information (email, password, first name, last name, company name, and address inputs), which flow to the internet via the two source-sink statement pairs.

Figure 4 is a simplified example of the code of this app. Its source statement (line 2) is located inside the `getValue(..)` method, which is triggered multiple times (lines 6, 7), with different widget ids, to obtain different types of user input (unlike in our motivating example in Figure 1, where the data from each widget was obtained explicitly). This causes multiple reported flows to have the same source and sink statements but different paths.

```
1  String getValue(int id) {
2    return (findViewById(i)).getText();//source
3  }
4  void onClick(){
5    JSONObject o = new JSONObject();
6    o.put("Email", getValue(R.id.email));
7    o.put("Password", getValue(R.id.pwd));
8    ...
9    sendToInternet(o);                    //sink
10 }
```

Figure 4: Multiple Paths.

ViaLin can identify and report all such cases. Yet, without these reports, when only using a dynamic tool that reports flow endpoints, it would be difficult for a human analyst to identify all involved paths manually, especially in large applications. Including more source statements in the analysis, as was done in GPBench-LL, further complicates matters: the difference between the number of reported paths and just the number of unique source-sink pairs becomes even more pronounced: 749-247=502.

Moreover, as shown in the table, the size of the paths averages at 57.2 statements (max: 509), with 14.9 methods on a path, on average (max: 156), spanning 7 different classes, on average (max: 70). ViaLin can help avoid the intensive manual labor involved in detecting these paths.

Answer to RQ1.1: ViaLin accurately detects taint flows with only a few false positives due to over-tainting of data structures and false negatives due to implicit flows and native code. The paths reported by ViaLin can help distinguish between different ways in which information can flow between the same source and sink statements.

Table 2: RQ1.1: Detection Accuracy on GPBench.

ID	Marked Flows [53]	GPBench-SL				GPBench-LL														
		Marked Flows Detected	Reports	Reported Paths	Source-Sink Pairs	Reports	Reported Paths	Source-Sink Pairs	Stmts on Path				Methods on Path				Classes on Path			
									Min.	Max.	Avg.	Med.	Min.	Max.	Avg.	Med.	Min.	Max.	Avg.	Med.
1	1	✓	5	5	3	29	16	11	7	105	28.9	23.5	1	25	5.3	4	1	7	2.6	2
2	1	✓	2	2	2	882	57	27	2	218	71.1	57	1	63	18.9	11	1	29	10.6	7
3	1	✓	2	1	1	7	7	4	5	69	25.5	26	1	9	5	4	1	5	3	3
4	1	✓	2	2	2	2	2	2	21	22	21.5	21.5	5	5	5	5	3	3	3	3
5	1	✓	4	4	3	17	16	5	10	212	71.5	67.5	3	30	12.4	9	2	13	5.6	5
6	1	✓	4	4	3	15	15	4	12	231	75.2	69	2	32	13.1	69	2	13	5.9	5
7	1	✓	4	4	3	7	7	3	2	95	57.2	69	1	22	10	9	1	10	4.3	3
8	1	✓	6	6	2	7	7	4	29	179	86.7	86	7	21	14	14	4	10	7	7
9	1	✓	12	12	4	1149	282	32	5	509	113.6	86.5	2	156	41	34.5	2	70	18	15
10	15	✓	21	21	21	30	30	29	10	67	34.9	22	4	22	10.6	7	4	13	6.7	4
11	1	✓	4	4	1	31	17	6	3	139	108.3	101	3	31	24.1	22	2	10	8.6	8
12	1	✓	4	3	3	4	3	3	4	5	4.7	5	2	3	2.7	3	2	3	2.7	3
13	1	✓	5	5	5	5	5	5	25	27	25.8	26	7	7	7	7	4	4	4	4
14	1	✓	2	2	1	148	88	16	2	108	31.4	32	1	40	11.5	11	1	19	5.7	6
15	1	✓	2	2	2	81	34	28	18	223	61.8	47	6	45	14.6	12	3	22	6.5	5
16	1	✓	7	7	7	280	163	68	5	206	97.9	109	4	72	33.2	33	2	35	17.6	17
Total	30	-	86	84	63	2694	749	247	-	-	-	-	-	-	-	-	-	-	-	-
Avg.	1.9	-	5.4	5.3	3.9	168.4	46.8	15.4	-	-	57.2	-	-	-	14.3	-	-	-	7	-

6.2 RQ1.2 (Path Construction Accuracy)

DroidICCBench. To evaluate the accuracy of path construction, we compare the paths reported by our tool to paths reported by FlowDroid – the only other tool that can produce taint propagation paths for Android apps, albeit statically. Out of the 205 true-positive paths reported by ViaLin on DroidICCBench, FlowDroid can identify and report 134. The remaining cases are challenging to analyze statically due to Java- and Android-specific constructs, such as reflection, inter-component communication, and more [40, 53]. We thus focused our comparison on the 134 paths found by both tools.

Our analysis, conducted by two authors of this paper, shows that ViaLin can correctly identify most of the statements in taint propagation paths. Out of 134 analyzed paths, 61 were identical and 73 were slightly different, i.e., contained extra or missing statements. We identified several reasons for the differences, described below.

By design, ViaLin does not report object aliasing statements on the path (55 aliasing statements overall, on 15 paths of 15 apps). Figure 5 shows an example: in line 3, `a.c` is tainted. Line 4 aliases `b` to `a` and, thus, the field `b.c` aliases the tainted field `a.c`. Then, `b.c` reaches the sink in line 5. ViaLin does not include the aliasing statement in its path, reporting only statements in lines 3 and 5, as it tracks the taint of the object field itself, not its containing object. FlowDroid reports the aliasing statements as it performs the analysis statically. Such reports could help the analyst to better understand how the taint reaches the sink.

```
  1 A a = new A();
  2 B B = new B();
  3 a.c = source();
- 4 b = a;
  5 sink(b.c);
```

Figure 5: Aliasing.

At the same time, we observed more than 120 extra statements on 55 paths reported by FlowDroid but not ViaLin. These are related to unneeded ‘return void’ in methods where tainted variables are assigned (even though the return statements do not propagate any tainted data), and extra method calls and lifecycle statements that do not propagate any taints.

Finally, FlowDroid only reports a single witness for each path between source and sink. That is, the tool aborts the path construction after the first route for a given pair of source and sink is found [9]. Due to that reason, FlowDroid paths contain only the shortest possible route between the source and sink pair, which causes it to miss statements on the pass. This occurred in three paths from three different apps, shown in our online appendix [8].

GPBench. As manually inspecting all paths reported by the tool is unfeasible, we focused on inspecting the 30 paths manually marked by the authors of GPBench. For these paths, we also observed cases of missing alias statements, as discussed for DroidICCBench. We also encounter cases where the reported path includes extra statements that do not propagate tainted data due to over-tainting arrays and data structures backed by arrays, inflating the paths in apps #1, #4, #10, and #11.

While inspecting the paths, we observed interesting cases where the reported path can help analysts identify whether the information was properly handled before being released. We discuss them in more detail in Section 6.4.

<u>**Answer to RQ1.2**</u>: ViaLin accurately reports the majority of path statements. It does not include aliasing statements and can include extra statements related to operations on array-backed data structures.

6.3 RQ2 (Performance)

To evaluate the performance of ViaLin, we run it with both the short and the long list of sources and sinks, i.e., for GPBench-SL and GPBench-LL. The first configuration intends to represent the “typical” use and the second – to stress the tool to run in a “heavy” scenario: as the apps in the GPBench-LL configuration have 1520 source and 55 sink statement instances, on average, this represents an upper bound of a reasonable scenario, where the analyst configures the tool with many possible sources/sinks at once. We compare the performance of the tool with the baseline execution of the app on an unmodified Android build and with the execution for our own re-implementation of one-bit taint tracking approach, TD+.

RQ2.1: Time Overhead. For each of the analyzed tools and configurations, we run each app ten times. We collect the total CPU execution time for each run from the `/proc/{pid}/stat` file and average the times of all runs. We then compute the time overhead w.r.t. the original execution, the overhead of ViaLin w.r.t. the original execution, and the overhead of ViaLin w.r.t. the TD+.

Table 3 shows the results of this experiment, for both GPBench-SL and GPBench-LL configurations. Columns “Max. Path Length” and “Source Stmt Instances” show the maximal length of the reported path for each app and the total number of source statement instances for an app, respectively. These are used as indication of “work” done by ViaLin: the number of source statement instances indicate the amount of taint injected into the system since each

Table 3: RQ2.1: Time Overhead.

ID	Orig	GPBench-SL					GPBench-LL						
		TD+		ViaLin			Max. Path Length	Source Stmt Instances	TD+		ViaLin		
	Time (s)	Time (s)	Overhead (%)	Time (s)	Overhead (%)	Overhead vs. TD+ (%)			Time (s)	Overhead (%)	Time (s)	Overhead (%)	Overhead vs. TD+ (%)
1	12.4	13.2	6.5	23.4	88.3	76.8	105	385	15.6	25.5	24.1	94.1	54.7
2	45.2	50.5	11.6	63.1	39.5	25	218	34	53.8	19	62.5	38.2	16.1
3	12.4	12.7	1.7	13	4.5	2.7	69	23	12.6	1.4	14.2	13.8	12.3
4	13.7	14.5	6.4	15.6	14.4	7.5	22	242	14.6	7.2	16.3	19.3	11.3
5	5.6	5.8	2.8	6.5	15.3	12.2	212	15	6	7.2	6.4	13.6	5.9
6	6.7	8	19.4	9	34.7	12.8	231	20	8.4	26.9	9.7	46.1	15.2
7	4.5	5	10.9	5.7	26.1	13.8	95	15	5	10.9	6.4	42	28
8	4.9	5.1	2.2	5.4	9.9	7.6	179	79	5.7	14.5	6	21.7	6.2
9	10.6	14.2	33.3	25.1	136.2	77.2	509	5528	16.5	55.3	40.2	278.4	143.7
10	9.3	10.2	10.3	11.6	25.2	13.4	67	17	10.3	11.5	11.5	23.8	11
11	25.3	27	6.6	28.5	12.8	5.7	139	13485	29	14.8	32.1	27.1	10.7
12	6.8	7.1	4	12.5	85	77.9	5	6	7.6	12.5	13	92	70.6
13	4.6	5	9.3	5.7	23.2	12.8	27	103	4.9	6.5	6	29.4	21.5
14	37.8	38.6	2.2	46.6	23.4	20.8	108	1188	40.4	6.8	43.3	14.8	7.4
15	15.9	20.9	31.3	29.4	84.7	40.7	223	318	21.2	33	40	151.7	89.2
16	6.6	6.7	1.1	7.5	13.8	12.6	206	2868	7.2	9.4	8.6	30.7	19.4
Med.	10	11.5	6.6	12.8	24.3	13.1	123.5	91	11.5	12	13.6	30.1	15.7
Avg.	13.9	15.3	10	19.3	39.8	26.2	150.9	1520.4	16.2	16.4	21.3	58.5	32.7

source statement instance produces at least one taint propagation path that may reach a sink. Likewise, the length of the taint propagation path is another indicator of the amount of taint propagation that was maintained during the runtime of the app.

As expected, TD+ maintains a relatively low time overhead in both setups: 10% and 16.4% on average for GPBench-SL and GPBench-LL, respectively. This result is comparable with that reported by the original TaintDroid implementation. The corresponding overhead of ViaLin over the original run is 39.8% in "typical" use and 58.5% in the "heavy" scenario. Interestingly, the median of the execution time overhead is 24.3% and 30.1% respectively, indicating that more than half the apps finish the execution in a reasonable time. The highest overhead, for both TD+ and ViaLin is in app #9, which has a high number of source statements instances (5528) and the longest reported path (509 statements).

RQ2.2: Memory Overhead. To evaluate the memory overhead of the tools, for each solution, we measure the Virtual Memory Resident Set Size (VmRSS) which indicates the resident physical memory. We sample VmRSS from the `/proc/{pid}/status` file every second, then average all samples for each run. We then further average the results among all runs. We compute the memory overhead compared with the baseline execution in a manner similar to that of the time overhead.

Our results show that the memory overhead induced by the tool is relatively low: 7.3% and 9.4% on average, in "typical" and "heavy" scenarios. This is largely due to the efficient memory management strategy employed by ViaLin which allows the Java garbage collector to clean up objects which are no longer in the execution scope. Due to space limitations, we include the full memory overhead table in the online appendix [8].

Remarkably, in our setup, we did not notice any slowdown running the apps on a real device. We thus consider the overhead induced by ViaLin acceptable, given the extra value it provides to the user compared with existing dynamic taint analysis techniques.

Answer to RQ2: On average, ViaLin increases the execution time by 39.8% and costs 7.3% more memory in a typical-use setup. This overhead is still acceptable as makes it possible to execute applications on the device without any notable slowdown. The overhead increases as the number of injected taints into the system increases.

6.4 RQ3 (Utility)

Analysis of GPBench apps. As discussed in Section 6.1, when manually inspecting the information flows from the user-entered password to the internet in the GPBench apps, we noted that paths reported by ViaLin can help identify security vulnerabilities, such as improper handling of sensitive data. Specifically, our analysis identified a vulnerable app which exhibited insecure programming practices: app #4 (10M+ downloads). In this app, a hardcoded JSON Web Token (JWT) secret key was used to sign the password before it is sent over the internet, which significantly increases the risk of compromising the user's privacy by an attacker [1, 4].

While this vulnerability was fixed in later versions of the app (starting from version 4.3.3, released in October 2021, where the app developers redesigned the authentication mechanism) it existed in all app versions between September 2015 and October 2021, including the version that we analyzed. Identifying flows of sensitive information in the app, as enabled by ViaLin, could encourage a dedicated review of the ways sensitive information is handled, helping to eliminate vulnerabilities sooner.

Analysis of YoWhatsApp. Two experts reverse-engineered and manually analyzed the malicious YoWhatsApp app, which steals a user authentication key, allowing the malicious developer to take control over victim WhatsApp accounts. Expert #1 performed the analysis without using any information provided by ViaLin while expert #2 received the taint path describing the information stealing flow, as generated by ViaLin. While building a proper controlled experiment with the same person (or two *identical* humans) analyzing the same app, with and without the help of the tool, is generally impossible, we believe the practitioners we recruited have similar expertise and thus their analysis times are comparable in this case.

The results of our experiment indicate that a path provided by ViaLin could indeed help cut the manual analysis time by around 40% in this case: it took the experts 15 and 9 hours, respectively, to identify and describe the malicious key-stealing behavior in this app. In a qualitative analysis, expert #2 believed that the provided path was helpful to understand "which functions are part of the malicious flow, to narrow down the amount of code that I analyze".

They also noted that the reported path could be extended by providing the calling context of methods, e.g., when a tainted variable is written in a callback A but is accessed in a callback B. Such context is needed to "better understand how and why a certain part of the code is called". Even though providing such information goes beyond standard data-flow-based taint paths (even if extended with implicit flows), we believe this is an interesting observation which could be addressed in future work, e.g., by augmenting statements in the reported taint paths with their corresponding stack trace dumps or backward slices from the reported statements.

Finally, after expert #1 completed their analysis and reported on the identified flow, we showed them the taint path detected by ViaLin. The expert confirmed that the path appears to accurately represent the flow of information they observed in the app and could have been helpful to speed up their manual analysis.

Answer to RQ3: Path-level information provided by ViaLin can help analysts identify security vulnerabilities and malicious behaviors in large real-world software.

7 LIMITATIONS AND THREATS TO VALIDITY

For **external validity**, our results may be affected by the subject apps' selection and may not necessarily generalize beyond our subjects. We attempted to mitigate this threat by using a set of apps available from related work without introducing investigator bias into the selection process. As we used both known benchmarks and Google Play apps of considerable size and complexity, we believe our results are reliable. For **internal validity**, we had to analyze the produced paths manually, to ensure their correctness. To mitigate possible bias related to this manual effort, two authors of this paper performed the analysis independently and cross-checked each other's results. We make our implementation and evaluation setup publicly available to encourage validation and replication of our results. The main **limitations of our approach**, shared with other taint analysis approaches, are the coarse-grained modeling of arrays, files, parcels, and native calls. ViaLin does not support file copying using shell commands or native code, does not support implicit flows (as they commonly lead to severe over-tainting, making the analysis impractical [18, 25]), and requires flashing the device to run. We plan to investigate ways to address some of these limitations, e.g., by borrowing approaches implemented in other tools [20, 25, 49, 50], as part of future work. Moreover, as ViaLin currently stores the file tainting artifacts in the user space, a malicious app equipped with knowledge of how the tool works could inspect and tamper with these files [12]. Developing anti-malware analysis techniques, e.g., to raise an alert when an app accesses a ".taint" file, could be another subject of possible future work.

8 DISCUSSION AND RELATED WORK

We now discuss closely-related information flow analysis techniques.

Dynamic Taint Analysis for Android. TaintDroid [18, 19] is probably the first and the most complete dynamic taint analysis tool for Android, which we extensively discussed in this paper. Several subsequent approaches focused on improving the efficiency of taint tracking, at the expense of tracking a smaller number of taints [45]. Others proposed approaches to tracking up to 2^{32} sources by sacrificing the performance of the tool [48].

Additional work focused on tracking taints within native code [41, 49, 50], tracking implicit flows along control dependencies [22, 26, 52], and using static analysis to optimize instrumentation to improve the efficiency of dynamic analysis at runtime [13, 54]. Our work is orthogonal to all these approaches as none of them focuses on reporting statement-level taint path information.

Dynamic Taint Analysis for non-Android Software. A few dynamic taint analysis approaches outside of the Android/mobile domain focus on reporting information beyond taint sources and sinks. For example, Panorama [51] is a Windows malware detection and analysis tool that reports how taint propagates through operating system processes and resources, e.g., threads and files. FAROS [38] collects provenance information about the network, processes/memory, and the file-system. That is, similar to Panorama, FAROS reports high-level data propagation within the system among network calls, processes, and file systems. Similarly, RAIN [27] and RTAG [28] produce a provenance graph which includes information about processes, files, and network endpoints. While these tools generate certain taint provenance information, none of them reports statement-level taint paths, as ViaLin does.

Other Dynamic Information Flow Analyses. Closely related to dynamic taint analysis are other types of information flow analysis techniques, such as techniques based on dynamic forward slicing [31] – an approach for computing statements that are affected by certain program inputs via control and data dependencies. When focusing on unsafe information flows, "classical" slices can include many irrelevant control dependencies, causing over-tainting. To address this issue, several approaches also devise algorithms for computing only a subset of "necessary" control dependencies.

For example, DynFlow [33–36] is a hybrid approach that performs a dynamic forward slicing for Java with an option to precompute certain implicit flows statically. As DynFlow iteratively computes and stores a forward slice for each program variable, it duplicates path information for related variables, inducing high time and memory overhead. The authors note that their approach does not scale for large interactive programs with long traces [35]. Instead, ViaLin takes a more memory-efficient approach.

The proposal of Shroff et al. [44] introduces a dynamic information flow system which also tracks direct and indirect information flows. The formalism borrows ideas from dynamic taint analysis by tracking a taint mark per variable, which indicates whether the variable is part of the tracked flow. It then proposes to keep a set of def-use edges, adding an edge each time a tainted variable is used in an assignment. While this approach avoids duplicating information, using one taint mark only does not allow it to distinguish between information flows originating from different variables. Moreover, the approach is only conceptual and was never implemented.

Overall, existing approaches suffer from several limitations, making ViaLin the first practical approach for path-aware dynamic taint tracking that (a) implements memory-efficient, non-repetitive, and garbage-collectable structures, (b) supports multiple taint sources, (c) supports statement-instance-level (rather than statement-level) analysis, and (d) deals with Android-related constructs.

There are several existing dynamic backward slicing approaches, in particular for Android [7, 11]. These approaches first collect an execution trace and then construct slices backwards by analyzing the trace. To accurately perform post-execution backward analysis, forward execution needs to collect high volumes of (potentially irrelevant) information, making the approach impractical for large programs and long execution traces [31, 36].

9 CONCLUSION

This paper presents a dynamic taint analysis approach for Android software, implemented in a tool named ViaLin, which identifies and reports accurate statement-level taint propagation information, i.e., *how* the information flows from a sensitive source to a sensitive sink. To the best of our knowledge, ViaLin is the first to provide such a statement-level flow tracking. We evaluated ViaLin on a number of practical use cases and demonstrated that it can help detect vulnerable and malicious behaviors in large real software.

10 DATA AVAILABILITY

The implementation of ViaLin and our evaluation package are available online [8].

ACKNOWLEDGMENTS

This work is partially supported by Meta/WhatsApp Security team. We thank them for the support. We also thank the security experts whose analysis contributed to the results in Section 6.4 and the anonymous reviewers whose comments helped improve this paper.

REFERENCES

[1] [n. d.]. Avoid Hard-coded JWT Secret Keys. https://www.appmarq.com/public/tqi,1025030,Avoid-hard-coded-JWT-secret-keys.

[2] [n. d.]. DroidBench 3.0. https://github.com/secure-software-engineering/DroidBench/tree/develop.

[3] [n. d.]. ICC-Bench. https://github.com/fgwei/ICC-Bench.

[4] [n. d.]. JWT Hardcoded Secret Key. https://docs.boostsecurity.io/rules/code-jwt-hardcoded-secret-key.html.

[5] [n. d.]. Malicious WhatsApp Mod Distributed Through Legitimate Apps. https://securelist.com/malicious-whatsapp-mod-distributed-through-legitimate-apps/107690/.

[6] [n. d.]. TaintBench. https://taintbench.github.io/taintbenchSuite/.

[7] Khaled Ahmed, Mieszko Lis, and Julia Rubin. 2021. Mandoline: Dynamic Slicing of Android Applications with Trace-Based Alias Analysis. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 105–115.

[8] Khaled Ahmed, Yingying Wang, Mieszko Lis, and Julia Rubin. 2023. *Supplementary Materials. https://resess.github.io/artifacts/ViaLin/*.

[9] Steven Arzt. 2017. *Static Data Flow Analysis for Android Applications*. Ph. D. Dissertation. Darmstadt University of Technology, Germany.

[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. of Conference on Programming Language Design and Implementation (PLDI)*. 259–269.

[11] Tanzirul Azim, Arash Alavi, Iulian Neamtiu, and Rajiv Gupta. 2019. Dynamic Slicing for Android. In *Proc. of International Conference on Software Engineering (ICSE)*. 1154–1164.

[12] Golam Sarwar Babil, Olivier Mehani, Roksana Boreli, and Mohamed-Ali Kaafar. 2013. On the Effectiveness of Dynamic Taint Analysis for Protecting Against Private Information Leaks on Android-based Devices. In *Proc. of International Conference on Security and Cryptography (SECRYPT)*. 1–8.

[13] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp Von Styp-Rekowsky, and Sebastian Weisgerber. 2017. ARTist: The Android Runtime Instrumentation and Security Toolkit. In *Proc. of European Symposium on Security and Privacy (EuroS&P)*. 481–495.

[14] David Brumley, Juan Caballero, Zhenkai Liang, and James Newsome. 2007. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. In *Proc. of USENIX Security Symposium*.

[15] Michael Cao, Khaled Ahmed, and Julia Rubin. 2022. Rotten Apples Spoil the Bunch: An Anatomy of Google Play Malware. In *Proc. of International Conference on Software Engineering (ICSE)*. 1919–1931.

[16] João Cartucho. [n. d.]. Record and Replay Touchscreen Events on Android. https://github.com/Cartucho/android-touch-record-replay.

[17] Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (1977), 504–513.

[18] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[19] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.

[20] Xiaoqin Fu and Haipeng Cai. 2021. FlowDist: Multi-Staged Refinement-Based Dynamic Information Flow Analysis for Distributed Software Systems. In *Proc. of USENIX Security Symposium*. 2093–2110.

[21] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *Proc. of Network and Distributed System Security Symposium (NDSS)*.

[22] Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, and Ana Cavalli. 2015. Detection of Illegal Control Flow in Android System: Protecting Private Data Used by Smartphone Apps. In *Proc. of International Symposium on Foundations and Practice of Security (FPS)*. 337–346.

[23] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. 2006. Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks. In *Proc. of International Symposium on Foundations of Software Engineering (FSE)*. 175–185.

[24] Behnaz Hassanshah and Roland H.C. Yap. 2017. Android Database Attacks Revisited. In *Proc. of Asia Conference on Computer and Communications Security (ASIA CSS)*. 625–639.

[25] Katherine Hough and Jonathan Bell. 2022. A Practical Approach for Dynamic Taint Tracking with Control-flow Relationships. *Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 26:1–26:43.

[26] Hiroki Inayoshi, Shohei Kakei, Eiji Takimoto, Koichi Mouri, and Shoichi Saito. 2019. Prevention of Data Leakage due to Implicit Information Flows in Android Applications. In *Proc. of Asia Joint Conference on Information Security (AsiaJCIS)*. 103–110.

[27] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2017. RAIN: Refinable Attack Investigation with On-Demand Inter-Process Information Flow Tracking. In *Proc. of Conference on Computer and Communications Security (CCS)*. 377–390.

[28] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2018. Enabling Refinable Cross-Host Attack Investigation with Efficient Data Flow Tagging and Tracking. In *Proc. of USENIX Security Symposium*. 1705–1722.

[29] Ulf Kargén and Nahid Shahmehri. 2012. InputTracer: A Data-Flow Analysis Tool for Manual Program Comprehension of x86 Binaries. In *Proc. of International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 138–143.

[30] Jingfei Kong, Cliff Changchun Zou, and Huiyang Zhou. 2006. Improving Software Security Via Runtime Instruction-level Taint Checking. In *Proc. of ASPLOS Workshop on Architectural and System Support for Improving Software Dependability (ASID)*. 18–24.

[31] Bogdan Korel and Satish Yalamanchili. 1994. Forward Computation of Dynamic Program Slices. In *Proc. of International Symposium on Software Testing and Analysis (ISSTA)*. 66–79.

[32] Timothy Robert Leek, Graham Z Baker, Ruben Edward Brown, Michael A Zhivich, and RP Lippmann. 2007. *Coverage Maximization Using Dynamic Taint Tracing*. Technical Report Technical Report TR-1112. MIT Lincoln Laboratory.

[33] Wes Masri, Nagi Nahas, and Andy Podgurski. 2006. Memoized Forward Computation of Dynamic Slices. In *Porc. of International Symposium on Software Reliability Engineering (ISSRE)*. 23–32.

[34] Wes Masri and Andy Podgurski. 2008. Application-based Anomaly Intrusion Detection with Dynamic Information Flow Analysis. *Computers & Security* 27, 5 (2008), 176–187.

[35] Wes Masri and Andy Podgurski. 2009. Algorithms and Tool Support for Dynamic Information Flow Analysis. *Information and Software Technology* 51, 2 (2009), 385–404.

[36] Wes Masri, Andy Podgurski, and David Leon. 2004. Detecting and Debugging Insecure Information Flows. In *Porc. of International Symposium on Software Reliability Engineering (ISSRE)*. 198–209.

[37] Wei Meng, Ren Ding, Simon P. Chung, Steven Han, and Wenke Lee. 2016. The Price of Free: Privacy Leakage in Personalized Mobile In-Apps Ads. In *Proc. of Network and Distributed System Security Symposium (NDSS)*.

[38] Meisam Navaki Arefi, Geoffrey Alexander, Hooman Rokham, Aokun Chen, Michalis Faloutsos, Xuetao Wei, Daniela Seabra Oliveira, and Jedidiah R. Crandall. 2018. FAROS: Illuminating In-memory Injection Attacks via Provenance-Based Whole-System Dynamic Information Flow Tracking. In *Proc. of International Conference on Dependable Systems and Networks (DSN)*. 231–242.

[39] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android Taint Analysis Tools Keep Their Promises?. In *Proc. of International Symposium on the Foundations of Software Engineering (FSE)*. 331–341.

[40] Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *Proc. of International Symposium on Software Testing and Analysis (ISSTA)*. 176–186.

[41] Ali Razeen, Alvin R. Lebeck, David H. Liu, Alexander Meijer, Valentin Pistol, and Landon P. Cox. 2018. SandTrap: Tracking Information Flows On Demand with Parallel Permissions. In *Proc. of Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 230–242.

[42] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proc. of Symposium on Principles of Programming Languages (POPL)*. 49–61.

[43] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proc. of Symposium on Security and Privacy (SP)*. 317–331.

[44] Paritosh Shroff, Scott Smith, and Mark Thober. 2007. Dynamic Dependency Monitoring to Secure Information Flow. In *Proc. of Computer Security Foundations Symposium (CSF)*. 203–217.

[45] Sun, Mingshen and Wei, Tao and Lui, John C.S. 2016. TaintART: A Practical Multi-Level Information-Flow Tracking System for Android RunTime. In *Proc. of Conference on Computer and Communications Security (CCS)*. 331–342.

[46] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*. 87–97.

[47] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proc. of Conference on Computer and Communications Security (CCS)*. 1329–1341.

[48] Zhen Xu, Chen Shi, Chris Chao-Chun Cheng, Neil Zhengqiang Gong, and Yong Guan. 2018. A Dynamic Taint Analysis Tool for Android App Forensics. In *Proc. of Security and Privacy Workshops (SPW)*. 160–169.

[49] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin T.S. Chan. 2019. NDroid: Toward Tracking Information Flows Across

Multiple Android Contexts. *Transactions on Information Forensics and Security (TIFS)* 14, 3 (2019), 814–828.

[50] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. 2017. Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART. In *Proc. of USENIX Security Symposium.* 289–306.

[51] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In *Proc. of Conference on Computer and Communications Security (CCS).* 116–127.

[52] Wei You, Bin Liang, Wenchang Shi, Peng Wang, and Xiangyu Zhang. 2020. TaintMan: An ART-Compatible Dynamic Taint Analysis Framework on Unmodified and Non-Rooted Android Devices. *Transactions on Dependable and Secure Computing (TDSC)* 17, 1 (2020), 209–222.

[53] Junbin Zhang, Yingying Wang, Lina Qiu, and Julia Rubin. 2021. Analyzing Android Taint Analysis Tools: FlowDroid, Amandroid, and DroidSafe. *Transactions on Software Engineering (TSE)* (2021).

[54] Mu Zhang and Heng Yin. 2014. Efficient, Context-Aware Privacy Leakage Confinement for Android Applications without Firmware Modding. In *Proc. of Symposium on Information, Computer and Communications Security (ASIA CCS).* 259–270.

Received 2023-02-02; accepted 2023-07-27