FSAFlow: Lightweight and Fast Dynamic Path Tracking and Control for Privacy Protection on Android Using Hybrid Analysis with State-Reduction Strategy

Zhi Yang[1](zynoah@163.com), Zhanhui Yuan[1§](15213077@bjtu.edu.cn), Shuyuan Jin[2](jinshuyuan@mail.sysu.edu.cn), Xingyuan Chen[1§](chxy302@vip.sina.com), Lei Sun[1§](13523556215@139.com), Xuehui Du[1], Wenfa Li[3], Hongqi Zhang[1]
[1]PLA Information Engineering University, Zhengzhou, China, [2]Sun Yat-sen University, Guangzhou, China,
[3]University of Science and Technology Beijing, Beijing, China

2022 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-1316-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/SP46214.2022.9833764

***Abstract*—Despite the demonstrated effectiveness of dynamic taint analysis (DTA) in a variety of security applications, the poor performance achieved by available DTA prototypes prevents their widespread adoption in production systems, especially the Android system with limited computation and storage resources.**

To overcome DTA's overhead bottlenecks, recent research efforts aim to decouple taint tracking logic from program execution. Continuing this line of research, this work proposes FSAFlow, a novel hybrid taint tracking and control system, to reduce DTA overhead significantly while ensuring sound Android privacy protection. FSAFlow further separates the path tracking logic from the corresponding taint tracking logic and the control of the information flow path is optimized. Specifically, a classic static analysis algorithm is first modified to search target paths and their key branch information. Then, the potential paths that violate the user's predefined privacy protection policy are chosen and encoded with a Finite State Automaton (FSA). A small amount of FSA-based state management code is inserted into the corresponding position in the program. Finally, it monitors the program's state of path execution and prevents information leakage during runtime.

The efficiency and correctness of FSAFlow are proved by theoretical analysis. The experimental results show that FSAFlow incurs lower overhead than several representative DTA optimization approaches, 2.06% for popular applications, and 5.41% on CaffeineMark 3.0. FSAFlow has fewer false negatives in implicit flow tracking than the Android DTA platform, TaintDroid, and achieves higher precision than the static analysis tool, FlowDroid, by verifying the paths that never occur and tracking in the complete execution stage of the loop body at runtime.

I. INTRODUCTION

Android systems increasingly contain private user information. Because of their openness, they often run a large number of untrusted programs, which results in frequent privacy leakage [1], [2]. The propagation and flow of sensitive information must be controlled [3]. Fine-grained information flow control is inseparable from information flow tracking, also known as taint tracking. Specifically, it first tags source data (e.g., GPS location data returned by API call) as tainted. Then, these taints are propagated through data flow or control flow. Finally, whether the tainted data reach sinks (e.g., the network output) is checked.

Solutions that rely on static taint analysis (STA) can comprehensively search for potential information leakage paths but may cause over-approximations by generalizing all possible behaviors of a program and produce false positives [1], [4]-[7]. Furthermore, if a discovered leakage path and a legitimate path share a sink, it is challenging for STA to apply correct judgments and control due to the lack of runtime path information (Refer to the example in Section II for this problem).

In contrast, solutions that rely on dynamic taint analysis (DTA) search for security issues at program runtime. The characteristics of DTA enable it to provide sufficient context, which helps DTA to avoid false positives. DTA has been used in various application domains, including information flow control [5], [8]-[13], vulnerability discovery [14]-[20], security attacks [19], [21]-[27], malware detection [27], [29], privacy leakage analysis [30]-[32], etc. Despite its advantages, DTA is rarely used in practice today [9]. The most serious problem is its performance overhead since, in a pure DTA situation, every instruction of the original program usually takes 6 to 8 extra taint tracking instructions to propagate a taint tag [35]-[37]. Several efforts have been made to reduce the overhead of DTA through hardware acceleration [5], [39]-[42], parallelization [36]-[37], [42]-[46], code optimization [14], [31], [52], etc. For example, TaintPipe [37] can reduce the slowdown of dynamic taint analysis to 1x in some test cases. However, other problems then arise. Hardware-accelerated DTA requires additional hardware support. Parallelized DTA sacrifices the spare cores and wastes energy to accelerate DTA. Code optimization can only be realized for specific code features. Moreover, few pure DTA tools track implicit flows, which incur a significant performance overhead due to the lack of comprehensive branch structure information at runtime [30].

Hybrid analysis (HA) has been explored to accelerate DTA by static pre-optimized tracking logic [9]-[10], [52], [60], static pre-reduced tracking range [10], [61], static part sharing tracking tasks [62], etc. Although HA can use STA rapidly to calculate implicit taint propagation caused by the control dependencies, these traditional HA are still based on dynamic taint tracking supplemented by STA, where the dynamic instruction-by-instruction tracking is still the main component. For example, Iodine [9] assumes that the fast path is often executed, while the slow path is less executed. Iodine only optimizes tracking for fast paths. If the fast and slow paths have equal probability of execution, or the profiler's prejudgment of whether a path is fast or slow is not accurate, the advantages of Iodine are nullified, but the path switching and recovery overhead will increase. Therefore, these methods only exhibit their advantages in certain cases and are not adequately robust. Moreover, HA's static analysis usually lacks runtime profile information, which leads to reduced optimization and accuracy. Additionally, HA

§ Corresponding Authors

DOI 10.1109/SP46214.2022.00042

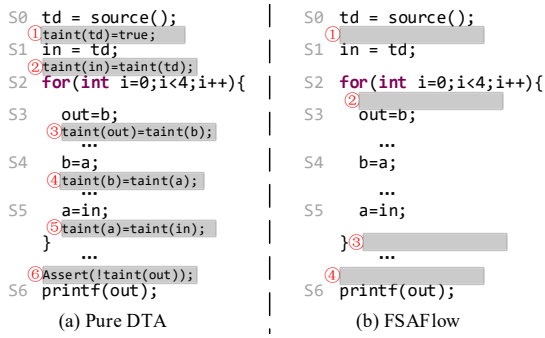

Fig. 1. Comparison of tracking between pure DTA and FSAFlow

introduces several STA's shortcomings. For example, the classic STA algorithm IFDS (Interprocedure, Finite, Distributive, Subset) [34] does not analyze the stable periods of loop structures to avoid path length explosions, which may produce false positives (Refer to Section III D for this problem).

As for the above work, the tracking logic is not separated from the program execution, or the separation is not complete. Regarding privacy protection and performance improvement, the use of instruction-by-instruction tracking mechanisms for running programs is undesirable. Instead of caring about how information is spread between variables, research should consider whether the information flow path has occurred, whether it has reached the sink node, and whether the policy allows it. This requires a more effective regulatory mechanism. This paper proposes FSAFlow to address the above-mentioned challenges in HA. Its key idea is to separate the slow taint tracking logic from the program execution and further separate the path control logic from the taint tracking logic. Its static analysis does not require dynamic profile information, and it only searches for potential paths. Its runtime control is based on changes in path state rather than taint state. Figure 1 compares pure DTA and FSAFlow using an example. It assumes that variable *td* is a source, and *printf*(*out*) is a sink. The taint is propagated to variables *in*, *a*, *b* and *out* successively. Finally, it reaches the sink, causing an information leakage. Figure 1 (a) shows the tracking of pure DTA. It adds taint tracking instructions for each data flow statement, as shown in ①-⑥. Figure 1 (b) shows the tracking of FSAFlow. It only inserts path tracking instructions at the start, end, and branches of the path, as shown in ①-④. The detailed FSAFlow instrumenting logic is shown in Section III-D and Figure 5. FSAFlow is directly applicable to Android APK. The demo video can be downloaded from https://github.com/ FSAFlow/FSAFlow.

The contributions of this paper are as follows:

1) A novel information flow protection system, FSAFlow, is proposed that uses path tracking rather than taint tracking for privacy protection. The target program is only instrumented with lightweight supervision code related to the path, which does not involve cumbersome variable taint propagation. Based on this, fast information flow tracking can be achieved at runtime.

2) FSAFlow uses a finite state machine to monitor the path state efficiently at runtime. It provides path-aware control which is difficult for STA and achieves higher precision than STA by verifying the paths that never occur and tracking throughout the execution stage of the loop body. It makes quick tracking operations by updating the path state only at branch statements, and can find and track implicit flow paths more easily than DTA by using STA comprehensively to search potential paths.

```
1  public class DoLogin extends Activity {
2     protected void onCreate(Bundle savedInstanceState) { ……
3       Intent data = getIntent();//source
4       password = data.getStringExtra("passed_password");
5       new RequestTask().execute("password");
6       …… }
7     class RequestTask extends AsyncTask <String,String,String> {
8       protected String doInBackground(String...params) {
9         postData(params[0]);
10        …… }
11      public void postData(String valueIWantToSend){
12        ……
13        InputStream in = responseBody.getEntity().getContent();
14        result = convertStreamToString( in );
15        ……
16        Loginfo=Loginfo_tmpt1;
17        Loginfo_tmpt2=Loginfo_tmpt1;
18        if (result.indexOf("Correct Credentials") != -1) {
19          if(MonitorFlag){
20            Loginfo[0]="Successful Login" +": account="+
                                    username + ":" + password;
21            }else{
22              Loginfo[0]="Unsuccessful Login";}
23            if(Loginfo[0].matches("Successful Login"))
24              MonitorFlag=false;
25            Log.d("Login status:",Loginfo_tmpt2[0]);//sink
26              …
27      } }
28      private String convertStreamToString(InputStream in ) {
29        …… }
30  }
```

Fig. 2. Code snippet of InsecureBankv2

3) The experimental results show that FSAFlow incurs a low overhead, 2.06% for popular applications, and 5.41% on CaffeineMark 3.0, which is lower than some representative DTA optimization approaches such as tracking on demand, local code optimization, optimal hybrid taint analysis.

II. MOTIVATION

In this section, a specific example, InsecureBankv2 [43], is provided to illustrate typical data leakage behavior and explain our motivation. InsecureBankv2 is used to evaluate the efficiency of security holes analysis tools. Data leakages designed in this app are basically the same as those in actual apps.

Figure 2 shows a code snippet from InsecureBankv2 with a slight modification. First, in the *onCreate* method, *data*, the initial position of privacy data (the 3rd line), is regarded as the source. The privacy information eventually flows to the *log* method (the 25th line), which is regarded as sink nodes. This code prints out the password in the log and the password is easily leaked out, which is a serious privacy issue.

It is understood that Android presently cannot prevent such leakages. Android's permission control mechanism only determines which sources (such as location, IMEI, etc.) or sinks (sending to networks, etc.) can be accessed by apps following the user's choices. Android systems usually give a prompt such as whether to allow the program to read location information, but provide no mechanism that controls data propagation from sources to sinks nor information flow-control policies such as whether to allow the program to write location information into files. Information flow policies are crucial for privacy protection.

Owing to commercial interests or untrusted third-party components, some popular apps, such as instant messaging, and navigation software, not only use private data to complete normal functions but also spread private data without informing users. In this case, information flow tracking and control are needed to prevent the leakage paths beyond normal functions. However, the current methods have the following problems:

1) This leakage path can be found by traditional STA, but the kind of control to take at *log.d* is unknown. In Fig. 1, two paths converge to this sink. One contains the password information (the if-then path starting from lines 19-20), and the other contains no password information (the if-else path starting from lines 21-22). There is no runtime context information at *log.d*. In practice, a sink node often contains multiple paths from the

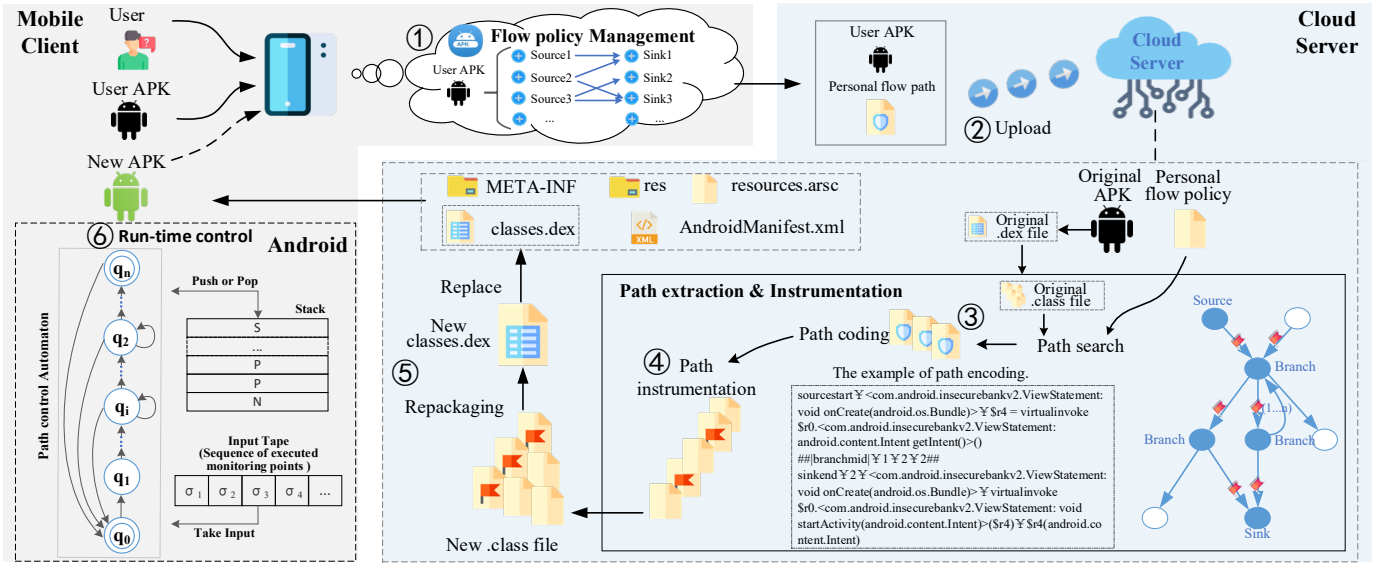

Fig. 3. Workflow of FSAFlow

sources. Simple prohibition or permission will cause usability or security problems.

2) If traditional DTA is adopted, the dynamic context information can be provided for correct control. However, the main problem is its excessive running overhead for single-step instruction track and its suffering in implicit flow tracking.

To solve these problems, a novel hybrid analysis method, FSAFlow is proposed for privacy protection. The key idea is to adopt global path tracking instead of micro taint tracking at runtime. Meanwhile, path tracking is optimized to ensure tracking efficiency at runtime. FSAFlow provides path-aware control at sink points to ensure the accuracy of control.

III. Design and Implementation

A. Overall framework

As shown in Figure 3, FSAFlow consists of a mobile client and a cloud server (distinguished by color), and its workflow is as follows: First, at ①, a user can use the client to customize the flow policies for any application on the mobile phone. By managing the information flow path from source to sink, the user obtains a customized privacy policy, such as whether the location information can be sent through the network or written to local files. Then, at ②, the personal flow policy file is uploaded to the cloud server with the corresponding APK file for processing.

On the server side, FSAFlow performs static analysis and instrumentation processing according to the uploaded flow policy. In the static analysis stage, as shown at ③, all the potential paths that violate the policy will be considered. Meanwhile, the key node information of these paths is recorded during this stage. In the static instrumentation stage, as shown at ④, the important nodes of the supervised path are instrumented for efficient path-state management. Then, at ⑤, the instrumented code is repackaged to generate a new APK file and returned to the client. Based on this, the security-enhanced application will be run and monitored efficiently on users' mobile phones as shown in ⑥.

B. Static analysis

The static analysis stage aims to mark potential leakage paths and record their necessary node information for static instrumentation. Since branch jumps are key nodes for distinguishing different paths, the contexts of branch nodes on the path will be recorded. Monitoring points are mainly instrumented at branches. In order to efficiently locate branch nodes during the instrumentation stage, the contexts of the function call and return will be recorded as separate nodes to indicate the method body where subsequent branch nodes are located. Therefore, for the branch node, we only need to record its relative position in the method body.

The FSAFlow's static analysis component is realized by modifying the classic IFDS framework [34] and the FlowDroid tool. Many static analysis problems, including taint analysis, pointer analysis, live variables, and constant propagation, can be solved by IFDS using a special graph-reachability algorithm.

The IFDS problem is represented by a tuple $(G^{\#}, D, F, M, \sqcap)$, where $G^{\#}= (N^{\#}, E^{\#})$ is called the Interprocedural Control Flow Graph (ICFG), and it provides the directed supergraph representation of a program. Each $n \in N^{\#}$ corresponds to a statement of the program, and each $(n_1,n_2) \in E^{\#}$ indicates that Statement n_2 is a direct successor to Statement n_1. D is a finite set of information flow facts, indicating the variables that are infected. $F \subseteq 2^D \rightarrow 2^D$ is a set of information flow functions. A flow function defines the impact of a statement on a set of flow facts. For example, the statement s: $x = y$ would be associated with a flow function that maps a fact set $\{y\}$ (i.e., y is tainted) to a fact set $\{x, y\}$ (x and y are both tainted), which can be expressed as $\{x,y\}=f_s(\{y\})$. M: $E^{\#} \rightarrow F$ is a map from the edges of $G^{\#}$ to flow functions, and the meet operator $\sqcap$ is the union for taint analysis.

$G^{\#}$ consists of a set of flow graphs $\{G_1,G_2,\ldots\}$ (one per function). The flowgraph G_p of a function p is composed of a unique *start* node s_p, a unique *exit* node e_p, and the remaining nodes representing the statements and predicates in p. In $G^{\#}$, a statement m of calling a function q is represented by two nodes, a *call* node c_m, and a *return-site* node r_m. Three edges are used to connect m and q: a *call-to-return* edge from c_m to r_m, a *call-to-start edge* from c_m to s_q, and an *exit-to-return edge* from e_q to r_m.

To convert static analysis problems to graph-reachability problems, $G^{\#}$ is extended to a *supergraph* $G^*=(N^*,E^*)$, where $N^* = N^{\#} \times (D \cup \emptyset)$ and $E^*=\{<u, d_x> \rightarrow <v, d_y> \mid (u, v) \in E^{\#}, d_y \in f_u$

Algorithm 1 Static analysis

1: **Input**: $G^{\#}=(N^{\#}, E^{\#})$ //the interprocedural CFG of the program
2: **Input**: *Sourcelist*, *Sinklist* //the lists of Sources and Sinks
3: **Output**: P_{FSA} //a path identified by key-node sequence
4: Set<Statement> *SourceStatements*= $G^{\#}$.*searchStatement*(*Sourcelist*)
5: **for each** *s* in *SourceStatements* **do**
6: P_{FSA}=SourceTagHead+[s,d_i] //Variable d_i is tainted if *s* is executed;
7: *Worklist.enQueue* ([<-,∅>→<s,d_i>,P_{FSA}]) //Enqueue an item on *Worklist*
8: *PathEdge.clearQueue*( ) //Clear the queue *PathEdge*
9: *sn*=0
10: **while** *worklist*≠*null* **do**
11: (<s_p,d_1>→ <n,d_2>, P_{FSA})= *Worklist.deQueue*();
12: **if** (<s_p,d_1>→ <n,d_2>, P_{FSA}) ∈ *PathEdge*
13: **continue**
14: *PathEdge.enQueue*((<s_p,d_1>→ <n,d_2>, P_{FSA}))
15: **Switch**(***n***)
16: **case** $G^{\#}$.*isSinkstatement* (*n*): //*n* is sink statement
17: P_{FSA}+= SinkTagHead+[*sn*++;*n*; d_2]
18: output P_{FSA}
19: **case** $G^{\#}$.*isCallstatement* (*n*): //*n* is call statement
20: *endSums*=*Summaryhash.get*([$G^{\#}$.*getCalleeMethod*(*n*), d_2])
21: **If** *endSums*≠*null*
22: **for each** (<*n,d2*>→<*returnsite*(*n*), d_3>,$Part_{FSA}$) in *endSums* **do**
23: *Worklist.enQueue*([<*n,d2*>→<*returnsite*(*n*), d_3>,P_{FSA}+$Part_{FSA}$])
24: **Else**
25: **for each** (<*n,d2*>→<*firstnodeofcallee* (*n*), d_3>,P_{FSA}) in $E^{\wedge}$ **do**
26: P_{FSA} += CallTagHead+[*n*; *firstnodeofcallee*(*n*)]
27: *Worklist.enQueue*([<*n,d2*>→<*firstnodeofcallee*(*n*), d_3>,P_{FSA}])
28: **case** $G^{\#}$.*isExitstatement*(*n*): //*n* is return statement
29: P_{FSA} += CallreturnTagHead+[*n*; *returnsite*(*n*)]
30: $Part_{FSA}$ = P_{FSA}.*Substring*(P_{FSA}.*lastIndexof*($G^{\#}$.*getMethodOf*(*n*));
31: *Summaryhash.put*($G^{\#}$.*getMethodOf*(n),[<s_p,*d1*>→<*n,d2*>, $Part_{FSA}$])
32: **for each** (<*n,d2*>→<*returnsite*(*n*), d_3>,P_{FSA}) in $E^{\wedge}$**do**
33: *Worklist.enQueue* ([<*n,d2*>→<*returnsite*(*n*), d_3>, P_{FSA}])
34: **case** $G^{\#}$.*isAssignmentstatement* (*n*): //*n* is assignment statement
35: **for each** (<*n,d2*>→<*m,d3*>,P_{FSA}) in $E^{\wedge}$ **do**
36: *Worklist.enQueue*([<s_p,d_1>→<*m,d3*>, P_{FSA}])
37: **for each** d_3'∈*Backforward_alias*(*m*,d_3) **do**
38: *Worklist.enQueue*([<s_p,d_1>→<*m*,d_3'>, P_{FSA}])
39: **case** $G^{\#}$.*isBranchstatement*(*n*): //*n* is branch jump statement
40: **for each** (<*n,d2*>→<*m,d3*>,P_{FSA})∈$E^{\wedge}$**do**
41: (*size*, *no*,*type*) = $G^{\#}$.*branchinfo*(*m*);
42: [*sn'*; *size'*; *no*; *type*]=*searchincurrentmethod*(P_{FSA}, *m*)
43: **if** [*sn'*; *size'*; *no'*]does not exist **then**
44: P_{FSA}+=BranchTagHead+[*sn*++; *size*; *no*; *type*; -1; *m*]
45: **Else**
46: P_{FSA}+=BranchTagHead+[*sn*++; *size*; *no*; type; *sn'*; *m*]
47: *Worklist.enQueue*([<s_p,d_1>→<*m,d3*>, P_{FSA}])
48: **end**

(d_x) }. For example, (n_1: x=y → n_2: z=x) ∈$E^{\#}$ can be extended to {(<n_1,y>→ <n_2, y>, <n_1,x>→ <n_2, y>)} ∈ E^* , if *y* has been tainted before n_1. Note that ∅ signifies an empty set of facts.

Based on the definition of flow functions, the IFDS algorithm performs width-first traversal of the ICFG from the source statement and searches for all paths ending with the sink statement in E^*. Therefore, if there is a path from node <n_0: y=source(), ∅> to node <n_{final}: sink(x),z> in G*, the sequence of statements corresponding to the nodes on the path constitutes the information flow path from source to sink.

FSAFlow continues to extend G^* to $G^{\wedge}$=($N^{\wedge}$,$E^{\wedge}$), where $N^{\wedge}$= N^*×String and $E^{\wedge}$={<*u*, d_x, P_{FSA}>→<*v*, d_y, *P'*$_{FSA}$> | (*u*, *v*)∈E^*, d_y ∈$f_u(d_x)$}. P_{FSA} records the key information to identify the path so far, including the context of branch statements and call/return statements, etc. The search and recording are performed concurrently. The specific static analysis is shown in Algorithm 1.

In Algorithm 1, The inputs *Sourcelist* and *Sinklist* are obtained by reading user policy files, and $G^{\#}$ is generated by calling the SOOT tool. First, $G^{\#}$ is searched to find the source statement set. Then, each source is taken as a starting point to search $E^{\wedge}$ for the paths to the target sinks. The searched edges are recorded in the variable *pathedge* and will not be searched again

Fig. 4. Information flow analysis and path coding

(lines 12-13). The variable *Worklist* records the set of edges that remain to be searched. The newly found edges will be added to *Worklist* (lines 7, 23,27, etc.), and the old edges that have been searched will be removed from the *worklist* (line 11).

The string array P_{FSA} stores the information of each key node sequentially. In P_{FSA}, different nodes are segmented with a special separator. FSAFlow extracts different key information for different statement types.

1) The statement type *sink* (lines 16-18). It indicates that a flow path has been found. At this time, the path expansion is terminated and P_{FSA} is output.

2) The statement type *function call* (lines 19-27). When this node is first encountered, the caller's and callee's *call-site* information is appended to P_{FSA}, and the path is extended to the callee. Otherwise, the new extended path fragment $Part_{FSA}$ about the callee can be extracted from the function summary *Summaryhash*, and the search is then performed on the *return-site node*.

3) The statement type *function return* (lines 28-33). The function summary is appended to *Summaryhash* for reuse. Then, the algorithm returns to the *call* node and continues to expand the next statement. *Summaryhash* stores the path fragment of the information flow when it flows through the function.

4) The statement type *assignment* (lines 34-38). P_{FSA} will not be updated. The assignment statement is not the key information for FSAFlow, but, in this case, the alias problem is considered. FSAFlow performs on-demand alias analysis. Whenever a variable is tainted, FSAFlow searches backwards for its aliases and then taints them as well. The result variable output

by the function of backward alias analysis, the current statement, and P_{FSA} combined form a new node in $N^{\wedge}$. This node will be exploited to continue the forward data flow analysis.

5) The statement type *branch* (lines 39-47). The information m_p of the first statement m_s of a subsequent branch path is appended to P_{FSA} as a node. The node m_p consists of a tuple <*sn*, *size*, *no*, *type*, *sn'*, *m* >, where *sn* represents m_p's position in P_{FSA}, *size* represents the branching number of the branch, *no* is for the branch serial number of the target path, and *type* is for the node type, indicating to which branch type the selected branch belongs. The node type can be *in* (entering loop body), *out* (the type of exiting loop body), and *x* (others). If m_s appears more than once on the path within the current function, *sn'* is used to record the first corresponding position of m_s in P_{FSA}. *m* is for the statement itself. Finally, when P_{FSA} is output, *m* is removed from this node in P_{FSA}. Refer to Section III-D and C for the use of branch information.

Taking the code snippet in section II as an example, algorithm 1's path analysis process is given. The polluted process at the sink can be abstracted as the process steps ①-⑧ in Figure 4(a). Meanwhile, the main structure of the path obtained by the static analysis is shown in Figure 4(b). FSAFlow operates on the Jimple code, therefore, to facilitate subsequent instrumentation, the output path is also represented by Jimple statements.

In step ①, the tainted variable *password* starts to propagate forward. At ②, the class *RequestTask* is tainted. Step ③ continues tracking when the tainted variable is called by *postData*. At ④, the tainted variable is called by *ConvertStream ToString*. Step ⑤ represents the return of the function call. The tainted variable continues to propagate forward through ⑥ and passes through two branches due to the *if* statements. At ⑥, *loginfo* is tainted. This triggers the backward alias analysis through steps ⑦ and ⑧, the alias *loginfo_tmpt2* for *loginfo* is found and then propagates forward as a normal taint. Finally, at ⑨, the tainted variable *loginfo_tmpt2* leaks at the sink.

FSAFlow adjusts the output format of P_{FSA} and aggregates the nodes according to classes and methods. After the static analysis is completed, the key information of paths is output and written into path files.

C. Static instrumentation

The static instrumentation stage aims to embed the lightweight code for path monitoring into the apps, which transforms untrusted apps into policy-enforcing apps. According to the results of static analysis and user-defined flow policies, the disallowed paths are extracted and instrumented for the source nodes, branch nodes, and sink nodes.

Before instrumentation, FSAFlow preprocesses the path file. Because there is a loop structure, a statement may repeatedly appear on the path, thus let *SN* be the set of all the positions (*sn*) on the path where a statement *m* appears. FSAFlow appends *SN* to the node where *m* first appears on the path for *m*'s location to be instrumented only once. After appending, other nodes about *m* remain on the path to ensure the consistency of the control flow when the following nodes are instrumented. Moreover, each loop structure involved in the path is assigned a unique ID that is used for status monitoring in nested loops. Refer to Section III-D for the use of loop ID.

Furthermore, all paths are managed hierarchically and uniformly by a multilevel hash table $HashMap_{policy}$ according to

Algorithm 2 Code Instrumentation

```
1: Input:  HashMap_policy //the paths stored with hash table
2: Input:  classfiles //the class files of the target program
3: Output: instrumented classfiles
4: for each class c in HashMap_policy do
5:   for each Method m in c do
6:     for each path_id in hash(c,m) do
7:       for each path fragement f in hash(c,m,id) do
8:         for each node n in f do
9:           if n is source type then
10:            stmt_n = seek n in m
11:            insert source-control-code before stmt_m to control the source
12:          else if n is branch type and n.sn'=-1 then
13:            Continue;
14:            for each following path p at the next branch
15:              stmt_m = the first statement of p
16:              insert branch-control-code before stmt_m to control the branch
17:          else if n is sink type then
18:            stmt_n = seek n in m
19:            insert sink-control-code before stmt_m to control the sink
20:    output instrumented class c
21: end
```

```
1  public class DoLogin ... {
2    protected void onCreate(...) {……
       detail_embeded_code.source_logic(114, 0,"Source information");
3      Intent data = getIntent();//source
4      password = data...;
       …… }
7    class RequestTask ... {
8      protected String doInBackground(...) {
         ……}
11     public void postData(...){
12       ……
           detail_embeded_code.branch_logic(114, 5, "accept");
18         if (...) {
             detail_embeded_code.branch_logic(114, 6,"accept");
19           if(...){
20             Loginfo[0]=username + password;
21             }else{
22                Loginfo[0]="...";  }
               ……
               detail_embeded_code.sink_logic(114, 7,"Sink information");
25             Log.d(Loginfo_tmpt2[0]);//sink
                 ……
```

Fig. 5. Code instrumentation

class, method, and path identification. Hereby, the involved classes and methods are each traversed only once. $HashMap_{policy}$ has a data structure of <classname, <*method*, <*pathid*, *pathfragment*>>>, and is directly accessed through the multi-level key-value. Each key value of the method index can store the fragment information of multiple paths while *pathfragment* only stores branch nodes. The nodes associated with function calls on the path can be removed and will not occur in $HashMap_{policy}$. Subsequent instrumentation is based on $HashMap_{policy}$.

FSAFlow extends SOOT's BodyTransformer and the method *internalTransform* is implemented to traverse the units (statements) of all method bodies and insert the monitoring code in the specified location. Algorithm 2 describes the process of code instrumentation in detail.

The classes that appear in $HashMap_{policy}$ are loaded item by item from the actual class files. Then, the methods in the class file are individually parsed. For each method, the corresponding path identification and path segment are extracted from the hash table, and there are three contexts for node processing:

1) If it matches the source type, the corresponding statement is located and the control code is inserted before it, which can activate the monitoring of the specified path at runtime.

2) If it matches the branch type, the next branch from the current location is located. For different following paths, the responding control codes are inserted before the first statement of the subsequent target path as shown in Table I in Section III-D. If a node is the non-first repeated node (*sn'*=-1), it means that this node has been processed and can be ignored.

3) If it matches the sink type, the corresponding statement

is located and the control code is inserted before this statement, which allows information flow control and auditing.

In a path, a method may be called more than once. The *newinvoke* type is used for updating the current instrument location to the method's initial node. Due to limited space, the details of the relevant *newinvoke* nodes in the description of Algorithms 1 and 2 are omitted.

After completing the code instrumentation for related class files by traversing $HashMap_{policy}$, a set of new class files is obtained. Then, all the new class files are repacked into .dex files to replace the .dex files in the original APK file. Using the compression package, a new APK file is generated.

The code instrumentation example of InsecureBankv2 is shown in Figure 5. Only the statements before and after the instrumentation node are retained in this figure, and part of the instrumentation for the branch node is omitted. The inserted code is actually a function call code. Considering the hierarchy and interface of the project, the functions needed for actual operation are encapsulated into separate classes. When these functions are called in the running process, the related functions can be realized.

D. Runtime control

The runtime control stage aims to efficiently monitor the change of path states and dynamically prevent information leakage when the leakage path occurs. This section describes the specific control principle based on a finite state machine. The code snippet shown in Figure 1 is taken as an example to illustrate the runtime control mechanism.

If the IFDS algorithm is used to analyze this example, after it traverses the loop body from $s2$ to $s5$ three times, the nodes <$s3$,*td*>, <$s3$,*in*>, <$s3$,*a*>, <$s3$,*b*>, <$s3$,*c*> in N^* of the program will be found. When IFDS tries to traverse $s3$ for the fourth time, it will find that there is no new node available for the path expansion. Then, IFDS stops traversing the loop body and continues to traverse $s6$. Finally, it will output a path $s0$-$s1$-$s2_1$-$s3_1$-$s4_1$-$s5_1$-$s2_2$-$s3_2$-$s4_2$-$s5_2$-$s2_3$-$s3_3$-$s4_3$-$s5_3$-$s6$ (si_n represents the n-th execution of si), while the actual execution path is $s0$-$s1$-$s2_1$-$s3_1$-$s4_1$-$s5_1$-$s2_2$-$s3_2$-$s4_2$-$s5_2$-$s2_3$-$s3_3$-$s4_3$-$s5_3$-$s2_4$-$s3_4$-$s4_4$-$s5_4$-$s6$, which leads to information leakage and causes false negatives in IFDS. However, all STA techniques may face this problem.

To deal with this problem effectively, the process of taint propagation is divided into three periods along the path: 1) Forward propagation period (N): The executed statement is outside the loop, corresponding to the execution period of $s1$ and $s6$. 2) Loop propagation period (P): The execution statement is in the loop body. According to IFDS, there are still new node extensions in the current loop round that corresponds to the execution period of "$s2_1$-$s3_1$-$s4_1$-$s5_1$-$s2_2$-$s3_2$-$s4_2$-$s5_2$-$s2_3$-$s3_3$-$s4_3$-$s5_3$". 3) Loop stable period (S): The executed statement is in the loop body structure. According to IFDS, the current loop round has no new node extension. The execution is then no longer tracked by IFDS, which corresponds to the execution period of " $s2_4$-$s3_4$-$s4_4$-$s5_4$".

To avoid the problem of path length explosion, IFDS does not analyze the S period yet still achieves good accuracy. However, further monitoring of the S period is beneficial. If there is a leakage path that includes an S period, it will cause false negatives in the IFDS. The use of an over-accurate STA algorithm for the S period can easily lead to excessive overhead because

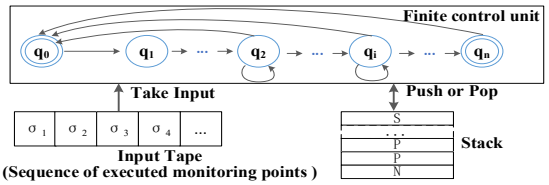

Fig. 6. Path control Automaton

STA has difficulty in determining the number of loop executions in the S period. Thus, a more effective and reasonable method is required to solve this problem. Before the execution enters into the S period of a loop, it has experienced the P period and reached the maximum taint state by taint accumulation. Thus, to maintain the maximum taint state of the loop, the repetitive execution of the same code segment of a loop in the S period often tends to be stable. In practice, the code in the loops may be repetitive logical behaviors, such as scientific calculation, file reading, and writing, etc. The execution of the path usually does not change the final propagation state of the P period. Thus, these paths are still valid but have no role in propagation. In most cases, to ensure safety, FSAFlow continues to monitor the stable period, and only monitors the key branch nodes to ensure performance and correctness, as shown in Table III. In a few cases, a false positive judgment may occur during the S period. Then, this path's monitoring policy is revised to reduce the false positive judgments by no longer monitoring its stable period.

FSAFlow implements the control of the path based on a pushdown automaton, which is an extension of the finite-state automaton and is composed of a state controller, an input, and a stack. As shown in Figure 6, given a path *path* output by static analysis and assuming its length is N, at runtime the corresponding automaton is formally expressed as $M=(Q,\Sigma,\Gamma,\delta,q_0,F)$, where:

1) Q is a finite set of states. Each node i on the *path* corresponds to a state $q_i(i>0)$. q_0 corresponds to the inactive state of the path; q_1 corresponds to the source node; q_n corresponds to the sink node.

2) Σ: $2^{\{0,1,\ldots,N\}} \times TYPE \times L_{ID}$ is the input alphabet. Each input symbol corresponds to a monitoring point of FSAFlow, and the input sequence represents the sequence execution of monitoring points for program execution. The input symbol σ is composed of a tuple $(SNs,type,l_{id})$. $SNs \in 2^{\{0,1,\ldots,N\}}$ is the set of allowed node numbers corresponding to the monitoring point. Each node $q_i(i>0)$ on the *path* is assigned a node number i in sequence. In a special case, $SNs=\{0\}$ indicates that the monitoring point is on the non-target branch path. $type \in \{in,out,x\}$ is the type of the location of the monitoring point. *in/out/x* respectively indicate that the monitoring point is an entry/exit point of loop structure and other types. l_{id} represents the identification of the loop structure. If $type=x$, then $l_{id}=-1$.

3) Γ: $Period \times L_{ID}$ is the stack alphabet. Stack is a "last in, first out" storage device that records the corresponding execution period of a loop when executing nested loops. *Period*: $\{N,P,S\}$ respectively represent the forward propagation period (N), loop propagation period (P), and loop stable period (S). L_{ID} is the set of IDs of the loop. For a stack symbol $\gamma = (period, l_{id})$, if $period=N$, then $l_{id}=-1$.

4) δ: $Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma$ is the transition function. According to the current state, if the current input and the symbol are at the top of the stack, the next action of M is determined, including the state transition, and the Push or Pop operation.

TABLE I. TRANSITION FUNCTION OF M

Rule	State	Input[a]	Stack[a]	Output/Action	
				State	Stack
1	q_0	<{1},-,->	empty	q_1	push(N,-1)
2	q_i ($N>i>0$)	<{i+1},x/out,->	<N,->	q_{i+1}	
3		<SNs,in, l_{id}> satisfying i+1∈SNs	<N,->	q_{i+1}	push(P, l_{id})
4		<SNs,x, -> or <SNs,in, l_{id}> satisfying i+1∈SNs	<P,l_{id}>	q_{i+1}	
5		<SNs,in, l_{id}> satisfying i+1∈SNs, l_{id}≠l_{id}	<P,l_{id}>	q_{i+1}	push(P, l_{id})
6		<SNs,out, l_{id}> satisfying i+1∈SNs	<P,l_{id}>	q_{i+1}	pop();
7		<SNs,in, l_{id}> satisfying i+1∉SNs	<P,l_{id}>	q_{i+1}	pop(); push(S, l_{id})
8		<{0},-,->	<N,-> or <P,->	q_0	Clear stack
9		<SNs,in, l_{id}> or <SNs,x, ->	<S,l_{id}>	q_i	
10		<SNs,in, l_{id}> satisfying l_{id} ≠ l_{id}	<S,l_{id}>	q_i	push(S, l_{id})
11		<SNs,out,l_{id}>	top:<S,l_{id}>	q_i	pop();
12	other			q_0	Clear stack

a. - = wildcard

```
   ①/*---- Monitor point of source---------------------------*/
1  path.state=1;          //Variable Path records path state,
2  push(N);               //stack operations are performed on Variable stack.
   ②/*---- Monitor point of branch entering a loop-----------*/
3  if (stack[top].phase!=S && path.state+1∈{2,3,4})
4       path.state++;
5       if(stack[top].phase==N)      //forward propagation period(N),
6          stack[top]=P;            //loop propagation period(P)
7  else if (stack[top].phase==P && path.state+1∉{2,3,4})
8          stack[top]=S;            //loop stable period(S)
9          path.state++;
10      else                        /*Here, there is only one loop on the path,
11         clear(stack);              and our simplified code does
12         path.state=0;              not distinguish the loop ID.*/
   ③/*-----Monitor point of branch exiting a loop------------*/
13 if (stack[top].phase!=S && paths.state+1∈{5})
14      Pop; paths.state++;
15 else if (stack[top].phase==S)
16         pop;
17      else
18         clear(stack);
19         path.state=0;
   ④/*-----Monitor point of sink-----------------------------*/
20 c=0; sendMessage("Warnning: FSAFlow:…information leak…");
```

Fig. 7. Code instrumentation

5) $q_0 \in Q$ is the initial state, which indicates that the path is not activated.

6) F:{q_0, q_n}⊆Q is the set of accepting states.

The transition function of M is listed in Table I. M starts from the state q_0 and the empty stack. After the source statement is executed, it reaches the q_1 state and enters the N period. During the N period, when a branch jump such as If/Switch is encountered, the state will be updated one step forward if the jumping target points to the next node in *path*; when a While/For loop is encountered, M will enter the P period, and the state is updated one step forward if both the jumping target and the target path point must enter the loop body. During the P period, when a branch jump is encountered, the state will be updated one step forward if the jumping target points to the next node in *path*; otherwise, M will enter the S period if the branch jump enters the loop body again while the next node in *path* points to the loop exit branch. In this case, it is indicated that the program should continue to execute the loop body after the execution reaches the path's maximum value of taint propagation. During the S period, the branch that does not jump from the loop body is continuously monitored, and the path state remains unchanged. When a branch that jumps from the loop body is encountered, M will pop up the stack and restore the last period indicated by the stack as the current period. Then, it continues tracking and monitoring according to different period requirements. During the P or S period, loop nesting will occur if M determines to enter a different loop by distinguishing the loop ID. In this case, new period information will be pushed into the stack, and the assignment of the new period remains that of the current period. If other situations are encountered, M will return to the q_0 state and clear the stack.

FSAFlow exploits global variables to record the execution states and the stack of all supervised paths. As shown in Figure 1, the example has four monitoring points. The main control codes of these monitoring points are shown in Figure 7.

If the execution reaches the corresponding monitoring point of the sink, FSAFlow will check whether the execution matches the required state. If so, FSAFlow will limit the execution of the sink. If the intercepted sensitive data are numeric, they are assigned the value 0. If the intercepted sensitive data are character data, the character is first replaced with the ASCII code for 0. To prevent covert communication by using the length of sensitive data, the random number generation API is further called to determine the length of cleared character data randomly. This confuses potential eavesdroppers, rendering it difficult to determine if the received information is truly sensitive and from their partners. Finally, to avoid affecting the normal functioning of the program, the purified data is passed to the sink point for execution. Moreover, the users will be notified that information has been intercepted, as shown in Figure 8 (c). To prevent attackers from removing path monitoring code from an app by upgrading the software, FSAFlow records the fingerprint of each app's executable file. FSAFlow will always first check whether the fingerprint has changed before an app runs. If it has changed, the user must re-apply for the app's path monitoring. Furthermore, to prevent attackers from escaping detection through complex reflections, we will adopt a strict taint-sensitive policy: if any input parameters are tainted, all output parameters in the reflections will be tainted. This taint propagation policy is relatively strict, but it may be the best practical approach to prevent attacks.

As concerns the tracking of native calls, similar to FlowDroid, FSAFlow directly updates the taint state of the parameter variables of the call by following predefined propagation rules, rather than entering the native call body to continue the analysis. This reduces the difficulty of path control in tracking. Accordingly, the internal statements of the native method will not be recorded on the path. FSAFlow can still monitor the path by setting monitoring points at branch statements without additional processing.

E. Implementation

The completion of each step in the FSAFlow system is shown in Figure 8. The interaction between the client and server adopts the WebSocket protocol and full-duplex mode to enhance the stability and availability of the system.

The cloud server was written in Java, JDK 1.8. The static path analysis module was developed by modifying more than 2000 lines of FlowDroid code. The path monitoring instrumentation framework of FSAFlow was developed on the SOOT platform, where the intermediate representation of the 3-address code provided by Jimple, and the accurate call graph analysis framework lay an important foundation for the FSAFlow system. Meanwhile, the Dexpler plugin and Heros framework were also used. The client apps can run on the latest version of Android, which is downward compatible with Android 4.0 and above.

Figure 8(a) illustrates the user-defined information flow-

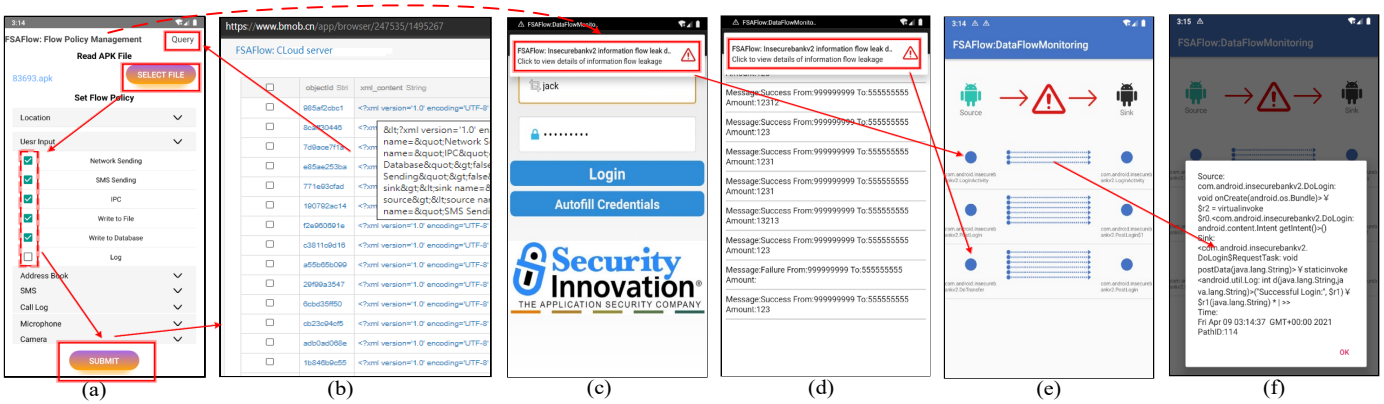

Fig. 8. Components of FSAFlow

path management interface of FSAFlow. The function of selecting the target APK file and selecting the information flow source node and sink node has been realized. After being submitted to the cloud server for processing, as shown in Figure 8(b), the new APK file can be downloaded through the query function. Taking InsecureBankv2 as an example, the running of the processed APK file is shown in Figure 8(c)-(f). Before the sensitive information is leaked out by the information flow, it will be limited, and a notification will be sent to the user after being successfully intercepted, as shown in Figure 8(c)-(d). The user can click on the notification to view the details of the leakage path, which is shown in Figure 8(e)-(f).

FastDroid [69] is an STA method, similar to FlowDroid. FastDroid implements a static information flow search based on a taint value graph by modifying the FlowDroid tool. Although the experiment in [69] showed that FastDroid can obtain better search performance, FSAFlow is not implemented on FastDroid since FastDroid's path is only composed of variable relationships. The statements on the path, including branch statements, are invisible in FastDroid, which makes it difficult to extract key path information from its output for path tracking.

F. Limitations

False negatives/positives: FSAFlow searches all paths using the STA tool FlowDroid. Although STA benefits from analyzing the complete program code, FSAFlow may exhibit false negatives due to implementation limitations [70], such as reflective calls and inaccessible code. FlowDroid resolves reflective calls only if their arguments are string constants. However, if reflective call targets are determined by external configuration or network servers, the edge of a reflection call will be lost in the ICFG, resulting in potential false negatives. In order to prevent malicious developers from writing their code deliberately like that to leak whatever they want to leak, our further work will adopt a strict taint-sensitive policy mentioned in Section III D. For native code, FlowDroid assumes a sensible default: the call arguments and the return value become tainted if at least one parameter is previously tainted. Although this may be the best practical approximation in a black-box setting, it is also generally unsound. To minimize the storage overhead, FSAFlow may also exhibit false positives in coarse tracking granularity for arrays, lists, etc.

Partially support for multithreading: FSAFlow has three phases: program static analysis, static instrumentation, and runtime control. FSAFlow's runtime control can handle multithreaded cases since it sets the path states as global variables that are managed through synchronized monitoring functions. Calls to these functions are multithreading-safe. FSAFlow's static analysis is implemented on the modified FlowDroid. FlowDroid itself partially supports multithreading analysis, including AsyncTask/Java's normal threading/Java's Runnable mechanisms. Should the improvement to FlowDroid support more multithreading mechanisms, such as Java's Executor mechanism, FSAFlow would provide full multithreading support.

Implicit flows: Untrusted programs can launder taint through implicit flows [63] [64], which poses a greater challenge to privacy protection. Similar to FlowDroid, FSAFlow cannot analyse implicit flows through table lookups. For implicit flows through control dependencies, pure DTA systems may incur significant performance overhead due to the lack of comprehensive branch structure information [30]. Although both our evaluations and the analysis of the IFDS algorithm in Section V-B-3) show that the performance of FSAFlow is not significantly affected in most cases, there remain a few where large-scale implicit flows can be identified, and solutions must still be developed. For how to decide whether an implicit flow is a true violation, FSAFlow comes to a decision according to the user-defined policies, avoiding the difficulties in judging the code among benign and malicious cases. All information flows, including implicit information flows, from a source to a sink that are prohibited by the user will be prevented by FSAFlow.

In general, implicit flows (as well as reflection and native code) remain major problems, however, our solution to implicit flow control caters mostly to benign code that is written so as to avoid such complicating constructs. For instance, a developer may vouch for a program's trustworthiness by indicating that it is fit for FSAFlow. Future work will attempt to analyze the specific characteristics and behaviors of malicious implicit flows for reducing misjudgment.

IV. Function & Complexity Analysis

This section reports on the theoretical analysis conducted to prove that FSAFlow can perform correct instruction tracking and it is more efficient than the classic DTA system.

A. Function Analysis

For ease of description, a definition is presented as follows:

Definition 1 (Basic block) Basic block refers to a sequence of statements executed sequentially in a program with only one entry and one exit. The entry is the first statement and the exit is the last statement.

FSAFlow continues to monitor the path in the loop stable period to avoid misreporting. For a repeatedly executed loop body, its execution in the loop stable period tends to be stable to maintain the final taint state in the previous propagation period. Table III in section V-A also shows that it is true in most cases. However, FSAFlow can track any path output by STA tools, by removing Rules 7 and 10 listed in Table I.

Theorem 1 (Correctness) In a program Q, for any path p found by IFDS/FlowDroid, by only tracking in the forward and loop propagation period, FSAFlow can report p if, and only if, p occurs when Q is running.

Proof: Assume that path $p=s_1->s_2->s_3\ldots s_i*\ldots s_j*\ldots s_k->s_m$ is judged by IFDS/FlowDroid as a leakage path, where s_1 is the source node and s_m is the sink node. The jump statement can be used as the split node to express p as a basic block sequence $q=B_1->B_2->B_3\ldots>Bn$. FSAFlow instrument a monitoring point M_{Bi} in ahead of each B_i and transforms Q into Q'.

Before any B_i is executed, its M_{Bi} is executed first. According to the transition rules, if the preorder state is q_{i-1}, the execution of M_{Bi} will update the state to q_i. When the source statement is executed, the path is activated, and Q' reaches the state q_1 from the initial state q_0. Any state transition from q_i to q_{i+1} indicates that after B_i is executed, the code of B_{i+1} is also subsequently executed. The possibility of other blocks being executed during the transition from B_i to B_{i+1} must be excluded. Assuming that there is an execution path $B_i->E_1->E_2\cdots->E_X->B_{i+1}$, consider the following two cases. First, E_1 occurs on the path q, and the position set of E_1 on q is labelled as $K=\{k \mid k\neq i+1, B_k=E_1\}$. Then, SNs of the monitoring point M_{E1} of E_1 is equal to K. Before E_1 is executed after B_i, M_{E1} is first executed. According to the rules, M_{E1} restores the path state to q_0 since $i+1\notin SNs$ does not meet Rules 2-6, indicating that tracking of the current path is stopped. Therefore, when $B_i->E_1->E_2\ldots->E_X->B_{i+1}$ is executed at q_i, the state will not enter q_{i+1} from qi in the current path. Thus, this state transition in the current path can only occur if there is a direct execution from B_i to B_{i+1}. Second, E_1 is not a basic block on path q. Then, SNs of the monitoring point M_{E1} of E_1 is $\{0\}$. If M_{E1} is executed, FSAFlow will restore the state to q_0, and also stop tracking the current path. Hence, the state transitions from qi to q_{i+1} occur if and only if B_{i+1} is executed immediately after B_i is executed. Furthermore, the generalized state transition $q_1->q_2\ldots, -> q_n$ must be accompanied by the flow path $B_1->B_2->B_3\ldots->B_n$ being established. Thus, the execution reaches q_n state if and only if p occurs. ■

B. Complexity Analysis

Theorem 2 (FSAFlow's tracking and control runtime overhead) Given a program Q with E being the set of statements in Q. Assume u paths are monitored in Q and the average number of different branch nodes on a path is n and the average number of statements of a monitoring point is r. Then, the average overhead of FSAFlow's runtime is $nru/|E|$.

Proof: Given a path p, according to the FSAFlow instrumenting mechanism, the monitoring points can only be set at branch nodes. Additionally, the repeated branch nodes share a monitoring point. The execution frequency of a monitoring point depends on that of the corresponding branch statement. Assume that all statements in E are uniformly executed, and p has n different branch nodes. Then, the executed frequency ratio of all monitoring pointers of p is $n/|E|$. If the average execution statement number of a monitoring point is r, the ratio of the average runtime overhead of monitoring p is $nr/|E|$. Extended to u paths, the ratio of FSAFlow's runtime overhead is $nru/|E|$.

The randomness of instruction execution for r must be considered. Most program execution does not follow a leakage path. Thus, when a monitoring point code is executed, it is most likely to be in an inactive state q_0[47]. For most monitoring points, the state transition from q_0 to q_i ($i\neq1$) is not consistent with the rules, indicating that only the beginning code of the monitoring point is executed, and the subsequent complex judgment may not be executed. Many set-determination operations may be omitted in the loop stable period. Consequently, the actual amount of code executed at each monitoring point may be small.

Concerning u, only the potential user paths that violate the relevant policies are monitored, rather than all information flow paths. According to the previous work on the analysis of actual software [9], in commercial systems, such leak paths are rare.

As for n, the nodes on the path mainly consist of sequential statements such as assignment, moving, and calculation. The number of branch statements of a path is usually considerably smaller than the path length. Besides, the path length is usually much smaller than $|E|$, thus n is far smaller than $|E|$.

In summary, since $nru/|E|<<1$, the performance overhead can be very low. As a comparison, in the DTA systems adopting an instruction-by-instruction tracking mechanism, assume that each instruction requires a instructions for tracking and that they are generally used for copying and calculating taints in a shadow memory. Then, its overhead is $a>1$, which usually increases by an order of magnitude and is significantly higher than that of FSAFlow. ■

Theorem 3 (Termination of Algorithms 1 and 2) Algorithms 1 and 2 can all always terminate.

Proof: Algorithm 1 continues to extend G^* to $G^\wedge=(N^\wedge,E^\wedge)$, where $N^\wedge=N^*\times$String and $E^\wedge=\{<u, d_x, P_{FSA}>\longrightarrow<v, d_y, P'_{FSA}> \mid (u, v)\in E^*, d_y\in f_u(d_x)\}$. Meanwhile, the P_{FSA} records the key node sequence on the information flow path thus far, including the information of function call and jump statement. Besides, to achieve better efficiency, Algorithm 1 records and updates the path information while searching.

In Algorithm 1, *pathedge* records the set of path edges that has been searched. As each extended edge is added, more variables will be infected. Meanwhile, each edge will be inserted into the list *pathedge* after the first extended analysis. Later, if the new extended edge is in the *pathedge* list, the task is already indicated to have been executed. Then, to avoid repeated analysis, this analysis is stopped. With the extension of the edge, the number of infected variables increases monotonically. Since the number of variables and the number of edges are both limited, the upper limit is that all variables are infected. Therefore, a graph search state exists where all new extended edge tasks have been executed, for Algorithm 1 to terminate.

For a single-source information flow tracking, the worst case is to search all the extended edges and taint all the variables. In this case, the complexity of Algorithm 1 is $O(|E||D|^2)$, where E is the set of program statements and D is the set of program variables. Considering the verification of the function repeat node, and assuming that the average number of statements in the method is k, then the complexity of Algorithm 1 is $O(k|E||D|^2)$. Moreover, if the backward alias analysis is considered, the complexity of Algorithm 1 is $O(k|E||D|^4)$. However, the extended

edge of calling a function is only propagated and analyzed once, because after it is first analyzed, the record is summarized into a hash table *Summaryhash*. When the same extension is encountered later, the function summary can be reused, making the approach highly efficient. Especially, at different call sites of the same method *m*, the summary function can be reused (gaining efficiency).

Regarding the static instrumentation in Algorithm 2, a multilevel hash mapping table $HashMap_{polic}$ is exploited to classify and record all the paths, and these are processed hierarchically by methods. Based on this, the methods of each class are analyzed only once, saving the long preparation and delay in repeatedly accessing them. The corresponding monitoring point is inserted at each branch node to monitor the path state. Because the path length and path set are both limited, Algorithm 2 can terminate. For *u* paths, the average number of branch nodes of a path is denoted as *v*, then the complexity of Algorithm 2 is *O(uv)*, which mainly depends on the number of paths and the number of branch nodes. ■

V. EVALUATION

This section reports on how FSAFlow's function and performance were evaluated. Experiments were performed on a simulator. The host ran Windows 10, and was equipped with an Intel(R) Core (TM) i7-10710U CPU with 16.0 GB RAM. The simulator environment is a Pixel 2 smartphone with Android 10.0, API 29.

A. Function Evaluation

This function evaluation verifies whether the system correctly prevents the release of prohibited information and releases authorized information. Conventional DTA may produce false negatives due to its low code coverage, while FSAFlow may exhibit false negatives due to its implementation limitations during the static analysis stage. We mainly compared FSAFlow with TaintDroid, an important DTA tool, from the perspective of false negatives. Considering that FSAFlow is based on FlowDroid, a well-known STA tool, FlowDroid was also chosen to mainly compare false positive rates because this is a common problem in STA. For the evaluations, implicit flow analysis was only enabled in FSAFlow and FlowDroid, since TaintDroid cannot support it.

1) First, DroidBench 2.0 [57] was exploited to verify function adaptability in mining specific information flow paths. DroidBench 2.0 is an open-source benchmark suite for comparing Android taint-analysis functions.

Since FlowDroid and TaintDroid can only perform analysis but not control, their test results were judged by whether the given path can be found, while the test results of FSAFlow were judged by whether the given path can be intercepted at runtime. All 120 apps in DroidBench 2.0 were used and divided into 13 categories. FSAFlow/FlowDroid defines more sources than DroidBench, such as getLastKnownLocation, etc. To focus on function evaluations, the leakage paths brought by such sources are marked as positive cases. The test results are listed in Table II.

There are 115 leakage paths among all the test cases. Both FlowDroid and FSAFlow found 101 of these paths, but FlowDroid misreported eight paths, while FSAFlow misreported five. The three paths that FlowDroid misreported and FSAFlow did not misreport actually never occur in tests. These are *Unregister1*/*Callbacks*, *Exceptions3*/*General Java*, and *VirtualDispatch3*/*General Java*. FlowDroid does not provide a more precise analysis of the codes in the unregistered-again callback, the non-occurring exceptions, and the never-called factory method. The five paths misreported by FSAFlow involved a coarse-grained array or list tracking, and inaccurate backward alias analysis. To avoid incurring a significant storage overhead, all data items in an array or a list share the same taint tag, which caused four false positives.

TABLE II. CORRECTNESS TEST RESULTS OF INFORMATION FLOW LEAK PATH DETECTION ON DROIDBENCH 2.0

DroidBench 2.0 Category	# leaks	FlowDroid			TaintDroid			FSAFlow		
		TP	FP	FN	TP	FP	FN	TP	FP	FN
Aliasing (1)	0	0	1	0	0	0	0	0	1	0
Arrays and Lists (7)	3	3	4	0	3	4	0	3	4	0
Callbacks (15)	17	17	1	0	17	0	0	17	0	0
Field and Object Sensitivity (7)	2	2	0	0	2	3	0	2	0	0
Inter-App Communication (3)	3	3	0	0	3	0	0	3	0	0
ICC (18)	19	17	0	2	17	0	2	17	0	2
Lifecycle (17)	17	17	0	0	16	0	1	17	0	0
General Java (23)	20	18	2	2	19	0	1	18	0	2
Android-Specific (13)	11	7	0	4	9	0	2	7	0	4
Implicit Flows (4)	8	7	0	1	1	0	7	7	0	1
Reflection (4)	4	1	0	3	4	0	0	1	0	3
Threading (5)	5	3	0	2	5	0	0	3	0	2
Emulator Detection (3)	6	6	0	0	4	0	2	6	0	0
Total	115	101	8	14	100	7	15	101	5	14
Precision=TP/(TP+FP)		92.66%			93.46%			95.28%		
Recall=TP/(TP+FN)		87.83%			86.96%			87.83%		

TABLE III. CORRECTNESS TEST RESULTS OF INFORMATION FLOW LEAK PATH DETECTION FOR THE LOOP STABLE PERIOD

M	N	L	K	#Cases	#Paths	FlowDroid		FSAFlow	
						FN	FP	FN	FP
5	3	7	1	20	20	20	24	0	0
10	8	12	2	20	19	19	31	0	1
15	17	18	3	20	18	18	40	0	2

FlowDroid and FSAFlow missed 14 paths, mainly because some special codes were not tracked, including propagating taints across interleavings of separate components (Singletons1/ ICC), ICC Handler constructs (ServiceCommunication1/I CC), static initialization method (StaticInitialization1/General java), formatter (StringFormatter1/General java), etc.

TaintDroid adopts a coarse-grained tracking strategy similar to FlowDroid, and stores only one taint tag per array/list/ object to minimize the storage overhead. Thus, it misreported seven paths in the tests of Arrays, Lists, and object fields. It missed 15 paths, seven of which were implicit flow paths. TaintDroid does not track control flows and it missed seven implicit flows generated by the control-flow dependency in the *ImplicitFlow* 1-4/*ImplicitFlows* tests. Instead, FSAFlow/FlowDroid tracks implicit flow generated by control dependency. Additionally, TaintDroid cannot track taint propagation in some specific objects, such as Sharedreferences (SharedPreference-Changed1/ lifecycle), or for specific sources such as findViewById (Private DataLeak1-2/AndroidSpecific), etc.

In short, FSAFlow achieved a recall of 87.83% and a precision of 95.28%. FlowDroid also achieved a recall of 87.83%, but only 92.66% for precision. TaintDroid achieved a recall of 86.96% and a precision of 93.46%.

2) The tracking ability during the loop stable period was then tested. DroidBench 2.0 have very few loop structures. Therefore, a java program, *GT*, was written to produce test cases with taint propagation in loop bodies. The input parameters of *GT* included the number *M* of variables (including a

source variable *in* and a sink variable *out*), the number *N* of statements contained in a loop body, the number of loop laps *L* (*L>M*), and the number *K* of loop nesting layers. Each statement was a simple assignment of *x=y* or *x*=0. *GT* first randomly generated schemes of assignment statements in loop bodies, then the schemes with propagation from *in* to *out* in the loop bodies were selected. A test case in the form of a java file was generated according to a selected scheme. The case first obtained the mobile IMEI and then sent the information by SMS after propagating in nested loops. Because *L>M*, all the cases had loop stable periods. Three groups of cases were output. Each group had 20 cases that used the same parameter settings. The test results are listed in Table III. FlowDroid did not track in loop stable periods, causing all leakage paths to be missed and misreported 95 paths that did not actually occur. FSAFlow only misreported three paths and had no false negatives. The exceptions had forms akin to a loop body: {*out=in*, *in*=0;}. Although *out* was infected in the first round, it was cleaned again in the second round. Such loop processing may have little meaning since it rarely appears in practice.

3) Finally, the functions were evaluated on 150 popular real-world apps from HUAWEI and Google's app store. These were randomly selected from 15 different categories. These apps were shown as benign by VirusTotal, an authoritative website that provides malware-analysis services. For the paths found by FlowDroid, FSAFlow, and TaintDroid, since there was no available information about the privacy leakage of these apps, manual horizontal comparison and analysis were conducted according to the app functions, self-declarations, and user comments on the app stores, The test results are listed in Table IV.

Thirty-two apps were found that have potential leakage paths. The main basis of judgment was that these release paths were not identified by the declarations of these apps, and they were not related to the main functions. For example, the *Currency world* app in the Finance category collects user accounts and writes them to a local file without prompting the user. The *Fit fitness* app in the Sports and Health category sends location data to advertising service providers.

All three tools exhibit analysis errors. As for false negatives, TaintDroid missed two leakage paths on *LETV video*, *dragonfly FM* in the Media and Entertainment category because it did not track interface input information (e.g., passwords), while FSAFlow and FlowDroid reported correctly. However, FSAFlow and FlowDroid failed to report one leakage path where the information of dynamic reflection is stored in configuration files, such as *Car Headlines* from the Cars category, while TaintDroid reported correctly. Regarding false positives, the three tools all misreported one leakage path involving arrays in *Baby Read* from the Kids category because of their coarse tracking granularity for arrays. Besides, FlowDroid misreported 1 leak path that will not occur in the *Touch* app of the Communication category. The APK file of this application contains debugging code that does not execute at runtime. FSAFlow could avoid this misreport by runtime verification.

B. Performance Evaluation

FSAFlow's performance was evaluated by comparing it with some representative optimization techniques on DTA. In the evaluations, the implicit flow analysis was enabled in FSAFlow only, since the DTA techniques used for comparison do not support such analysis.

TABLE IV. CORRECTNESS TEST RESULTS OF INFORMATION FLOW LEAK PATH DETECTION ON REAL-WORLD APPS

App Category	# leaks	FlowDroid			TaintDroid			FSAFlow		
		TP	FP	FN	TP	FP	FN	TP	FP	FN
Media & Entertainment(9)	5	5	0	0	3	0	2	5	0	0
Tools(11)	4	4	0	0	4	0	0	4	0	0
Communication(12)	4	4	0	0	4	0	0	4	0	0
Education(8)	1	1	0	0	1	0	0	1	0	0
Books & References(11)	3	3	0	0	3	0	0	3	0	0
Photography (12)	1	1	0	0	1	0	0	1	0	0
Travel & Navigation(17)	5	5	0	0	5	0	0	5	0	0
Shopping(7)	3	3	0	0	3	0	0	3	0	0
Business(10)	1	1	1	0	1	0	0	1	0	0
Kids(13)	4	4	1	0	4	1	0	4	1	0
Finance(13)	2	2	0	0	2	0	0	2	0	0
Sports & Health(4)	4	3	0	1	4	0	0	3	0	1
Lifestyle & Convenience(7)	3	3	0	0	3	0	0	3	0	0
Personalized Themes(10)	1	1	0	0	1	0	0	1	0	0
Cars(6)	1	1	0	0	1	0	0	1	0	0
Total	42	41	2	1	40	1	2	41	1	1

TABLE V. PERFORMANCE TEST RESULTS OF INFORMATION FLOW LEAK PATH DETECTION ON REAL-WORLD APPS

APP Name	Static			Dynamic				
	Flowdroid analysis (ms)	FSAFlow analysis (ms)	FSAFlow instrument (ms)	Baseline (ms)	Cdroid (%)[b]	LIFT (%)[b]	Iodine (%)[b]	FSAFlow (%)[b]
Draft design	6684	6991	2235	2186	14.72	8.68	5.54	1.98
Baisou video	4858	5074	1450	1326	14.57	5.75	2.36	1.28
Dragonfly FM	5223	6133	1735	878	13.77	8.74	5.41	2.76
You health	6514	9713	1866	148	13.53	4.34	2.83	2.22
Learning pass	3003	2957	1155	217	14.55	8.70	5.42	1.23
Currency world	3945	4313	1285	1482	14.02	3.89	3.01	2.20
Car headlines	3902	4247	1542	3476	14.37	6.05	3.07	2.28
Two step outdoor	4135	3711	1482	604	13.26	16.71	9.15	2.62
Wanshun taxi	5162	5359	1475	977	13.36	8.04	4.25	1.24
Leeboo projection	2419	2628	934	1256	14.82	7.38	3.28	2.42
YaoWang	3350	3621	957	2648	14.80	9.41	4.16	2.16
Aikangyue	5443	5911	1569	2799	13.80	14.49	8.31	2.92
Beautiful practice	6889	8829	2251	840	13.02	9.09	5.27	2.76
Da Runfa Youxian	6125	6201	2450	688	14.47	6.24	2.33	2.06
Maka design	3515	4138	1321	624	14.19	11.97	7.00	2.74
Part time cat	6882	7786	2373	2256	13.96	9.51	5.46	2.00
Class suspension bell	6663	6691	1915	1924	13.51	6.88	2.90	2.96
Ease Flower	4827	6272	1788	1619	14.90	7.77	3.17	1.28
Huiwan	5116	6349	1827	3055	13.93	8.81	5.60	1.71
eHi taxi	4041	5985	1214	157	13.65	8.81	5.57	1.58
Micro carp weather	5932	6218	2089	2718	13.38	11.90	7.03	1.36
Settled guest	2583	3213	1029	312	13.27	8.97	4.35	2.27
Guangdong Mobile	7613	8301	2456	1164	13.40	6.11	4.18	2.28
Litchi news	5164	5743	1733	2918	14.15	14.19	8.25	2.88
Change Icon	5527	5771	1777	919	13.08	3.39	2.89	1.33
Good rabbit video	3891	3892	1276	386	14.17	9.78	5.57	2.16
Sound encounter	3233	3838	1119	1680	13.95	9.78	4.96	1.05
WuLi headlines	5434	6222	1698	1533	13.45	12.09	6.35	2.03
Task Wizard	2581	3876	939	879	14.04	9.90	5.48	2.82
LETV video	4151	5091	1609	933	14.28	15.72	8.91	1.37
Shantao Street	6749	7734	2385	1230	13.92	8.35	5.07	2.54
Walker	6146	6351	1824	189	13.56	8.57	4.39	1.57
Average	4928	5599	1649	-	13.93%	9.06	5.05%	2.06%

[b] (%)=(a-b)/b, where a=Corresponding DTA time, b=Baseline time.

1) Evaluation on real-world apps. The selected test cases are 32 apps with potential leakage paths from evaluation A-3).

We compared FSAFlow's static analysis with FlowDroid, and compared FSAFlow's runtime control with Tracking On Demand (TOD), Local Code Optimization (LCO), Optimal Hybrid Analysis (OHA) and native Android system (Baseline). LIFT [47] was chosen as the TOD reference. LIFT checks whether all live-in/out variables are safe before tracking a basic block. If so, it runs the basic block without any tracking; otherwise, it runs the basic block with conventional tracking. Since LIFT's implementation does not support Android, a simulation analysis was conducted on the best case of LIFT. In this case, LIFT did not spend any time determining whether basic blocks need to be tracked, but made quick operations directly on the basic blocks that had been inserted. In our experiment, it was manually determined whether there are already contaminated variables in a basic block before the execution enters this basic block; if so, we inserted tracking instructions into the basic block. CDroid [21] was chosen as an

LCD reference, and it deploys local code optimization methods on hot traces, including reductant taint load/store elimination, reductant taint compute elimination, taint load hoisting and taint store sinking. Iodine [9] was chosen as an OHA reference. In the static analysis stage, Iodine only inserts tracking instructions for the statements that change the taint state of expected executions rather than all executions. Iodine realized rollback-free recovery from unexpected executions for its DTA to be sound. Since Iodine's implementation also does not support Android, simulation analysis was conducted on the best case of Iodine. In this case, let Iodine execute the leakage path as a fast path without triggering a tracking failure/recovery mechanism. According to Iodine's principle for fast path optimization, we only inserted tracking instructions for the statements whose source operands could be tainted by the taint source in forward STA, and destination operands could reach the sink in backward STA along the leakage path.

In these test cases, different operations were performed for different applications to trigger the corresponding paths and test their execution time. For instance, in shopping apps, the leakage path of a payment password could be triggered by clicking the confirm button; in communication apps, the leakage path of the chat messages could be triggered by clicking the send button. Twenty tests were performed on each app, and the average execution time was recorded. The test results are listed in Table V.

FSAFlow incurred a slightly higher overhead in static analysis than FlowDroid, which indicates that FSAFlow takes less time to extract key additional path information. FSAFlow needed less time in static analysis than in static instrumentation, mainly because only the key nodes of a few paths were instrumented with monitoring pointers. As for runtime tracking, compared with native Android, the average overhead of CDroid, LIFT, Iodine and FSAFlow respectively was 13.93%, 9.06%, 5.05% and 2.06%. Both LIFT and Iodine have lower overhead than CDroid. Although the test result of Iodine is closest to that of FSAFlow, it is obtained in Iodine's best case, that is, let Iodine execute the leak path as a fast path without triggering a failure/recovery mechanism during the experiment. Note that the overheads of LIFT and Iodine fluctuated, because of the diverse sizes of flow statements contained in the different paths they encountered. In contrast, FSAFlow had the lowest overhead with good robustness, which is attributable to the branch statements accounting for a small proportion of statements on most paths. Thus, the code execution of the corresponding monitoring point had little impact on the program execution time.

2) Evaluation on the CaffeineMark 3.0 benchmark. This benchmark uses a scoring metric to measure various aspects of system performance. The test on CaffeineMark 3.0 is scored by dividing the number of executed cases by the time taken to execute all the cases. Its six test cases that can run on Android were modified, and all were computationally intensive. The test objective was to obtain the overhead of FSAFlow when the CPU performed intensive sensitive information flow operations. Specifically, the initially assigned variable was set as the source node, and the output result variable was set as the sink node to perform path tracking. LIFT and Iodine were simulated using the method in 1). The test results are shown in Figure 9.

The respective scores of the native Android system, CDroid, LIFT, Iodine, and FSAFlow are 5975 (S_B), 4965 (S_{O1}), 5190 (S_{O2}), 5431 (S_{O3}), and 5652 (S_{O4}). Compared with the native Android, the average overhead of CDroid, LIFT, Iodine and FSAFlow respectively are 16.90%, 13.14%, 9.10% and 5.41% (overhead percentage = $(S_B\text{-}S_{Oi})/S_B$). The results in Figure 9 show that the overhead of FSAFlow caused by using loop and logic cases is higher than that of other cases. This may be due to the code structure of loop and logic encountering more branch nodes, which increases path state monitoring. CDroid incurs higher overhead in *string* cases than other cases. This is probably due to the additional memory comparisons for string objects in method prototypes. Iodine and Lift have higher overhead in *method* cases than in other cases, which may be caused by more taint propagation statements in the target path in *method* cases. The score of FSAFlow is close to that of the native Android system, proving that our tracking method is more efficient.

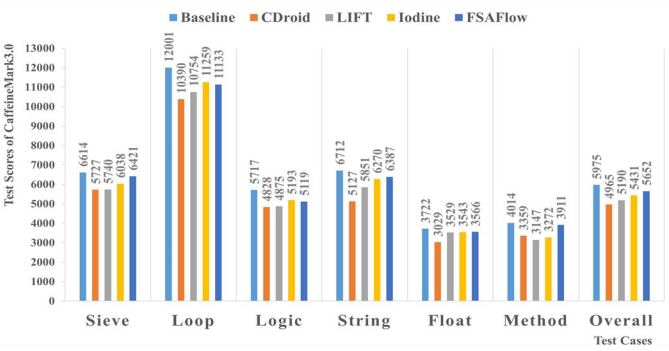

Fig. 9. Performance Test Scores of CaffeineMark 3.0

TABLE VI. PERFORMANCE TEST RESULTS WITH AND WITHOUT IMPLICIT FLOW ANALYSIS ENABLED ON REAL-WORLD APPS & MALWARE

App Groups	Static Analysis				Static Instrument		Runtime Control	
	Enable		*Disable*					
	Time(s)	*Leaks*	*Time(s)*	*Leaks*	*Enable(s)*	*Disable(s)*	*Enable(s)*	*Disable(s)*
Real-word APP	5.599	42	4.451	42	1.649	1.649	1.405	1.405
BaseBridge	3.971	47	3.185	45	3.337	2.661	1.347	1.346
KMin	3.487	39	2.798	36	3.052	2.237	1.321	1.318
Geinimi	4.002	43	3.214	41	3.210	2.559	1.373	1.371
DroidDream	4.995	54	4.017	50	4.005	3.091	1.398	1.391

The results of Evaluation 1) and 2) show that FSAFlow has better performance than several representative optimization approaches, such as Iodine (Hybrid Analysis), LIFT(Tracking On Demand), and CDroid (Local Code Optimization).

3) Evaluation of implicit flow analysis. The test is divided into five groups. The first group consists of 32 apps in Table V, and each remaining group consists of eight randomly selected samples from the same family in the malware dataset Drebin [66]. Comparative experiments were conducted with and without implicit flow enabled. Ten tests were performed on each group, and the average execution time was recorded. The final test results are listed in Table VI.

The results show that adding implicit flow analysis may add insignificant overhead to FSAFlow. Moreover, the number of leak paths output by FSAFlow increases minimally. The main reasons are as follows: (a) The static IFDS algorithm performs width-first traversal to access the branch structure information, thus rapidly calculating the taint propagation caused by the control dependencies. (b) In IFDS, the control-dependent taint propagation in conditional branches will not occur if independent variables are not tainted. (c) Furthermore, in these test cases, the tainted variables were found to appear rarely in conditional jump statements, indicating that the sensitive information flows seldom affect the control behaviors of real apps. (d) Even in the case of some malware attempting to convey sensitive information through implicit flows, to hide itself, the number of in-

structions involved in the leakage path is very small, resulting in the cost of FSAFlow tracking the small fragment of code controllable.

VI. Related Work

In this section, the previous research on DTA, DTA optimization, and hybrid analysis (HA) are introduced. The limiting factors are also discussed, casting the problem in terms of DTA optimization and proposing a new methodology based on HA.

A. Dynamic Taint Analysis

There has been considerable research on DTA systems [19], [28], [30], [33], [35], [53]-[56]. The time overhead of Dytan for data-flow based propagation alone was approximately 30x, whereas the overhead imposed by control- and data-flow based propagation was approximately 50x [33]. Panorama was 20 times slower on average [28].

B. DTA optimization

Past work has developed many optimized DTA techniques. Important paths explored by previous research include:

Hardware support: The hardware typically consists of logic blocks that monitor the execution of each instruction in the processor and keep track of the tag information flowing from the execution unit at every cycle [5], [38]-[41]. Many hardware acceleration schemes require additional non-standard commodity components or a redesign of the entire processor core, which limits the practicality of these approaches.

Parallelization: The proliferation of multicore systems has inspired researchers to decouple taint tracking logic to spare cores to improve performance [36], [37], [44]-[46], [65]. ShadowReplica [65] spawns a secondary shadow thread from the original process to run DTA on spare cores in parallel. ShadowReplica performs a combined offline dynamic and static analysis of the application to minimize the data that must be communicated for decoupling the analysis. However, ShadowReplica's performance improvement achieved by this "primary & secondary" thread model is fixed and cannot be improved further when more cores are available [37]. TaintPipe [37] spawns multiple threads or processes in different stages of a pipeline to carry segmented symbolic taint analysis in parallel. Its pipeline style relies on straight-line code with very few runtime values, enabling lightweight online logging and a significantly lower runtime overhead. These efforts reduce the latency of DTA through parallelization, but also require OS or hardware level support, which limits commercial applications.

Intermittent tracking： Performance is improved by dynamically deactivating tracking under certain conditions, e.g., when the program does not operate on tagged data [41], [47], [48] or when it is not handling risky tasks (reading network data) [49], [51]. Data tracking can also be applied on-demand based on CPU usage, or by manual activation [50]. The disadvantage of this kind of research work is that it fails to determine the timing of dynamic opening or closing. Non-professional users may suffer from privacy leakage when turning off taint tracking. Performance loss can also be incurred by the dynamic switch.

Code Optimization： Frequently executed taint logic code incurs substantial overhead. The work in [31] developed function summaries to track taint at the function level. Jee et al. [52] proposed Taint Flow Algebra to abstract and optimize taint logic for basic blocks. In [13] several lightweight taint propagation optimization methods were deployed on hot traces. Taint Rabbit [68] employed a JIT compiler to optimize generic taint analysis. Basic blocks without operating taints were not tracked, while the frequently executed basic blocks with operating taints were further subdivided to only track sub-blocks that operate taints. For software vulnerability analysis, Neutaint [15] exploited machine learning methods to model the reachability from hot bytes in the input to the sink nodes. Such vulnerability can be exploited and attacked using the hot byte. In privacy protection, privacy data such as phone numbers are often sent to the sink in entirety. Often, the status of hot bytes is equivalent, which may be not suitable for privacy protection. Code optimization can only be realized for specific code features, and their effect is limited.

C. Hybrid analysis

Hybrid analysis (HA) uses static analysis to narrow the scope of code pieces to be examined at runtime and then perform dynamic analysis on them. Many well-known STA tools [57]-[59] can be exploited by hybrid analysis to complete pre-optimized tracking logic [9]-[10], [52], [60], pre-reduced tracking range [10], [61], and share some tracking tasks [62]. While [10] used sound STA conservatively to reduce dynamic overheads. Iodine [9] and OHA [60] further reduced runtime overheads using unsound and predicated static analysis. They provided a recovery mechanism to handle any potential unsoundness in speculative execution. In OHA, the program execution was replayed from the beginning and analyzed, which is not feasible for online security analysis of live executions [9]. To solve this problem, Iodine exploited constraint predicated static analysis to achieve forward recovery. To optimize tracking, Iodine inserted monitoring points only at the statements that can change the taint state of variables when the fast-path is executed. However, if the execution probabilities of fast- and slow-paths are equal in the program to be analyzed, or the profiling of whether the path is fast or slow is inaccurate, they exhibit no advantages, conversely, considerable path switching and recovery overheads will be incurred. The current hybrid analysis shows that the DTA retains several dynamic single-step taint tracking instructions. The extant hybrid analysis methods only have advantages in some specific situations. The proposed FSAFlow differs from these methods since FSAFlow realizes a complete separation of the slow one-by-one tracking logic and dynamic operation in a program, and is continuously efficient for Android privacy protection through path monitoring.

HA also has many practical applications. As a vulnerability protection tool, DynPTA [67] uses scoped byte-level DTA to narrow the range of objects identified for static pointer analysis. DynPTA only optimizes DTA specifically in loop optimizations for array accesses. According to the results of pointer analysis, the selective data is encrypted to prevent external attacks from exploiting the vulnerability of pointer misuse to obtain sensitive data. FSAFlow's purpose differs from that of DynPTA. FSAFlow optimizes the performance of taint tracking and further prevents untrusted programs from actively leaking private information.

VII. Conclusion

A novel approach, FSAFlow, is proposed to perform a hy-

brid analysis for privacy protection. This solves the key overhead problem of applying DTA. The central concept of FSAFlow is to completely separate the one-by-one tracking logic and the program execution and perform tracking control based on the path instead of the taint. FSAFlow manages the path state through FSA. Because the state transition of the path mainly depends on the branch statements rather than all the statements, FSAFlow runs efficiently. The theoretical and experimental analysis proves that implementing FSAFlow is rational and efficient.

VIII. ACKNOWLEDGEMENT

We would like to thank Herbert Bos and the anonymous reviewers for their insightful comments. This work is supported by the National Natural Science Foundation of China (No. 62176265).

REFERENCES

[1] Z. B. Celik, E. Fernandes, E. Pauley, G. Tan and P. McDaniel, "Program analysis of commodity IoT applications for security and privacy: challenges and opportunities," *ACM Computing Surveys*, vol.52, pp.1-30, Sept. 2019.

[2] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang *et al.*, "Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS," *2016 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA, 2016, pp.357-376.

[3] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Transactions on Software Engineering and Methodology*, vol.9, pp.1557-7392, Oct. 2000.

[4] V. P. Ranganath and J. Mitra, "Are free Android app security analysis tools effective in detecting known vulnerabilities?" *Empirical Software Engineering*, vol.25, pp.178-219, Jan. 2020.

[5] N. Vachharajani, M. J. Bridg, J. Cha, R. Ranga, G. Otton, J. A. Blome *et al.*, "RIFLE: an architectural framework for user-centric information-flow security," *37th International Symposium on Microarchitecture (MICRO-37'04)*, Portland, Oregon, 2004, pp.243-254.

[6] Sufatrio, D. J. J. Tan, T. W. Chua and V. L. L. Thing, "Securing Android: a survey, taxonomy, and challenges," *ACM Computing Surveys*, vol.47, pp.1-45, July 2015.

[7] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. Singh Gaur, M. Conti and M. Rajarajan, "Android security: a survey of issues, malware penetration, and defenses," *IEEE Communications Surveys & Tutorials*, vol.17, pp.998-1022, Secondquarter 2015.

[8] Y. Nan, Z. Yang, X. Wang, Y. Zhang, D. Zhu and M. Yang, "Finding clues for your secrets: semantics-driven, learning-based privacy discovery in mobile apps," *in Proceedings of the Network and Distributed System Security Symposium(NDSS'18)*, 2018.

[9] S. Banerjee, D. Devecsery, P. M. Chen and S. Narayanasamy, "Iodine: fast dynamic taint tracking using rollback-free optimistic hybrid analysis," *2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2019, pp.490-504.

[10] M. Zhang, H. Yin, "Efficient, Context-Aware Privacy Leakage Confinement for Android Applications without Firmware Modding," *in Proceedings of the 9th ACM symposium on Information, computer and communications security (ASIA CCS '14)*, Kyoto, Japan, 2014, pp.259-270.

[11] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao and A. Prakash, "ContexIoT: Towards providing contextual integrity to appified IoT platforms," *in Proceedings of the Network and Distributed Systems Symposium (NDSS'17)*, 2017.

[12] W. Xu, S. Bhatkar and R. Sekar, "Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks," *in Proceedings of the 15th conference on USENIX Security Symposium (USENIX-SS'06)*, USA, 2006.

[13] Z. Wu, X. Chen, Z. Yang and X. Du, "Reducing security risks of suspicious data and codes through a novel dynamic defense model," *in IEEE Transactions on Information Forensics and Security*, vol.14, pp.2427-2440, Sept. 2019.

[14] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang and K. Chen, "FuzzGuard: filtering out unreachable inputs in directed grey-box fuzzing through deep learning," *in 29th USENIX Security Symposium*, 2020.

[15] D. She, Y. Chen, A. Shah, B. Ray and S. Jana, "Neutaint: efficient dynamic taint analysis with neural networks," *2020 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2020, pp.1527-1543.

[16] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley and D. Evans, "Automatically hardening web applications using precise tainting," *IFIP International Information Security Conference*, Boston, MA, vol.181, pp.295-307, 2005.

[17] V. Ganesh, T. Leek and M. Rinard, "Taint-based directed whitebox fuzzing," *2009 IEEE 31st International Conference on Software Engineering*, Vancouver, BC, Canada, 2009, pp.474-484.

[18] S. Lekies, B. Stock and M. Johns, "25 million flows later: large-scale detection of DOM-based XSS," *in Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS '13)*, New York, NY, USA, 2013, pp.1193-1204.

[19] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," *in Proceedings of the Network and Distributed System Security Symposium (NDSS '05)*, 2005.

[20] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan and O. Weisman, "TAJ: effective taint analysis of web applications," *in Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, New York, NY, USA, 2009, pp.87-97.

[21] Z. Wu, X. Chen, X. Du and Z. Yang, "CDroid: practically implementation a formal-analyzed CIFC model on Android," *Computers & Security*, vol.78, pp.231-244, 2018.

[22] G. E. Suh, J. W. Lee, D. Zhang and S. Devadas, "Secure program execution via dynamic information flow tracking," *in Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (ASPLOS XI)*, New York, NY, USA, 2004, pp.85-96.

[23] J. Kong, C. C. Zou and H. Zhou, "Improving software security via runtime instruction-level taint checking," *in Proceedings of the 1st workshop on Architectural and system support for improving software dependability (ASID '06)*, New York, NY, USA, 2006, pp.18-24.

[24] W. G. J. Halfond, A. Orso and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks," *in Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (SIGSOFT '06/FSE-14)*, New York, NY, USA, 2006, pp.175-185.

[25] T. Pietraszek and C.V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," *Springer*, Berlin, Heidelberg, 2005, pp.124-145.

[26] V. Haldar, D. Chandra and M. Franz, "Dynamic taint propagation for Java," *21st Annual Computer Security Applications Conference (ACSAC'05)*, Tucson, AZ, USA, 2005.

[27] P. Vogt, F. Nentwich, N. Jovanovic and E. Kirda, "Cross site scripting prevention with dynamic data tainting and static analysis," *in Proceedings of the Network and Distributed System Security Symposium (NDSS '07)*, 2007.

[28] H. Yin, D. Song, M. Egele, C. Kruegel and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," *in Proceedings of the 14th ACM conference on Computer and communications security (CCS '07)*, New York, NY, USA, 2007, pp.116-127.

[29] J. Caballero, P. Poosankam, S. McCamant, D. Babi ć and D. Song, "Input generation via decomposition and re-stitching: finding bugs in Malware," *in Proceedings of the 17th ACM conference on Computer and communications security (CCS '10)*, New York, NY, USA, 2010, pp.413-425.

[30] W. Enck, P. Gilbert, B. G. Chun, L. P. Cox, J. Jung, P. McDaniel and A. N. Sheth, "TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones," *Communications of ACM*, vol.57, pp.99-106, March 2014.

[31] D. Zhu, J. Jung, D. Song, T. Kohno and D. Wetherall, "TaintEraser: protecting sensitive data leaks using application-level taint tracking," *ACM SIGOPS Operating Systems Review*, vol.45, pp.142-154, Feb. 2011.

[32] M. G. Kang, S. McCamant, P. Poosankam and D. Song, "DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation," *in Proceedings of the Network and Distributed System Security Symposium (NDSS '11)*, 2011.

[33] J. Clause, W. Li and A. Orso, "Dytan: a generic dynamic taint analysis framework," *in Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA '07)*, New York, NY, USA, 2007, pp.196-206.

[34] T. Reps, S. Horwitz and M. Sagiv. "Precise interprocedural dataflow analysis via graph reachability," *in Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '95)*, New York, NY, USA, 1995, pp.49-61.

[35] W. Cheng, Qin Zhao, Bei Yu and S. Hiroshige, "TaintTrace: efficient flow tracing with dynamic binary rewriting," *11th IEEE Symposium on Computers and Communications (ISCC'06)*, Cagliari, Italy, 2006, pp.749-754.

[36] J. Ming, D. Wu, J. Wang, G. Xiao and P. Liu, "StraightTaint: decoupled offline symbolic taint analysis," *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Singapore, 2016, pp.308-319.

[37] J. Ming, D. Wu, G. Xiao, J. Wang and P. Liu, "TaintPipe: pipelined symbolic taint analysis," *in Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*, USA, 2015, pp.65-80.

[38] J. Lee, I. Heo, Y. Lee and Y. Paek, "Efficient dynamic information flow tracking on a processor with core debug interface," *in Proceedings of the 52nd Annual Design Automation Conference (DAC '15)*, New York, NY, USA, 2015, pp.1-6.

[39] G. Venkataramani, I. Doudalis, Y. Solihin and M. Prvulovic, "FlexiTaint: a programmable accelerator for dynamic taint propagation," *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, Salt Lake City, UT, USA, 2008, pp.173-184.

[40] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry *et al.*, "Flexible hardware acceleration for instruction-grain program monitoring," *2008 International Symposium on Computer Architecture*, Beijing, China, 2008, pp.377-388.

[41] E. Bosman, A. Slowinska and H. Bos, "Minemu: the world's fastest taint tracker," *Springer*, Berlin, Heidelberg, 2011.

[42] O. Ruwase, "Parallelizing dynamic information flow tracking," *in Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures (SPAA '08)*, New York, NY, USA, 2008, pp.35-45.

[43] InsecureBankv2. https://github.com/dineshshetty/Android-InsecureBankv2

[44] E. B. Nightingale, D. Peek, P. M. Chen and J. Flinn, "Parallelizing security checks on commodity hardware," *in Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS XIII)*, New York, NY, USA, 2008, pp.308-318.

[45] A. Quinn, D. Devecsery, P. M. Chen and J. Flinn, "JetStream: cluster-scale parallelization of information flow queries," *in Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'16)*, USA, 2016, pp.451-466.

[46] S. Ma, X. Zhang and D. Xu, "ProTracer: towards practical provenance tracing by alternating between logging and tainting," *in Proceedings of the Network and Distributed System Security Symposium (NDSS '16)*, 2016.

[47] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou and Y. Wu, "LIFT: a low-overhead practical information flow tracking system for detecting security attacks," *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Orlando, FL, USA, 2006, pp.135-148.

[48] A. Ho, M. Fetterman, C. Clark, A. Warfield and S. Hand, "Practical taint-based protection using demand emulation," *in Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*, New York, NY, USA, 2006, pp.29-41.

[49] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim *et al.*, "RAIN: refinable attack investigation with on-demand inter-process information flow tracking," *in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, New York, NY, USA, 2017, pp.377-390.

[50] G. Portokalidis and H. Bos, "Eudaemon: involuntary and on-demand emulation against zero-day exploits," *in Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Eurosys '08)*, New York, NY, USA, 2008, pp.287-299.

[51] S. Moore and S. Chong, "Static Analysis for Efficient Hybrid Information-Flow Control," *2011 IEEE 24th Computer Security Foundations Symposium*, Cernay-la-Ville, France, 2011, pp.146-160.

[52] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh and A. D. Keromytis, "A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware," *in Proceedings of the Network and Distributed System Security Symposium (NDSS '12)*, 2012.

[53] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti and A. Prakash, "FlowFence: practical data protection for emerging IoT application frameworks," *in Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*, USA, 2016, pp.531-548.

[54] M. Sun, T. Wei and J. C.S. 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 (CCS '16)*, New York, NY, USA, 2016, pp.331-342.

[55] V. P. Kemerlis, G. Portokalidis, K. Jee and A. D. Keromytis, "Libdft: practical dynamic data flow tracking for commodity systems," *in Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE '12)*, New York, NY, USA, 2012, pp.121-132.

[56] Triton: A Dynamic Symbolic Execution Framework. SSTIC, 2015.

[57] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein *et al.*, "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, New York, NY, USA, 2014, pp.259-269.

[58] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel and A. S. Uluagac, "Sensitive information tracking in commodity IoT", *in Proceedings of the USENIX Security Symposium*, 2018, pp.1687-1704.

[59] F. Wei, S. Roy, X. Ou and Robby, "Amandroid: a precise and general inter-component data flow analysis framework for security vetting of Android apps," *in Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*, New York, NY, USA, 2014, pp.1329-1341.

[60] D. Devecsery, P. M. Chen, J. Flinn and S. Narayanasamy, "Optimistic hybrid analysis: accelerating dynamic analysis through predicated static analysis," *in Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*, New York, NY, USA, 2018, pp.348-362.

[61] W. Chang, B. Streiff and C. Lin, "Efficient and extensible security enforcement using dynamic data flow analysis," *in Proceedings of the 15th ACM conference on Computer and communications security (CCS '08)*, New York, NY, USA, 2008, pp.39-50.

[62] P. Saxena, R Sekar and V. Puranik, "Efficient fine-grained binary instrumentation with applications to taint-tracking," *in Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization (CGO '08)*, New York, NY, USA, 2008, pp.74-83.

[63] L.Cavallaro, P. Saxena and R. Sekar, "On the limits of information flow techniques for malware analysis and containment," *in Proceedings of the International conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Berlin, Heidelberg, 2008, pp.143-163.

[64] Asia Slowinska, Herbert Bos, "Pointless Tainting? Evaluating the Practicality of Pointer Tainting," *in Proceedings of the 4th ACM European conference on Computer systems (EuroSys '0)*, New York, NY, USA, 2009, pp.61-74.

[65] K. Jee, V. P. Kemerlis, A. D. Keromytis and G. Portokalidis, "ShadowReplica: efficient parallelization of dynamic data flow tracking," *in Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security*, 2013, pp.235-246.

[66] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your

pocket," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium* (*NDSS*). 2014. pp.1-15.

[67] T. Palit, J. F. Moon, F. Monrose and M. Polychronakis, "DynPTA: Combining Static and Dynamic Analysis for Practical Selective Data Protection," *in Proceedings of 2021 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, US, 2021, pp.1919-1937.

[68] J. Galea and D. Kroening, "The taint rabbit: Optimizing generic taint analysis with dynamic fast path generation," *in Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp.622-636.

[69] J. Zhang, C. Tian and Z. Duan, "FastDroid: Efficient Taint Analysis for Android Applications," *In 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019, pp.236-237.

[70] X. Zhang, X. Wang, R. Slavin and J. Niu, "ConDySTA: Context-Aware Dynamic Supplement to Static Taint Analysis," *in Proceedings of 2021 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, US, 2021, pp.796-812.