The ISC Int'l Journal of Information Security

ISeCure

January 2022, Volume 14, Number 1 (pp. 13–25)

http://www.isecure-journal.org

SANT: Static Analysis of Native Threads for Security Vetting of Android Applications

Seyed Behnam Andarzian [1] and Behrouz Tork Ladani [2,*]

[1] *Department of Information Technology, University of Isfahan,Isfahan,Iran.*

[2] *Department of Software Engineering, University of Isfahan,Isfahan,Iran.*

ARTICLE INFO.

Article history:

Received: September 12, 2020

Revised: May 15, 2021

Accepted: June 12, 2021

Published Online: September 5, 2021

Keywords:

Android Security, Information Leakage, Mobile Security, Static Analysis

Type: Research Article

doi: 10.22042/isecure.2021.247906.572

dor: 20.1001.1.20082045.2022.14.1.2.2

ABSTRACT

Most of the current research on static analysis of Android applications for security vetting either works on Java source code or the Dalvik bytecode. Nevertheless, Android allows developers to use C or C++ code in their programs compiled into various binary architectures. Moreover, Java and the native code components (C or C++) can collaborate using the Java Native Interface. Recent research shows that native codes are frequently used in both benign and malicious Android applications. Most of the present Android static analysis tools avert considering native codes in their analysis and applied trivial models for their data-flow analysis. As we know, only the open-source *JN-SAF* tool has tried to solve this issue statically. However, there are still challenges like libC functions and multi-threading in native codes that we want to address in this work. We presented *SANT* as an extension of *JN-SAF* for supporting Static Analysis of Native Threads. We considered modeling libC functions in our data-flow analysis to have a more precise analysis when dealing with security vetting of native codes. We also used control flow and data dependence graphs in *SANT* to handle multiple concurrent threads and find implicit data-flow between them. Our experiments show that the conducted improvements outperform *JN-SAF* in real-world benchmark applications.

1 Introduction

The Android operating system has been dominated by the smartphone market with more than 70% of the total devices due to a recent study by Statcounter [1]. The applications that have been developed by third-party developers are in different categories from banking and socialization to entertainment and travel, etc. However, it has been reported many times that Android application has been a threat to its users for privacy issues like information leakage of sensitive data by malicious applications. Even benign Android applications can be vulnerable and leak sensitive information without an intention. There are researches [2–6] that addressed this issue and tried to find information leakage through taint analysis. However, it should be mentioned that none of them can find all the information leakages through native codes in the program because the usage of native code is one of the tough challenges in the security vetting of Android apps. Recent studies have shown that many applications use native codes for different purposes like efficiency and hiding premium codes from reverse engineering. There are also other

* Corresponding author.

Email addresses: behnam@eng.ui.ac.ir, ladani@eng.ui.ac.ir

ISSN: 2008-2045

objectives like evading antivirus detection programs for malicious applications. Recent years witnessed a substantial rise in the number of apps using native code and libraries. *Avdiienko et.al* in [7] have claimed that from more than one million apps that they statically analyzed, 37% of them have used native code in their applications. Also *Wei et.al* in [8] have argued that there is substantial usage (39.7%) of native code in benign apps on 100,000 Google Play applications crawled randomly; therefore, the need for security vetting these applications and their native codes is essentially vital for user's privacy.

Most of the previous works addressing information leakage through native codes are using dynamic approaches that make the analysis not cover all the program's execution paths. As far as we know, there is only one work [8] that has tried to solve this issue by using static analysis techniques. However, there are some limitations like multi-threaded native code and libC functions in taint analysis. With the advancement of technology, mobile processors with more cores have come to the market, making the use of multi-threaded applications more relevant even for native codes. Multi-threaded programmings can affect the precision of the static analysis by the dynamic nature of threads. In this work, we have addressed some of these limitations and solve the challenges related to these issues. We have also implemented the proposed solution as an extension to *JN-SAF* that we call it *SANT* (a tool for Static Analysis of Native Threads). This article is the first work that analyzes multi-threaded Android native codes statically to the best of our knowledge. Note that static analysis has more advantages than dynamic analysis in many cases. While dynamic analysis only considers a single execution path in each round of execution, static analysis can explore the whole execution state space at once. This is why static analysis promises to achieve completeness in its analysis, while dynamic analysis cannot. Hence, considering native threads in the static analysis process can significantly enhance Android applications' security vetting. This is even more important when considering that dynamic analysis of parallel execution, as in native multi-threading in Android Apps, is highly complicated and time-consuming.

In a brief statement, we can argue that the residual challenges in static analysis of Android native codes are as follows:

(1) Using native threads that are available through POSIX libraries can initiate multiple threads. These threads can pass data between each other and make a static analyzer lose the tracking taint.
(2) Having comprehensive modeling of functions in the libC library and JNI is important for taint analysis to track the tainted data more precisely.

This work presents the following contributions:

(1) Studying the behavior of remaining not modeled functions that existed in libC and JNI libraries and modeled them for more precise analysis in native codes.
(2) Using the control flow graph to distinguish the created threads in the program and separate possible concurrent threads using novel heuristics.
(3) Finding data leakage between the source and sink threads using the data dependence graph and increasing the taint analysis's precision.
(4) Implementing the proposed solutions as an extension to *JN-SAF* named *SANT*.

The rest of the paper is organized as follows: Section 2 presents the related work. The background and example is presented in Section 3. Section 4 debates over challenges and our solutions, while Section 5 describes our implementation. We discuss the conducted experiments and evaluation results of the proposed tool in Section 6. Limitations of *SANT* are discussed in Section 7, and finally, Section 8 concludes the paper.

2 Related Work

Here in this section, we briefly review several more related works to highlight our contribution's position.

FlowDroid [2] leverages taint analysis to track the sensitive data-flow in the program. It uses an app-level dummy-main method to handle Android system events then uses a flow and context-sensitive IFDS [12] algorithm to conduct taint analysis. *FlowDroid* does not handle native code and puts in a comprehensive model for native method calls.

IccTA [4] extends *FlowDroid* and leverages *IC3* [13] to resolve the application Intents. It tracks data-flow for intent calls and extracts their parameters .However, same as *Flowdroid* it does not handle native method calls.

LeakDoctor [14] is an analysis system that looks for privacy leaks by concluding if a privacy revelation from an app is necessary for all functionalities of the app. Privacy disclosures that are functionality-irrelevant for the application are not justifiable; Therefore, they will be considered potential privacy leak cases. For this purpose *LeakDoctor* combines dynamic response differential analysis with static response taint analysis from *Flowdroid*.

SoProtector [15] has consolidated static and dynamic analysis techniques to find privacy leaks in Android native codes. It uses *Flowdroid* for taint analysis

and completes the Control Flow Graph(CFG) with dynamic analysis of each app. It also uses machine learning to find similarities in Opcodes' grayscale images in native code and similar sequences between Opcodes of different apps that tend to be malicious. Like all of the dynamic analysis platforms, it cannot cover all the execution paths in native code.

TaintART [16] instruments the ART compiler and runtime and conducts a dynamic taint analysis.

GoingNative [7] first uses static analysis to find the apps using native code then leverages dynamic analysis to generate sandboxing security policies for Android native codes.

Amandroid [3] is a data-flow analysis tool that implements a component-based analysis using taint analysis and tracks possible Intra/inter-component data flows. This tool does not consider native method calls. Javadroid part of the *JN-SAF* is based on *Amandroid* and utilizes its features to capture inter-component data-flows.

NDroid [17] implements dynamic taint analysis with QEMU machine virtualizer and looks for any information that flows through native code. It uses code instrumentation in the native world to deal with information flows like JNI entry/exit and object creation. However, like all dynamic analysis tools, NDroid suffers from path coverage problems.

As the above literature review shows, the related works either do not support handling native codes or work dynamically to examine them. However, in general, dynamic analysis tools fail when we tend to consider the whole state space of the execution, which is essential when we deal with security vetting of applications. *JN-SAF* [8] is the first and the single tool that claims to do static analysis of the Android native codes. However, as we will see in the next section, it lacks some essential requirements for precise evaluation of security properties in Android applications. Hence this is our motivation to enhance them in this paper.

3 Background and Example

We have provided the necessary background information to understand how Android native codes can work with multiple threads using the POSIX API. We further discuss the remaining functions that have to be modeled correctly in previous work [8], so we can have a more precise analysis. We also provide a motivating example to discuss the challenges in tracking data-flow between multiple threads in Android native code and how remaining not modeled native functions can affect data-flow tracking in the static analysis.

3.1 Using Pthreads in Android Native Code

With the expansion of multi-core CPU in Android devices, multi-threaded programming is getting more prevalent between developers, and as a result, most of the non-trivial Android apps use more than one thread, therefore multi-threaded programming is essential to Android development. At Android NDK, POSIX Threads (pthreads) are bundled in Android's Bionic C library to support multi-threading. We will first introduce thread creation and termination. A thread can be created with the `pthread_create` function, which has the following prototype: `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void*(*start_routine)-(void*), void *arg)`. This function creates and starts a new thread with attributes specified by the `attr` input argument. If `attr` is set to NULL, default attributes are used. The `start_routine` argument points to the function to be executed by the newly created thread with `arg` as the input argument to the function. When the function returns, the thread input argument will point to a location where the thread ID is stored, and the return value will be zero to indicate success or other values to indicate an error. The thread is terminated after it returns from the `start_routine` function or we explicitly call `pthread_exit`. The `pthread_exit` function has the following prototype: `void pthread_exit(void *value_ptr)`. This function terminates the calling thread and returns the value pointed by `value_ptr` to any successful join with the calling thread. The `pthread_join` function has the following prototype: `int pthread_join(pthread_t thread, void **value_ptr)`. The function suspends the thread's execution until the thread specified by the first input argument terminates [9].

3.2 Native Functions Affecting Taint Analysis

The Native Development Kit, a.k.a NDK [10] is a set of tools that allow developers to implement a part or all of the Android applications using C or C++ languages. NDK has come up with platform libraries to manage native Activity components and use physical device units. It uses the Java Native Interface (JNI) [11] as the interface through which the Java and C++ components communicate with each other. The most use cases for NDK are reusing existing third-party C or C++ libraries, improving performance for CPU-intensive workloads such as game engines, signal processing, physics simulation, and so on. Although *JN-SAF* has modeled the JNI and NDK libraries precisely, there is still a need to model other neglected functions that can cause the

taint analysis to lose its precision in tracking the data-flow.

3.3 Motivating Example

A malicious or careless developer can leverage NDK and develop a part or all of his/her application in native code. Listing 1 demonstrates an example of this situation by an application that has been developed in two parts consisting of Java and the native world. This application leaks data through native code, and because of the presence of native threads, previous work [8] cannot detect the data leakage that happens in this situation. In this example, we have an activity class in Java world that loads a native library named `threadLeak` and imports native method `send`. On the other hand, in the native world, we have an exported native function called `send`.

As an example, here is the following sequence of events that can happen in reality:

(1) **Java world:** An Activity Component named main activity initiates a system service object and extracts the *device_id* (IMEI) from that object and assign it to a string variable and send it to the native world with the help of native function `send`.

(2) **Native world:** Native function `send` receives the sensitive value as an argument named *data*. To get the object data in string format, the program has to use the `GetStringUTFChars` and assign it to a const `char*` (in this example we named it *source*). We also have a libC function named `snprintf` which gets several characters as input and copies the second argument into the first argument. After that, the main function initiates two separate threads, where one of them receives sensitive data as input. Each of these threads has an inner loop; one of them continuously reads the global variable and logs it to the console, while the other thread will sleep at each loop iteration. When all loops have been finished, this thread propagates the sensitive data it has previously received as an argument and assigns it to the global variable. With this assignment, the other thread that was logging this global variable will log the sensitive data, and we are going to encounter a sensitive data leakage in our program.

Thanks to work done in [8], we have a somehow complete model of the JNI interactions with Java and native code that can capture the data flow between the two worlds. However, some remaining challenges still exist, and solving them could lead to more precision in our analysis. For example, to track the data-flow in native code, we have to model all the functions that can affect the data-flow, like the `snprintf` function at line 25 that copies sensitive data, assigning it

Listing 1 Example android application

```
1  package test.multiple_interactions;
2  public class MainActivity extends Activity{
3  static{System.loadLibrary("threadLeak");}
   ↪  //"libthreadLeak.so"
4
5  public static native void send(String data);
6  protected void onCreate(Bundle
   ↪  savedInstanceState){
7  super.onCreate(savedInstanceState);
8  setContentView(R.layout.activity_main);
9  TelephonyManager tel = (TelephonyManager)
   ↪  getSystemService(TELEPHONY_SERVICE);
10 String imei = tel.getDeviceId();//source
11 send(imei);
12 }}
```

```
1
2  char* Global = "globalVariable"
3  void* run_by_thread1(void* arg){
4  int count = 5 , i;
5  char* source =(char*) arg;
6  for(i=0 , i<count;i++){
7  sleep(1);
8  LOG("thread 1 looping");
9  }
10 Global = source;
11 return;
12 }
13 void* run_by_thread2(void* arg){
14 int count = 10 , i;
15 for(i=0 , i<count;i++){
16 sleep(1);
17 LOG("thread 2 sink %s ",Global);
18 }
19 return;
20 }
21 Java_threadLeak_MainActivity_send(JNIEnv
   ↪  *env,object thisObj, jstring data){
22 const char* source =
   ↪  env->GetStringUTFChars(data,0);
23 pthread_t th1,th2;
24 int ret,someInt=0;
25 char* sensetive =
   ↪  (char*)malloc(sizeOf(source));
26 snprintf(sensetive , sizeof(source),"%s",
   ↪  *source);
27 pthread_create(&th2,NULL,run_by_thread2,(void*)
   ↪  someInt);
28 pthread_create(&th1,NULL,run_by_thread1,(void*)
   ↪  sensetive);
29 pthread_join(&th1);
30 pthread_join(&th2);
31 LOG("%s","notSensetive");
32 }
```

to the other variable. If these kinds of functions are not modeled in the static analyzer, it cannot track the sensitive data, and it is going to have false negatives in its analysis. Lots of JNI functions have been modeled in previous work [8] but functions in the libC library are neglected.

Threads have a dynamic soul in most programs, so static analyzers cannot focus on them easily. Tracking data-flow between threads is a non-trivial work, and there is no static analyzer to capture data-flow between them in Android native code. For instance, in the example code depicted in Listing 1, we cannot track the data-flow statically because we do not know which thread will write to a specific variable or read from it. As we can see, there is a loop in each thread that causes threads to be concurrent, and a race would start at any thread-shared variable in the program. Therefore there is a need to have a new approach for solving this kind of challenges.

4 Challenges and Our Solutions

Geared toward having a precise security vetting in static analysis of native threads in Android applications, we chose to use the static analysis tools that exist out there and leverage them, so we do not need to start from scratch. Previous work [8] has done a great job, and we have built our ideas upon it. In the remainder of this section, we will be discussing the targeted challenges in previous work and what ideas and approaches have been made in this work to conquer them.

4.1 Tracking the Data-Flow in Library Functions

JN-SAF has leveraged the *angr* [18] symbolic execution framework and its annotation option to track the data-flow in the program. *JN-SAF* propagates the annotations through the program execution path so that each assignment or similar actions would cause the appropriate operands to get annotated. However, the library functions like libC remain intact. To track the data-flow in functions like libC or other functions affecting the data-flow, one has to simulate their exact behavior with the data. Therefore, we decided to extend the *JN-SAF* to support libC functions. For this goal, we have to simulate the propagation in these functions and implement them in the *simprocedure* functionality of *angr*. libC functions that affect the taint analysis behave similarly according to their parameters in annotation propagation. They get the first argument and copy it into the second one. Therefore we have to analyze the function's overall behavior in ARM assembly and apply our annotation propagation rules. In the arm processors' prologue phase, functions get their argument in particular registers and the program's stack. If the arguments are less than 5, their value or address would be at registers r0 to r4, and if there is more, they would be placed at stack. All of the targeted functions have less than five arguments. Therefore we use the *simprocedure* in *angr* in the way that related arguments in the prologue phase of each function would be read. If the argument was tainted, we would taint the receiver argument to track the data-flow in our analysis. For example, in Listing 1 at line 25 when `snprintf()` has been called, we would look at its parameter at register r3, and if it were tainted, we would propagate the taint to the argument at register r0 too. This example would apply to all remaining functions that can affect the data-flow tracking in our analysis.

4.2 Finding Concurrent Threads

Thread usage in programs is a prevalent task. Programs usually need to separate the process-intensive tasks or synchronized data transfers in other threads rather than the program's main thread to prevent latency in the program interface. Because of the threads' dynamic nature, trivial static analysis techniques cannot capture the data flow in these scenarios. To track the data-flow in threads, first, we need to find the concurrent ones. There may be different threads in a particular program, but most are not associated with each other. For this purpose, we have presented a new approach that leverages the path groups in the control flow graph of the program and looks for thread create and terminate functions vertices traversing the CFG. A path group is a group of finite sequence of vertices (basic blocks of the program) connected with edges as the program's execution flow. These groups start from the root vertex in the graph and continue to the leaf such that each path group denotes a different execution path in the program's control flow. As we mentioned in previous sections, threads can be created and terminated with different functions like `pthread_create()` and `pthread_join()` (there are also other functions that can terminate or create a thread). Considering these types of functions, we can track each path group in the control flow graph and differentiate between multiple concurrent thread groups. We have presented our algorithm for this purpose in Algorithm 1, and we will trace the proposed algorithm in a CFG that is shown in Figure 1. This CFG is an example of a program with two different path groups. The CFG is summarized to the thread create and terminate functions for space reasons.

Initially, we have defined three sets for retaining the concurrent functions that are created by different thread vertices in the graph. There is an exclude set as the terminated thread functions and a set

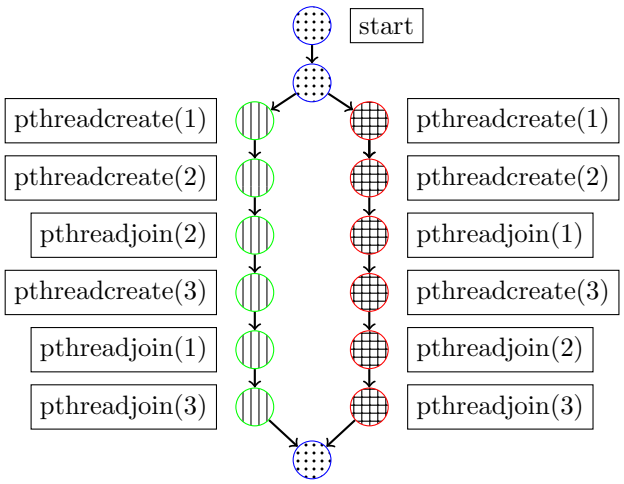

Figure 1. Control flow graph

of tuples that maps each thread group number to that group's concurrent thread functions. After that, at line 9, we traverse the vertices in a path group and check if they are members of the TTF (Thread Termination Functions) or TCF (Thread Creation Functions), respectively. While traversing these nodes, If the vertex is a member of the TTF set, we add the function which is related to that vertex to the exclude set for later use. Otherwise, if it is a TCF set member, we will save it to the concurrent group set at line 18. Meanwhile at line 12, we will be checking that if we already have any function vertices in our exclude set so to subtract them from our collected function vertices in the concurrent group set leading to a new thread group which is going to be added to the thread group map set. thread group map contains all separate thread groups and their related vertices in the control flow graph. Finally, at line 20, if the algorithm have reached to the leaf vertex, we will be saving that group as the last one, returning the thread group map set as our algorithm output.

Considering the sample CFG in Figure 1, Algorithm 1 would take each path group (vertical lines and grid lines) and distinguishes the concurrent threads. In this example, the *TGroupMap* set for the grid line path would be {(1,{thread1,thread2}),(2,{thread2,-thread3})} and for the vertical line path would be {(1,{thread1,thread2}),(2,{thread1,thread3})} respectively.

4.3 Implicit Data Leakage Between Concurrent Threads

In the previous section, we demonstrated our algorithm to collect concurrent threads and retain them as separate groups. This section will use these groups and find any data dependency that goes from a source thread to a sink thread. We call a thread source if one or more of the thread function's input arguments are tainted as sensitive data. In contrast, a thread is called a sink if the thread function has called a sink function after itself in the function call graph. After finding the source and sink threads for each concurrent group, we will leverage the data dependence graph and check if there is any data dependency between the tainted input argument in the source thread function and the argument passed to the function after the sink thread. DDG(Data Dependence Graph) is a graph that consists of vertices as variables and edges as data dependency between these variables in the analyzed program. We have proposed our approach with the help of the set theory in Algorithm 2. This algorithm takes various inputs as a set. VIAF set is an abbreviation for Vertices Input Argument Function that is the set which has been extracted from CFG where we used symbolic execution to get each basic block as vertices and their related function and input arguments for that function. Other inputs are Function Call Graph (FCG), Data Dependence Graph (DDG), set of tainted arguments in the program, and the set of sink functions in the taint analysis. The remaining inputs are the same as Algorithm 1 that will be used when we call the concurrency handler function from the Algorithm 1. At first, we define separate sets to retain sources, sinks, and leakage points. For each path group of the CFG, we will call the Algorithm 2 and get the concurrent threads for that particular path group. Afterward, we iterate over thread groups as a list and compare the thread vertices inside each group.

At line 8, each thread group has been extracted to compare the function vertices inside each group to detect sink or source threads. For this goal, we get the related function for the thread vertex in CFG. This function is the child vertex of each thread create or terminate vertex in the CFG graph. We can verify if this function is a source by checking its input arguments for being tainted with this information. Otherwise, at line 14, we look for the child function in FCG to see if the thread function has any sink function called after itself and if it is, we will save the sink function input argument for later use. After this step, we use the prior information and leverage the DDG for finding any data dependency between the source and sink thread functions. Thereby at line 18, we will start to iterate in source and sink sets and look for any path from source to sink arguments of the thread functions. If there are any paths from source to sink, we will save the source and sink points in the Leakage point set as our algorithm's output.

5 Implementation

We have implemented SANT on top of the *JN-SAF* and made it publicly available as a forked ver-

Algorithm 1 The concurrency heuristic algorithm

Input: pg: List of nodes of a PathGroup, TCF: set of Thread Create Functions, TTF: Set of Thread Terminate Functions, VIAFuncSet: set of basic blocks and their realted function and its input arguments
Output:TGroupMap: set of groups for Concurrent Threads

```
1: procedure ConcurrencyHandler(pg, TCF,TTF,VIAF)
2:     TGroupSet ← ∅
3:     ExcludeSet ← ∅
4:     TGroupMap ← ∅
5:     I ← 0
6:     let pg = v1, v2, ..., vn
7:     for j ← 1, n do
8:         Func = {func|(vj, IA, func) ∈ VIAF}
9:         if vj ∈ TTF then
10:            ExcludeSet ← ExcludeSet ∪ {Func}
11:        else if vj ∈ TCF then
12:            if ExcludeSet ≠ ∅ then
13:                I ← I + 1
14:                TGroupMap ← TGroupMap ∪ {(I, TGroupSet)}
15:                TGroupSet ← TGroupSet − ExcludeSet
16:                ExcludeSet ← ∅
17:            end if
18:            TGroupSet ← TGroupSet ∪ {Func}
19:        end if
20:        if j = n then
21:            I ← I + 1
22:            TGroupMap ← TGroupMap ∪ {(I, TGroupSet)}
23:        end if
24:    end for
25:    return TGroupMap
26: end procedure
```

sion [1] . *JN-SAF* consists of three main parts named JavaDroid, NativeDroid, and JNI Bridge. JavaDroid is implemented on top of the Amandroid [3], and it does the Dalvik-bytecode analysis. NativeDroid is built on top of *angr* [18] and it is responsible for binary code analysis. JNI Bridge is the intermediate layer that connects JavaDroid (implemented in Scala) with NativeDroid (implemented in Python) and makes them able to exchange control and data communication. JNI bridge is built on top of *jpy* [19] which is a bi-directional Java-Python bridge. There are three major steps in this framework, and we implemented improvements as an extension to one of them.

Preprocess is the first step where the APK is decompiled into the intermediate representation, pilar [3] for Java, and vexIR [20] for binary code. The second step in *JN-SAF* is environment modeling, where the entry points for the application components will be created to capture all lifecycle methods of the program. The third and last step is Summary-Based-Bottom-Up data-flow analysis (SBDA) [8] that is consisted of multiple parts, and we have implemented our approach in the *native function summary builder* part of it. Native function summary builder is accountable for adding JNI function models with *sim-procedure* option from *angr* and manage the annotation-based data-flow analysis. We have implemented the remaining functions discussed in Section 3.1 using *sim-procedure* and improved this part of the framework.

For implementing the proposed improvements in Section 3.2 and Section 3.3, we used DDG, CFG, and FCG from *angr* framework and completed the function summary builder part of the *JN-SAF*.

We take the Android application presented in Listing 1 as an example to walk through the thread leakage detection process. Considering the native world, function `Java_threadLeak_mainActivity_send()` would receive the sensitive variable `data` and assign it to another variable named `sensitive` with the help of `snprintf()` function. In this part of the program, we use our modeled libC functions and propagate the tainted value from `data` to `sensitive`. In continue, two separate threads would be created that one of them is receiving tainted argument. Algorithm 1 will capture these two `pthread_create()` functions in CFG and save them in one group as concurrent threads. After that `run_by_thread2` function will go into a loop and leaks the `global` variable. At first iterations of the loop, this variable is not sensitive but another thread named `run_by_thread1` would assign sensitive data to `global` and make it sensitive at some point in program execution. We handle this by using DDG from *angr* and check for any data dependency between source and sink thread arguments

[1] github.com/behnamandarz/argus-saf

Algorithm 2 Thread leakage algorithm

Inputs: VIAF: set of basic blocks and their realted function and its input arguments ,PG: set of PathGroups in CFG, FCG: function call graph, DDG: data dependence graph , SFS: set of functions known as a sink, TAS: Set of tainted arguments and variables in program, TCF: set of Thread Create Functions, TTF: set of Thread Terminate Functions
Output: LeakagePointSet: set of all program points that are detected as leak

procedure ThreadHandler($VIAF, PG, FCG, DDG, SFS, TAS, TCF, TTF$)
  $LeakagePointSet \leftarrow \emptyset$
  $Sources \leftarrow \emptyset$
  $Sinks \leftarrow \emptyset$
  for all $pg \in PathGroups$ **do**
    $TGroups \leftarrow ConcurrencyHandler(pg, TCF, TTF, VIAF)$
    for $x \leftarrow 1, n = |TGroups|$ **do**
      Let $(x, TgSet) \in TGroups$
      for all $Func \in TgSet$ **do**
        $ChildFunc = Child(FCG, Func)$ ▷ Thread function to be invoked
        $InputArg = \{IA | (v, IA, ChildFunc) \in VIAF\}$
        if $InputArg \in TAS$ **then** ▷ Source thread
          $Sources \leftarrow Sources \cup \{(ChildFunc, InputArg)\}$
        else if $ChildFunc \in SFS$ **then** ▷ Sink thread
          $Sinks \leftarrow Sinks \cup \{(ChildFunc, InputArg)\}$
        end if
      end for
      for all $source \in Sources$ **do** ▷ Search for any data dependency between source and sink
        $SoIA = \{InputArg | (ChildFunc, InputArg) \in source\}$
        for all $sink \in Sinks$ **do**
          $SiIA = \{InputArg | (ChildFunc, InputArg) \in sink\}$
          $ChildSet = \emptyset$
          $ChildSet = ChildSet \cup Child(DDG, SoIA)$
          while $ChildSet \neq \emptyset$ **do**
            $NewChild = ChooseRandomMember(ChildSet)$
            $ChildSet = ChildSet - NewChild$
            if $NewChild = SiIA$ **then**
              $LeakagePointSet \leftarrow LeakagePointSet \cup \{(Source, Sink)\}$ ▷ New leakage point
              break
            else $ChildSet = ChildSet \cup Child(DDG, NewChild)$
            end if
          end while
        end for
      end for
      $Source, Sink = \emptyset$
    end for
  end for
  return $LeakagePointSet$
end procedure

(in this case `sensitive` and `global`) and report them as sensitive data leakage point in the program.

6 Experimental Evaluation

We evaluated *SANT* on benchmark and real-world apps. Our dataset includes: (1) An extended version of NativeFlowBench created by [8] and extended by us so we can consider other perspectives like libC functions and concurrent execution challenges. (2) One thousand randomly chosen malware from *AMD* [21] dataset that has used native code. We perform experiments to answer the following research questions (RQ):

RQ1: How does *SANT* performs on benchmark apps?

RQ2: How is the usage of thread and libC functions in the AMD dataset?

RQ3: Does *SANT* has improvements in the security vetting of real-world applications?

RQ4: How is the performance of *SANT* compared to its related previous work?

6.1 RQ1: How Does *SANT* Perform on Benchmark Apps?

For evaluation objectives, we have developed and added eight apps to *NativeFlowBench* since there is not any existing benchmark for evaluating thread related data-flow analysis capability of Android static analysis tools for native code. *NativeFlowBench+* involves a set of hand-crafted apps, developed to test specific analysis features. Because those apps are hand-crafted, the ground truth is well known, and we can compute measures like precision and recall. *Na-*

tiveFlowBench+ is classified in 5 parts: Part A concentrates on inter-language data-flow analysis challenges, Part B concentrates on resolving native Activities, Part C concentrates on inter-component communication between Java and native components, Part D concentrates on implicit data-flow between treads in different situations and Part E concentrates on data-flow affecting functions that have not been considered in previous work. As it is stated by [8] "The apps in these test suites are not crafted to favor a particular tool and they present common scenarios one will find when reasoning about the relevant security issues".

To extend *NativeFlowBench* and cover our thread related data-flow issues, we have developed different apps in various aspects of thread implementation and also libC/JNI functions that could affect the data-flow analysis and made this benchmark publicly available [2]. These aspects are as follows:

- Thread communication through global variables
- Implementing threads using conditional variables
- Thread synchronization using the `mutex`
- Thread synchronization using R/W locks
- Thread synchronization using semaphores
- LibC and ignored JNI functions from JN-SAF

We compare the performance of *SANT* with the previous work *JN-SAF* by running these tools against each of the benchmark apps to check if they can report the correct data leakage point, and the detailed comparison is reported in Table 1. The results are shown in terms of True Positive (O), False Positive (*), and False Negative (X), if any. If some app has more than one leakage point, then the result is shown for each of them.

As it is illustrated in Table 1, *SANT* and *JN-SAF* have the same results on Parts A, B, and C considering our tool is an extended version of *JN-SAF*. However, it has outperformed the previous work in Part D and E of the benchmark. In Part D of the benchmark, because *SANT* can analyze implicit data-flow between threads, it has outperformed *JN-SAF* in three cases of *native_thread_global_var* and *native_thread_global_mutex* and *native_thread_global_conditional_var*. In two cases of *native_thread_semaphores* and *native_thread_r/w_locks*, we could not capture the data leakage through threads because of the complexity of the synchronization process between threads. Analyzing this kind of apps requires the exact modeling of how these synchronization mechanisms work. We leave this to future work. In Part E, we detected the apps using libC and JNI

Table 1. The experimental results on benchmark

App Name	*SANT*	*JN-SAF*
Part A: Inter-language dataflow		
native_source	O	O
native_nosource		
native_source_clean		
native_leak	O	O
native_leak_dynamic_register	O	O
native_dynamic_register_multiple	O	O
native_noleak		
native_noleak_array	*	*
native_method_overloading		
native_multiple_interactions	O	O
native_multiple_libraries	O	O
native_complexdata	O	O
native_complexdata_stringop	*	*
native_heap_modify	O	O
native_set_field_from_native	OO	OO
native_set_field_from_arg	OO	OO
native_set_field_from_arg_field	OO	OO
Part B:Native Activity Resolve		
native_pure	O	O
native_pure_direct	O	O
native_pure_direct_customized	O	O
Part C:Inter-component Communication		
icc_Javatonative	O	O
icc_nativetoJava	O	O
Part D:Native Threads		
native_thread_global_var	O	X
native_thread_mutex	O	X
native_thread_conditional_var	O	X
native_thread_semaphore	X	X
native_thread_r/w_locks	X	X
native_thread_noleak		
Part E:Remaining functions		
jni_array_region	O	X
libc_copy_string	O	X
Sum, Precision and Recall		
O, higher is better	22	17
*, lower is better	2	2
X, lower is better	2	7
Precision p = O/(O + *)	91%	89%
Recall r = O/(O + X)	91%	70%
F-measure f = 2pr/(p + r)	91%	77%

O = True Positive, * = False Positive, X = False Negative

[2] github.com/behnamandarz/NativeFlowBenchPlus

functions affection data-flow that have not been considered and modeled in previous work.

Eventually, both *SANT* and *JN-SAF* had false alarm on *native_complexdata_stringop* and *native_noleak_array* because both tools do not do precise string analysis and cannot distinguish different indexes of a Java array.

6.2 RQ2: How Much Is the Usage of Thread and libC Functions in AMD Dataset?

Android malware dataset (AMD) consists of 24,384 samples that are from 71 different families, with a total of 135 varieties. These malwares have diverse behaviors as they include various types of malware from trojan, backdoor, and ransomware to spyware and adware. In this dataset, 34 families are covering 5,365 samples that have native code inside their applications. We have randomly chosen 1000 malware samples from these 5,365 samples to cover all 34 malware families, as mentioned earlier. In the first place, we have analyzed their CFG to look for libC or thread related functions to discover how much the usage of these functions is prevalent in Android apps. We have presented our findings in Table 2. As we can see in Table 2, these functions' usage is pretty common. From 34 malware families that we have analyzed, 74% used libC and 44% used thread related functions. There was about 8% of .so files that we could not analyze since they were compiled with other architectures rather than ARMv5, and our binary analysis tool [18] does not support them properly.

6.3 RQ3: Does *SANT* Has Improvements in Security Vetting of Real-World Applications?

We ran our experiment using the previously mentioned 1000 samples from *AMD* dataset and compared *SANT* with its base work *JN-SAF*. There are nine families in *AMD* dataset that has been reported to have information leakage through native codes. Both *SANT* and *JN-SAF* were able to detect information leakage in 8 families, and the missed one is *Lotoor*, which is a family of all the rooting tools. *SANT* has a better results in families named *Boqx*, *DroidKungFu* and *VikingHorde*, because of handling threads and libC functions. In *Boqx* family *SANT* were able to report 52 more samples to have information leakage. Similarly, it has reported 31 and 3 more information leakage in samples from *DroidKungFu* and *VikingHore* families. However, in other families, the results were the same, and there were 18 false positives in other families that do not have information leakage. We assume that these false positives are because of insufficient taint granularity in both tools. The number of detected samples in the nine malware families with information leakage for both *SANT* and *JN-SAF* is presented in Figure 2. Our experiment's final results with *AMD* dataset in three analysis measures are presented in Table 3. In the following, we will discuss our observations from this experiment.

Table 2. Usage of thread and libC functions

Family name	Samples	Thread	LibC
Airpush	36	✓	✓
Andup	36	✓	✓
Boqx	110	✓	✓
Dowgin	33	✓	✓
Droidkungfu	80	✗	✓
FakeAV	4	✗	✗
FakeUpdates	3	✓	✗
Fjcon	10	N/A	N/A
Gingermaster	18	N/A	N/A
GoldDream	31	N/A	N/A
Gumen	46	✗	✓
Jisut	20	✗	✓
Kemoge	15	✓	✓
Ksapp	23	✓	✓
Kyview	22	✗	✓
Leech	29	✗	✗
Lotoor	66	✗	✓
Mmarketpay	13	N/A	N/A
Mseg	25	✓	✓
Ogel	6	✗	✓
Opfake	7	✗	✗
Penetho	17	✗	✗
Ramnit	8	N/A	N/A
Slembunk	80	✓	✓
Smskey	29	✗	✗
Spybubble	9	✗	✗
Tesbo	5	✗	✗
Triada	73	✓	✓
Updtkiller	22	✗	✓
Vikinghorde	7	✓	✗
Winge	18	✗	✗
Youmi	48	✗	✓
Ztorg	16	✗	✗

After modeling the remaining libC functions from previous work, we have detected more malware samples from *DroidKungFu* malware family. This malware is a backdoor that tries to get root privilege on the device and execute malicious codes. To get root privilege, it uses *secbino* program and if it failed to root the device, it will copy *secbino* to */data/data/p-kg/secbino* and use the *chmod 4755* command to get the execution permission and execute *secbino* to get the root privilege and download other malware APKs. *JN-SAF* detects these behaviors by modeling those

Linux programs that can execute shell commands such as *popen*, *system*, *execv* and extract the parameters of those system API calls and apprehend what shell commands are executed. However, other malware variants from this family are using libC functions that prevent the parameters from reaching the shell commands. By modeling these functions, *SANT* can capture the data leakage behavior in these variants.

Triada acquires the IMSI of the device in Java world and sends it to the native method named `nativeSayTest()`, and that function will leak IMSI by invoking `SmsManager` class from Java world and transfers the sensitive data out of the device. *Gumen* also had the same behavior as *Triada* but leaking the IMEI of the device.

Boqx malware family leverage native code for command and control communications. This malware starts a service that launches two threads named *statThread* and *extThread* to handle C&C communication. Using this communication, *Boqx* tries to load malicious payloads from *xbox.ooqqxx.com* dynamically. *SANT* can capture the native threads implicit dataflow and report it as a leakage point. It has reported 52 more samples of this malware family that previous work did not.

Ogel and *UpdtKiller* use the native code to cover their malicious identities like URL or premium numbers and retrieve this data whenever needed. *SANT* and *JN-SAF* both have the same results on these families.

VikingHorde malware family creates a botnet that uses proxied IP addresses to disguise ad clicks, generating income for the attacker. This malware communicates with the C&C servers using native code to receive commands or new malware payloads. Our tool has detected implicit information leakage through threads in the native world from 6 samples of this malware, while previous work only reports 3 of them.

6.4 RQ4: How Is the Performance of *SANT* Compared to Its Related Previous Work?

We have tested both SANT and JN-SAF on five different malware samples from five families in the AMD dataset to compare our tool's overhead with previous work. These malware samples were chosen from *Triada*, *Boqx*, *DroidKungFu*, *Gumen*, and *SlemBunk* families. The results have shown that there is a low overhead on all samples, based on the size of their Native code. As it can be seen from Table 4, the most overhead was on *DroidKungfu* family with about 12%, and the lowest overhead was on *Triada* with only 5%. The results of these samples and others are presented in Table 4.

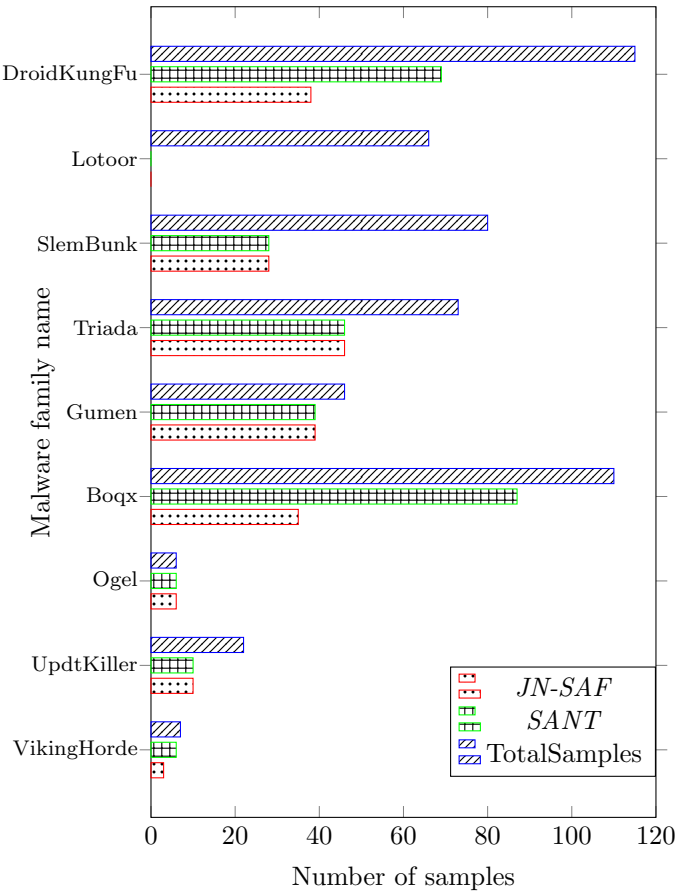

Figure 2. *SANT* and *JN-SAF* results on AMD dataset

7 Discussion

Analyzing Java and native world inter-language communication requires precise resolution of string values. *SANT* propagates constant strings in both worlds, and it will not be able to get the precise string values if they are manipulated in the way. As explained in prior research [22–24], string analysis is an expensive and non-trivial job, and we leave it to future research.

JavaDroid in *SANT* inherited some limitations like Java reflection and dynamic class loading from *Amandroid* [3]. We can mitigate Java reflection limitation using [25] or [26] in our future works.

Path explosion issue can occur in NativeDroid because of the symbolic execution of large programs in *angr*. There are ways to handle this issue partially. We can use SMT solvers to guide our symbolic execution engine to explore the preferred paths or merge some particular states [27]. We can also use catching mechanisms like function summarization for symbolic execution [6] to handle repetitive functions in the program. One other mitigation could be implementing chopped symbolic execution [28] in *angr* tool. Dynamic code loading issues cannot be handled by static analysis because of downloading the program run-time codes.

Table 3. Comparison between *SANT* and *JN-SAF* on AMD dataset

Measure	*SANT*	*JN-SAF*
Precision	94.1%	91.9%
Recall	55.4%	39%
F-measure	69.7%	54.8%

Table 4. SANT run-time overhead compared to *JN-SAF* on AMD dataset

Malware Family	*SANT* run-time overhead
DroidKungFu	12%
Gumen	10%
SlemBumk	8%
Boqx	8%
Triada	5%

One solution could be a sandbox mechanism to let the program run and download its possible dynamic loaded codes and pass them to our static analyzer.

As in all static analyzers, obfuscated or packed programs are hard to be analyzed because of the possibility of generating run time codebase or strings. This is because obfuscated and packed programs may hide some of their codes in encrypted from, which will be decrypted in the run time. Therefore, since we do not have any clue what these encrypted codes might be, we cannot analyze them with our tool.
We considered and modeled libC functions that could affect the taint analysis procedure. We did not consider other third-party libraries in our analysis since they are not in our work scope.

8 Conclusion

We have presented *SANT*, an extended version of *JN-SAF*, that is a static analyzer of Android native codes for security vetting of Android applications. We leveraged various program graphs(CFG, DDG, FCG) from *angr* binary analysis tool and tracked the data-flow between possible concurrent threads with our new approach. As far as we know, this is the first work in Android native code that considers threads in the binary analysis. Our experiments show that *SANT* can be used for security vetting of Android applications using native codes.

References

[1] Mobile Operating System Market Share Worldwide. Accessed 9 Feb 2020. http://gs.statcounter.com/os-market-share/mobile/worldwide

[2] Arzt, Steven, et al. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps." *Acm Sigplan Notices* 49.6 (2014): 259-269.

[3] Wei, Fengguo, Sankardas Roy, and Xinming Ou. "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps." *ACM Transactions on Privacy and Security (TOPS)* 21.3 (2018): 1-32.

[4] Li, Li, et al. "Iccta: Detecting inter-component privacy leaks in android apps." In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE, 2015.

[5] Avdiienko, Vitalii, et al. "Mining apps for abnormal usage of sensitive data." In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE, 2015.

[6] Andarzian, Seyed Behnam, and Behrouz Tork Ladani. "Compositional Taint Analysis of Native Codes for Security Vetting of Android Applications." In *2020 10th International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE, 2020.

[7] Afonso, Vitor, et al. "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy." In *The Network and Distributed System Security Symposium*. 2016.

[8] Wei, Fengguo, et al. "Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code." In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018.

[9] Feipeng Liu.:Android Native Development Kit Cookbook. *PACKT publishing*. (2013)

[10] Android NDK. Accessed 15 Feb 2020. https://developer.android.com/ndk/.

[11] Java Native Interface. Accessed 5 March 2020. https://docs.oracle.com/javase/8/docs/technotes-/guides/jni/

[12] Sagiv, Mooly, Thomas Reps, and Susan Horwitz. "Precise interprocedural dataflow analysis with applications to constant propagation." *Theoretical Computer Science* 167.1-2 (1996): 131-170.

[13] Octeau, Damien, et al. "Composite constant propagation: Application to android inter-component communication analysis." In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE, 2015.

[14] Wang, Xiaolei, et al. "Leakdoctor: Toward automatically diagnosing privacy leaks in mobile applications." In *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 3.1 (2019): 1-25.

[15] Xu, Guangquan, et al. "SoProtector: safeguard privacy for native SO files in evolving mobile IoT applications." In *IEEE Internet of Things Journal*

7.4 (2019): 2539-2552.
[16] Sun, Mingshen, Tao Wei, and John CS Lui. "Taintart: A practical multi-level information-flow tracking system for android runtime." In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016.
[17] Xue, Lei, et al. "NDroid: Toward tracking information flows across multiple Android contexts." In *IEEE Transactions on Information Forensics and Security* 14.3 (2018): 814-828.
[18] Shoshitaishvili, Yan, et al. "Sok:(state of) the art of war: Offensive techniques in binary analysis." In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016.
[19] jpy a Python Java Bridge. Accessed 14 May 2020 https://github.com/bcdev/jpy
[20] Shoshitaishvili, Yan, et al. "Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware." NDSS. Vol. 1. 2015.
[21] Wei, Fengguo, et al. "Deep ground truth analysis of current android malware. " In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, Cham, 2017.
[22] Christensen, Aske Simon, Anders Møller, and Michael I. Schwartzbach. "Precise analysis of string expressions." In *International Static Analysis Symposium*. Springer, Berlin, Heidelberg, 2003.
[23] Li, Ding, et al. "String analysis for Java and Android applications." In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015.
[24] Shannon, Daryl, et al. "Abstracting symbolic execution with string analysis. " In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007.
[25] Zhang, Yifei, et al. "Ripple: Reflection analysis for Android apps in incomplete information environments." *Software: Practice and Experience* 48.8 (2018): 1419-1437.
[26] Li, Li, et al. "Droidra: Taming reflection to support whole-program analysis of android apps." In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 2016.
[27] Baldoni, Roberto, et al. "A survey of symbolic execution techniques." In *ACM Computing Surveys (CSUR)* 51.3 (2018): 1-39.
[28] Trabish, David, et al. "Chopped symbolic execution." In *Proceedings of the 40th International Conference on Software Engineering*. 2018.

Seyed Behnam Andarzian received his B.S. degree in Computer Science from Shahid Chamran University and his M.S. degree in computer science from University of Isfahan in 2020. His research interests lies in the binary analysis and information leakage detection in smartphone applications. He has been working on different static and dynamic analysis tools and published his works in this research topics.

Behrouz Tork Ladani received his B.Sc. in software engineering from University of Isfahan, Isfahan, Iran in 1996, and M.Sc. in software engineering from Amir-Kabir University of Technology, Tehran, Iran in 1998. He received his Ph.D. in computer engineering from Tarbiat-Modarres University, Tehran, Iran in 2005. He is currently full professor and Dean of Faculty of Computer Engineering at University of Isfahan. Dr. Ladani is member of Iranian Society of Cryptology (ISC). He is also managing editor of the Journal of Computing and Security (JCS) and member of editorial board of the International Journal of Information Security Science (IJISS). Dr. Ladani's research interests include security modeling and analysis, software security, computational trust, soft security, and rumor analysis.