TABLE 3
Results on the Apps of DroidBench 2.0 in Dataset S1

App Category	#apps	GTP	DroidSafe			JN-SAF			μDep		
			TP	FP	FN	TP	FP	FN	TP	FP	FN
Aliasing	1	0	0	0	0	0	1	0	0	0	0
Arrays and Lists	7	3	3	4	0	0	4	3	3	4	0
Callbacks	15	17	17	6	0	10	3	7	17	6	0
Field and Obj Sens	7	2	2	2	0	2	0	0	2	2	0
IAC	3	8	8	12	0	4	0	4	8	12	0
ICC	18	24	24	5	0	17	0	7	24	5	0
Lifecycle	17	17	17	4	0	10	0	7	17	4	0
General Java	23	20	20	8	0	11	3	9	20	8	0
Misc Android-Specific	12	11	9	2	2	7	0	4	9	2	2
Implicit Flows	4	8	2	0	6	0	0	8	2	0	6
Reflection	4	4	4	0	0	1	0	3	4	0	0
Threading	5	5	4	1	1	4	0	1	4	1	1
Emulator Detection	3	6	0	0	6	4	0	2	0	0	6
Total	119	125	110	44	15	70	11	55	110	44	15

but only one `LOGI` releases sensitive data. Because μDep conservatively takes both native methods as sinks due to their control-flow correlation with `LOGI` and does not consider the sensitivity of data delivered into `LOGI`, μDep then reaches a false positive.

False Negatives of Native-Code-Involved Flows. As we explained earlier, DroidSafe is incapable of detecting any sensitive flows in NativeFlowBench. Thus, all the ground-truth flows are false negatives of DroidSafe, including the native-code-involved flow in our motivating example in Fig. 1. Meanwhile, JN-SAF also misses the native-code-involved flow in Fig. 1. Our investigation indicates that JN-SAF's implementation issue makes its heap manipulation summary imprecise on this example. In contrast, μDep's mutation-based dynamic analysis figures out better summaries for more complicated memory accesses. As a comparison, the false negatives reported by μDep are all due to its inability to deal with the native Activity components, as demonstrated by apps `native_pure`, `native_pure_direct`, and `native_pure_direct_customized` in Table 2.

Differences in Detecting Native-Code-Uninvolved Flows. Since none of the apps in Table 3 contains native code and μDep is built upon DroidSafe, the two tools report identical results. Next, we compare μDep with JN-SAF. According to Table 3, we observe that μDep and JN-SAF have diverse behaviors. In most app categories of DroidBench, μDep reports both more true positives and false positives than JN-SAF, while JN-SAF reports more false negatives. We infer the reason for such difference is that JN-SAF's summary-based bottom-up data-flow analysis is specialized in the inter-language data-flow analysis but is limited in propagating the points-to information to the callee. In other words, we believe μDep achieves better context sensitivity than JN-SAF. Therefore, JN-SAF cannot achieve similar effectiveness compared with the state-of-the-art context-, object- and field-sensitive interprocedural data-flow analysis, e.g., [11], [13]. Besides, merging the detection results of DroidSafe and JN-SAF to achieve better coverage is not trivial since there may be contradictory detection results from them on native-code-involved flows. In contrast, μDep is integrated into DroidSafe, which prevents this possibility.

TABLE 4
Metrics Comparison of Different Tools on Dataset S1

Tool	Precision(%)	Recall(%)	F1(%)
DroidSafe	71.6	75.0	73.3
JN-SAF	87.6	62.2	72.7
μDep	74.3	87.8	80.5

5.2 Capability of μDep Compared With Another Dynamic Taint Analysis

μDep has a dynamic phase to generate the data dependencies for native methods, indicating that the incompleteness of dependencies derived by our dynamic analysis is a source of false negatives. Therefore, we evaluate the code coverage of native code during our mutation-based dependency generation. We use static binary instrumentation with Dyninst [29] to profile the basic blocks of the shared object files reached by our dynamic analysis. The code coverage of the native code is measured by

$$\frac{\#reached\,basic\,blocks}{\#basic\,blocks} \times 100\%$$

For the test cases with native code in dataset S1, each native method has 6.7 basic blocks on average. We show the code coverage in Table 5. Besides the 11 apps of NativeFlowBench reporting instrumentation I/O errors and 119 apps of DroidBench containing no native code, the code coverage on the rest of the apps is 53.1% on average (29.2%$\sim$100%) under the default configuration of μDep. Several cases have low coverage rates and certain control-flow branches are missed, but the branch misses have not triggered any false negative in the later static taint analysis of μDep.

To compare the capability of taint analysis with the state-of-the-art dynamic taint analysis systems, e.g., NDroid [7], [30], we label the test cases in dataset S1 with the scenarios (*P1*$\sim$*P6*) of information leakages defined in Fig. 3 of [30]. Each scenario depicts one pattern of whether and how

TABLE 5
Information Leakage Scenarios of Dataset S1 and Code Coverage of Native Code

App Name	Scenario	Coverage(%)
native_source	*P3*	57.1
native_nosource	No leak	100.0
native_source_clean	No leak	100.0
native_leak	*P1*	N/A
native_leak_array	*P1*	N/A
native_leak_dynamic_register	*P1*	N/A
native_dynamic_register_multiple	*P1*	62.5
native_noleak	No leak	100.0
native_noleak_array	No leak	N/A
native_method_overloading	*P1*	N/A
native_multiple_interactions	*P1+P4*	N/A
native_multiple_libraries	*P1*	N/A
native_complexdata	*P1*	70.0
native_complexdata_stringop	*P1*	N/A
native_heap_modify	*P3*	68.0
native_set_field_from_native	*P3*	48.7
native_set_field_from_arg	*P2*	100.0
native_set_field_from_arg_field	*P2*	100.0
native_pure* (3 apps)	*P6*	N/A
icc_javatonative	*P1*	29.2
icc_nativetojava	*P2*	80.6
example_fig1	*P2+P5*	100.0
DroidBench (119 apps)	*P5*/No leak	No *.so*
Avg.	–	53.1

information flow crosses the native and Java contexts. For example, P2 represents the scenario where sensitive flows start from the Java context, then switch to the native context, and eventually return to the Java context. Our labeling results of test cases are listed in Table 5. Based on our labeling, considering the TP/FP/FN presented in Tables 2 and 3, we can see that μDep is able to reach high precision and high recall for test cases of scenario *P1*∼*P5*. The only exception is scenario *P6*, which represents the case that the sensitive flow stays at the native side. For such native-only sensitive flows, if they are implemented by native Activity, μDep will report only false negative, as demonstrated by the three native Activity cases in Table 2. Otherwise, our static binary analysis will deem the native methods containing such flows as both sources and sinks, but it cannot tell the exact locations of the sources or sinks in the native method. Because we failed to adjust the sources and sinks of NDroid with the source/sink list of μDep, we are unable to give a fair comparison of the metrics with ground truths.

5.3 Improvement of μDep Upon DroidSafe

μDep is built upon the backend of DroidSafe and its ADI model. Based on the precision and recall metrics presented in Section 5.1, we evaluate how μDep can improve DroidSafe by introducing native code analysis and new stub generation. Furthermore, we apply μDep to understand how shared object files are involved in sensitive information flows in real-world apps and malware.

Our experiments are conducted on dataset S2 and S3. The number of new sensitive flows introduced by native code is considerable, as illustrated in the first three columns of Table 6. For the app in S2, DroidSafe can detect 5,052 sensitive flows while μDep can detect 6,427, with an increase rate of 27.2%. For the app in S3, DroidSafe detects 14,330 sensitive flows, and μDep can detect 16,920, with an increase rate of 18.1%. We are aware that the incremental component contains false positives. Therefore, we hope to evaluate the true positive increase rates. However, due to the difficulty in obtaining the ground truths of real-world apps, we can only use the precision yielded on S1 to estimate the increase rates of true positives on S2 and S3. In detail, with the precision of DroidSafe being 71.6%, we estimate that DroidSafe detects 3,617 and 10,260 true positives for S2 and S3, respectively. In comparison, we estimate that μDep detects 4,775 and 12,571 true positives, with increase rates being 32.0% and 22.5% for S2 and S3, respectively. In fact, we have considered relying on runtime information to collect ground truth for dataset S2 and S3, which turned out to be challenging. On one hand, determining if a profiled execution trace carries sensitive data-flows incurs manual efforts, which is unscalable, while relying on any automated tool would introduce inaccuracy to the ground truth. On the other hand, the collected ground truth is likely to be incomplete, which reduces the credibility of the ground truth. Therefore, we leave the ground-truth collection from runtime information as future work.

TABLE 6
Number of Sensitive Flows Detected by Different Approaches

	#flows (DroidSafe)		#flows (μDep)		#flows (JN-SAF)	
Dataset	Total	Avg.	Total	Avg.	Total	Avg.
S2	5,052	8.28	6,427	10.50	497	2.54
S3	14,330	94.28	16,920	111.32	635	4.44

To figure out the characteristics of sensitive flows caused by native code, we classify and rank the sensitive flows in the apps of S2 and S3 detected by μDep and compare with the respective numbers of flows detected by DroidSafe. The top-10 categories of sensitive data-flows detected by μDep are listed in Table 7. We know the most common sensitive flows are in the category `IO⇝IO`, no matter whether we consider the behavior of native code. An example is reading the content of a file and writing it to another file. There are three categories of sensitive flows increasing enormously after we consider analyzing the native code, i.e., `FILE_INFORMATION⇝IO`, `LOCATION⇝NETWORK`, and `UNIQUE_IDENTIFIER⇝NETWORK`. We investigate the reason for such increases. We found that the increase in `FILE_INFORMATION⇝IO` flows highly depends on the native libraries of information pushing (libjcore.so, libcocklogic.so, libbdpush.so), map/location (libBaiduMapSDK.so, libtencentloc.so, liblocSDK.so), image processing (libgifimage.so, libwebpbackport.so), and exception report (libBugly.so, libBugtags.so). The increase in `LOCATION⇝NETWORK` flows frequently depends on the native library of SQLite storage (libsqlc-native-driver.so). The increase in `UNIQUE_IDENTIFIER⇝NETWORK` flows depends on the native library of privilege promotion (libandroidterm.so). Also, some unknown native libraries, e.g., libopenterm.so, are frequently involved in the apps with these sensitive flows.

TABLE 7
Top Ranking Sensitive Flows Detected by μDep Compared With DroidSafe

source⇝sink	#flows (DroidSafe)	#flows (μDep)	Increasing rate
IO⇝IO	2,128	2,209	3.81%
FILE_INFORMATION⇝IO	486	1,608	230.86%
NETWORK⇝NETWORK	920	1,040	13.04%
IO⇝NETWORK	906	947	4.53%
LOCATION⇝NETWORK	552	845	53.08%
UNIQUE_IDENTIFIER⇝NETWORK	531	718	35.22%
NETWORK⇝IO	572	659	15.21%
content.res⇝IO	563	563	0%
FILE_INFORMATION⇝NETWORK	495	555	12.12%
IO⇝IPC	516	521	0.97%

5.4 Differences Between μDep and JN-SAF When Applied to Real-World Apps

In our experiment, we observed that μDep detected a lot more sensitive flows than JN-SAF did on datasets S2 and S3. As shown in the last column of Table 6, JN-SAF detected 497 sensitive flows on S2 and 635 sensitive flows on S3, while μDep detected 6,427 and 16,920 sensitive flows on S2 and S3, respectively. Thus, in this section, our goal is to interpret such a vast discrepancy. However, due to the lack of ground truths of sensitive information flow in real-world apps, we have to resort to a less decisive approach to performing the comparison. We first attempt to compare false positives; however, we are not aware of a good way to investigate. Instead, we first enumerate the differences we observed as an objective comparison. Then we estimate the potential true positives of μDep.

There are only 21 apps (4 real-world apps in S2 and 17 malware in S3) detected to be vulnerable by both JN-SAF and μDep. In these apps, μDep detected 26 out of 50 sensitive flows that JN-SAF detected, and μDep also detected many more sensitive flows undetectable by JN-SAF. Thus, albeit μDep and JN-SAF give similar results over the NativeFlowBench suite, for real-world apps, we conclude that the two tools demonstrate significant divergences. More details are shown in Table 8. In terms of notation, $flows_{\mathrm{JS}}$ and $flows_{\mu\mathrm{D}}$ respectively represent the type of sensitive flows detected by JN-SAF and μDep, in the form of *source_category⇝sink_category*:#*flows*. For example, in the category *DroidKungFu* of dataset Drebin [26] (i.e., the 5th row in Table 8), JN-SAF can detect one sensitive flow in each of the seven apps, whose type is IPC⇝IPC. This sensitive flow is also detected by μDep (i.e., $flows_{\text{JN-SAF}} - flows_{\mu\mathrm{Dep}} = \emptyset$). Meanwhile, μDep can detect other 379 sensitive flows distributed in different flow categories in each of these apps. There are also test cases that most of the sensitive flows detected by JN-SAF are missed by μDep, e.g., the 18 flows in Mobinauten of Drebin.

To investigate the true positives of μDep in more detail, based on our evaluation on dataset S1, for apps without native code (e.g., apps in DroidBench 2.0), μDep built upon DroidSafe is prone to achieve a more complete detection than JN-SAF. It shows that compared to JN-SAF, μDep is more capable of discovering sensitive information flows where native code is not involved. For the real-world apps, we argue that there must exist a large portion of such sensitive information flows. So, for this portion of sensitive information flows, we believe μDep should detect more true positives. Moreover, due to the lack of ground truths, we use another independent taint analyzer, FlowDroid [11] (v2.8), to estimate the magnitude of true positives detected by μDep. In Table 8, $\#flows_{\mathrm{FD}}$ is the average number of flows detected in each app by FlowDroid. $\#flows_{\mu\mathrm{D}\cap\mathrm{FD}}$ is the average number of flows in each app detected by both FlowDroid and μDep. Because of the difference in the approaches and configurations, we cannot infer the number of true positives outside $flows_{\mu\mathrm{D}\cap\mathrm{FD}}$. However, we can deduce the flows in $flows_{\mu\mathrm{D}\cap\mathrm{FD}}$ tend to be true positives, which are in general more than the flows detected by JN-SAF. Besides, FlowDroid only uses mock stubs for system-defined native code and should miss sensitive flows related

TABLE 8
Detailed Results of the Sensitive Flows Detected by Both JN-SAF and μDep (Notations: JS=JN-SAF, μD = μDep, FD=FlowDroid)

Category (Dataset)	#apps (μD & JS)	$\#flows_{\mathrm{JS}}$	$flows_{\mathrm{JS}} \cap flows_{\mu\mathrm{D}}$	$flows_{\mathrm{JS}} - flows_{\mu\mathrm{D}}$	$\#flows_{\mu\mathrm{D}}$	$\#flows_{\mathrm{FD}}$	$\#flows_{\mu\mathrm{D}\cap\mathrm{FD}}$
N/A (S2)	2	2	USER_INPUT⇝IPC:1	USER_INPUT⇝SHARED_PREFERENCES:1	43	44	12.5
N/A (S2)	1	1	∅	IPC⇝LOG:1	108	37	16
N/A (S2)	1	1	∅	USER_INPUT⇝SHARED_PREFERENCES:1	23	46	8
DroidKungFu (Drebin)	7	1	IPC⇝IPC:1	∅	380	105.6	13.3
Xsider (Drebin)	2	4	UNIQUE_IDENTIFIER⇝LOG:2, IPC⇝LOG:2	∅	46	89	26.5
	1	2	UNIQUE_IDENTIFIER⇝LOG:2	∅	41	105	31
	1	2	UNIQUE_IDENTIFIER⇝LOG:2	∅	35	99	32
Mobinauten (Drebin)	1	20	UNIQUE_IDENTIFIER⇝LOG:2	IPC⇝IPC:15, UNIQUE_IDENTIFIER⇝SHARED_PREFERENCES:3	47	105	18
Adrd (Drebin)	1	1	∅	os⇝IPC:1	64	24	2
KungFu (DroidAnalytics)	3	1	IPC⇝IPC:1	∅	380	97	14.3
faketaoBao (CICInvesAndMal2019)	1	1	∅	os⇝IPC:1	36	11	1

TABLE 9
Efficiency Comparison of JN-SAF and μDep (Average on Each App)

		Time(s)			Mem(GB)
JN-SAF		SBDA		Native	
		564.50		16.67	8.44
μDep	DepGen	StubGen	DroidSafe	CFBA	
	23.20	0.00	678.96	0.89	21.56

to the native side, which justifies the true positives should be more than $\#flows_{\mu D \cap FD}$.

5.5 Analysis Efficiency of μDep

We evaluate the efficiency of our approach on the apps analyzable by both JN-SAF and μDep. The average analysis runtime and the peak memory consumption of both approaches are presented in Table 9. For JN-SAF, the analysis is divided into two phases, summary-based bottom-up data-flow analysis (SBDA) and native code analysis (Native). For μDep, our analysis consists of the control-flow based static binary analysis (CFBA), the mutation-based dependency generation (DepGen), the stub generation (StubGen), and the information flow analysis using DroidSafe (DroidSafe). For both approaches, the data-flow analysis is in the lead of the computational cost in general. Because of the difference in the points-to analysis, the SBDA of JN-SAF is generally more efficient than the data-flow analysis of μDep (based on DroidSafe) on both time and memory cost. For the native-side static analysis, μDep resorts to control-flow analysis, while JN-SAF implements an annotation-based data-flow analysis. Besides, the automated stub generation is efficient, and the computational cost can be ignored.

The dynamic dependency generation of our approach relies on several configurations. To collect data for the efficiency comparison, we used the default configuration, i.e., *BOUND*=15, and the depth of field applied on the atomic predicates, e.g., cmp_T and $mutate_T$, is 5. The effect of different *BOUND* values and depths of field on the time costs of dependency generation is illustrated in Fig. 4. The time cost increases linearly with the increment of *BOUND*, for each choice on the depth of field. On the other hand, there is no apparent differentiation in the time costs given different depths of field, indicating that, in our implementation, the argument preparations and comparisons using the atomic predicates are efficient.

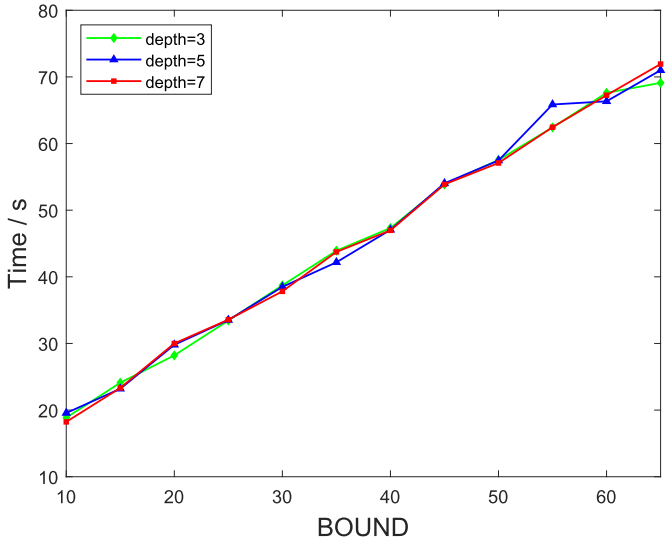


Fig. 4. Effect of configurations of dependency generation on the time cost.

6 Discussion

In this section, we discuss the sources of false positives and false negatives introduced by the dynamic analysis performed on the native code (*.so* files) and the threats to the validity of our approach.

Sources of False Negatives. First, μDep cannot deal with native Activity components, thus will not report the sensitive flows inside these components, which lead to false negatives. Second, the randomly constructed values, objects, and object fields that we feed to the pairwise execution of the native method in Algorithm 2 may be insufficient to discover all data dependencies between arguments and return values of native methods, especially dependencies relying on specific system states, e.g., data from other threads. μDep relies on increasing the mutation iterations, i.e., the value of `BOUND` in Algorithm 2, to improve the coverage of dynamic analysis and to reduce the chance of such false negatives.

Sources of False Positives. Randomness at the native side can be a factor in causing false positives. For example, if a random event can bypass the input cloning and trigger some differentiation of the output of the native function, the dependency generation algorithm will mistakenly build a dependency between the current mutated input and such output. Another source of false positives is the binary-level static control-flow analysis that correlates existing sinks with newly added sinks. Such an analysis introduces over-approximations, which may bring spurious correlations of native sinks to the taint analysis, as shown in `native_complexdata` in Table 2. Moreover, if the native code accesses a stateful function, such a function might be another source of randomness to the local state of the native code, resulting in false positives.

Other Threats. μDep uses dynamic analysis to generate data dependencies for the native methods and then builds the code stubs based on these data dependencies for the static taint analysis. Such *dynamic-over-static* strategy confronts the difficulty of analyzing obfuscated or hardened apps. Such limitations are common in most of the static taint analyses of Android apps, e.g., [2], [11], [13]. We expect unpackers, e.g., PackerGrind [31], [32], and deobfuscators, such as [33], may help mitigate such limitations. In contrast, the dynamic taint analyses, e.g., [7], [9], suffer less from such limitations. On the other hand, our static taint analysis framework inherited from DroidSafe, whose ADI model modified by μDep, mainly supports Android 4.4 APIs, even though our dynamic analysis can support Android 8.0 runtime. Another issue is the difference between the patterns of randomly generated inputs and the specific patterns of the inputs used by app developers. Our random inputs and

mutations may discover a considerable number of valid but rarely used data dependencies. An expected improvement is to profile the context of each native method call to capture the commonly used patterns of arguments.

7 Related Work

Security on Native Code of Android. Native code dramatically benefits the performance-critical applications, e.g., games and graphical acceleration, and for the purpose of anti-reverse engineering [34][35]. Using native code and libraries is very popular in mobile scenarios [36][1]. However, a large portion of native code used in Android apps is migrated from open source projects [35]. Meanwhile, the lack of control mechanisms for the execution of native code, and the misuse of domain-specific native functions, pose a significant threat to the security of the Android platform and apps.

The native code and libraries have been treated as an important source of root exploits and root privilege escalation. Fedler *et al*.[37] propose to control the execution of native binaries and libraries at the system level by modifying `chmod` or customizing specific Java library APIs. RiskRanker [38] compares the native code with the signatures of known root exploits. It also detects whether the encrypted native exploit code is stored in an irregular directory and decrypted for execution at runtime. DroidRanger [39] uses native code as features to identify the family of zero-day malicious apps. It relies on a dynamic execution monitor to inspect the runtime behaviors of untrusted code, especially to collect the system calls made by native code. The PREC framework [40] bridges offline behavior learning and runtime anomaly detection to mitigate root exploits. It uses thread-based dynamic analysis to identify system calls originating from risky third-party native code. By matching the client-side runtime system call sequences to the normal behavior model of the app, PREC identifies malicious system call sequences and suppresses the malicious activity in the native thread.

Moreover, native code is usually exploited to dynamically load external malicious code [41] or violate privacy and safety at the application level [36]. User-level sandboxing is a promising approach to compartmentalize the actions of native code and restrict the communication between native code and Java code. NativeGuard [36] confines the potential malicious behaviors of third-party native libraries by separating the native libraries from Android application to another stand-alone application, where native code resides in a different address space and is deprived of unnecessary privileges, to improve the overall security of the application. AppCage [42] proposes a lightweight inner-process native sandbox to prevent the native libraries of an app from modifying data and code outside the sandbox. The native sandbox relies on SFI and is implemented through binary rewriting and instrumentation. NaClDroid [43] also takes a thread-level SFI to confine the untrusted native code. It uses Google's Native Client program in a separated thread to redirect specific calls for loading the modules which host the native code. NativeProtector [44] follows the process-based isolations of NativeGuard and intercepts sensitive native calls to perform fine-grained access control.

Several system-level approaches treat the Java and native code analyses indiscriminately by capturing the critical runtime features, e.g., system calls, of malicious behaviors. Emulator-level debug tools, e.g., *ltrace*/*strace* [1], [45] and the interception on specific instruction [46], are useful to capture the sensitive system calls, which are drastically relied on by the dynamic analysis for identifying the behaviors of the native part of apps. DroidScope [10] is a general-purpose emulation-based framework using simultaneous two-level VM introspection to rebuild the semantics of Java and native components. CrowDroid [47] uses a client-side app to monitor system calls reflecting the behavior of other apps, collects such system calls on the server side, and builds normality model to detect anomalies of malicious apps. Mobile-Sandbox [45] traces the system calls made into shared objects and logs such events as features for the learning-based malicious behavior detection. CopperDroid [46] also employs VM introspection to record the system calls regardless of whether they are from Dalvik or native code, and then rebuilds the high-level semantics of objects and behaviors. Afonso *et al*.[1] implement a dynamic analysis by instrumenting the core libraries of the emulator to monitor the native-side events. The dynamic analysis automatically generates security policies, i.e., white-list of normal system calls and Java methods invoked by native code, for the existing native sandboxing approach, e.g., NativeGuard [36]. The objective of [1] is to derive proper sandboxing policies for native code that can avoid malicious behaviors but facilitate the normal execution of the native code, while the dynamic analysis of μDep is to assist the static taint analysis. Harvester [3] combines backward slicing with dynamic code execution to resolve reflective method calls and extract malicious features hidden by reflections. The derived reflection information has been used to improve both static and dynamic taint analysis [9], [11]. The reflections derived by Harvest are different from the data dependencies derived by our dynamic dependency generation, which makes Harvester and μDep support diverse aspects of static taint analysis.

Impreciseness of Information Flow Analysis for Android. The permission system of Android has been proved in practice insufficient to detect inconspicuous misbehavior, which violates information flow security policy. For example, suppose the data of one component are permitted to be accessed by another component. In that case, this component will have full authority to dispose of the data, including misoperating on or leaking the data without control. More sophisticated taint tracking approaches or information flow analysis are obligatory to avoid this kind of misbehavior. Taint tracking usually focuses on the explicit flows of sensitive data, while the information flow analysis may also take into account the implicit flows thus is generally more fine-grained. The frequent use of native code makes the taint tracking and information flow analysis more complicated in both static and dynamic approaches. Dynamic taint tracking [7], [9], [10], [48], [49], [50] and static information flow analysis [2], [11], [12], [13], [14], [15], [16] have been widely studied to detect privacy leakages of Android apps. These approaches have built different models for the effects of native code, leading to different accuracy.

In the dynamic approaches, TaintDroid [9] tracks the propagation of labeled sensitive data and reports when sensitive data reach the sinks. Its method-level tracking propagates the taints through the JNI call bridges, conservatively specifies that the tainted primitive-type or string arguments of JNI calls can be delivered to taint the returned value. DroidScope [10] takes a more fine-grained perspective on inspecting the data flow inside the native instructions but ignores implicit information flow. NDroid [7] is built on QEMU as modules that track data flow through JNI. It instruments important JNI functions, e.g., JNI entry/exit, object construction, to track different flows through native contexts, and models the propagation of taint for popular system calls to reduce the performance overhead caused by hooking these frequently invoked functions.

In the static information flow analyses, the state-of-the-art static analyzers [11], [12], [13], [14] do not analyze inside the native component of apps. Instead, they generally resort to some conservative rules to bypass the native calls. These rules can express the taint-propagating relations between used objects, arguments, and return value of native code. Although this kind of abstractions are efficient, they may be neither sound nor precise in modeling taint propagation. Also, the manually crafted models cannot scale up to various third-party native code.

8 Conclusion and Future Work

In this work, we propose a hybrid framework that combines a control-flow based static binary analysis with a dynamic dependency modeling to build the tainting models of native code in Android apps. Based on such tainting models, we derive fine-grained stubs for the native functions and merge them into the ADI model of information flow analysis engine DroidSafe. With such tainting behavior summaries of native code, the DroidSafe engine can detect sensitive data flows triggered by different types of vulnerabilities of native code. The evaluations have demonstrated the applicability of our approach. Without any intention to depreciate the state-of-the-art inter-language approach, our experimental results emphasize that our approach behaves differently on the accuracy and effectiveness compared with the state-of-the-art inter-language analyzer JN-SAF. Especially, our approach can detect more sensitive flows due to its tight integration with the context-, object- and field-sensitive data-flow analysis. Meanwhile, our approach still confronts performance issues, according to the illustration of Table 9.

As future work, we plan to apply our approach to some more efficient analysis framework, e.g., FlowDroid [11]. Lightweight symbolic execution and binary-level points-to analysis [51] are also expected to improve the efficiency of mutation-based dependency generation or to guide the input choices and reduce false negatives. Moreover, we are endeavoring in parameterizing the disassembly part in our system so that users can substitute IDA with other disassemblers such as [52], [53], [54].

Acknowledgments

Cong Sun and Yuwan Ma have contributed equally to this work.

References

[1] V. M. Afonso *et al.*, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy," in *Proc. 23rd Annu. Netw. Distrib. Syst. Secur. Symp.*, 2016.

[2] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "JN-SAF: precise and efficient NDK/JNI-aware inter-language static analysis framework for security vetting of android applications with native code," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1137–1150.

[3] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques," in *Proc. 23rd Annu. Netw. Distrib. Syst. Secur. Symp.*, 2016.

[4] S. Rasthofer, I. Asrar, S. Huber, and E. Bodden, "How current android malware seeks to evade automated code analysis," in *Proc. 9th IFIP WG 11.2 Int. Conf. Informat. Secur. Theory Pract.*, 2015, pp. 187–202.

[5] M. Y. Wong and D. Lie, "Tackling runtime-based obfuscation in android with TIRO," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 1247–1262.

[6] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim, "FLEXDROID: enforcing in-app privilege separation in android," in *Proc. 23rd Annu. Netw. Distrib. Syst. Secur. Symp.*, 2016.

[7] C. Qian, X. Luo, Y. Shao, and A. T. S. Chan, "On tracking information flows through JNI in android applications," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2014, pp. 180–191.

[8] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards on-device non-invasive mobile malware analysis for ART," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 289–306.

[9] W. Enck *et al.*, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. 9th USENIX Symp. Operating Syst. Des. Implementation*, 2010, pp. 393–407.

[10] L. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis," in *Proc. 21th USENIX Secur. Symp.*, 2012, pp. 569–584.

[11] S. Arzt *et al.*, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Conf. Prog.. Lang. Des. Implementation*, 2014, pp. 259–269.

[12] 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 *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1329–1341.

[13] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in DroidSafe," in *Proc. 22nd Annu. Netw. Distrib. Syst. Secur. Symp.*, 2015.

[14] S. Calzavara, I. Grishchenko, and M. Maffei, "HornDroid: Practical and sound static analysis of android applications by SMT solving," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2016, pp. 47–62.

[15] S. Arzt and E. Bodden, "StubDroid: Automatic inference of precise data-flow summaries for the android framework," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 725–735.

[16] P. Lantz and B. Johansson, "Towards bridging the gap between dalvik bytecode and native code during static analysis of android applications," in *Proc. Int. Wireless Commun. Mobile Comput. Conf.*, 2015, pp. 587–593.

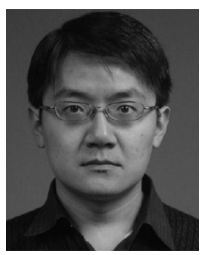
[17] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv, "Precise and compact modular procedure summaries for heap manipulating programs," in *Proc. 32nd ACM SIGPLAN Conf. Prog. Lang. Des. Implementation*, 2011, pp. 567–577.

[18] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "DifFuzz: Differential fuzzing for side-channel analysis," in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 176–187.

[19] Hex-Rays, The IDA Pro disassembler and debugger, 2008. [Online]. Available: https://www.hex-rays.com/products/ida/

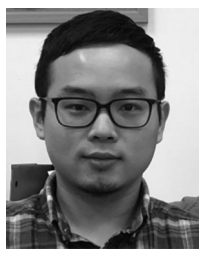
[20] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *Proc. 17th IEEE Comput. Secur. Found. Workshop*, 2004, pp. 100–114.

[21] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proc. Conf. Centre Adv. Stud. Collaborative Res.*, 1999, p. 13.
[22] Deep clone java objects, 2010. [Online]. Available: https://github.com/kostaskougios/cloning
[23] java-util, 2013. [Online]. Available: https://github.com/jdereg/java-util
[24] Droidbench 2.0, 2015. [Online]. Available: https://github.com/secure-software-engineering/DroidBench
[25] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, "AndroZoo: Collecting millions of android apps for the research community," in *Proc. 13th Int. Conf. Mining Softw. Repositories*, 2016, pp. 468–471.
[26] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and explainable detection of android malware in your pocket," in *Proc. 21st Annu. Netw. Distrib. Syst. Secur. Symp.*, 2014.
[27] M. Zheng, M. Sun, and J. C. S. Lui, "Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware," in *Proc. 12th Int. Conf. Trust, Secur. Privacy Comput. Commun.*, 2013, pp. 163–171.
[28] L. Taheri, A. F. A. Kadir, and A. H. Lashkari, "Extensible android malware detection and family classification using network-flows and api-calls," in *Proc. Int. Carnahan Conf. Secur. Technol.*, 2019, pp. 1–8.
[29] Dyninst, 2019. [Online]. Available: https://dyninst.org/
[30] L. Xue *et al.*, "NDroid: Toward tracking information flows across multiple android contexts," *IEEE Trans. Informat. Forensics Secur.*, vol. 14, no. 3, pp. 814–828, Mar. 2019.
[31] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of android apps," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 358–369.
[32] L. Xue *et al.*, "PackerGrind: An adaptive unpacking system for android apps," *IEEE Trans. Softw. Eng.*, vol. 48, no. 2, pp. 551–570, Feb. 2022.
[33] Simple-deobfuscator, 2020. [Online]. Available: https://github.com/SLenik/simple-deobfuscator
[34] M. Protsenko and T. Müller, "Protecting android apps against reverse engineering by the use of the native code," in *Proc. Int. Conf. Trust Privacy Digit. Bus.*, 2015, pp. 99–110.
[35] Q. Wang *et al.*, "NativeSpeaker: Identifying crypto misuses in android native code libraries," in *Proc. 13th Int. Conf. Informat. Secur. Cryptol.*, 2017, pp. 301–320.
[36] M. Sun and G. Tan, "NativeGuard: Protecting android applications from third-party native libraries," in *Proc. ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2014, pp. 165–176.
[37] R. Fedler, M. Kulicke, and J. Schütte, "Native code execution control for attack mitigation on android," in *Proc. 3rd ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2013, pp. 15–20.
[38] M. C. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and accurate zero-day android malware detection," in *Proc. 10th Int. Conf. Mobile Syst., Appl., Serv.*, 2012, pp. 281–294.
[39] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets," in *Proc. 19th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2012.
[40] T. Ho, D. J. Dean, X. Gu, and W. Enck, "PREC: Practical root exploit containment for android devices," in *Proc. 4th ACM Conf. Data Appl. Secur. Privacy*, 2014, pp. 187–198.
[41] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications," in *Proc. 21st Annu. Netw. Distrib. Syst. Secur. Symp.*, 2014.
[42] Y. Zhou, K. Patel, L. Wu, Z. Wang, and X. Jiang, "Hybrid user-level sandboxing of third-party android apps," in *Proc. 10th ACM Symp. Informat., Comput. Commun. Secur.*, 2015, pp. 19–30.
[43] E. Athanasopoulos, V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "NaClDroid: Native code isolation for android applications," in *Proc. 21st Eur. Symp. Res. Comput. Secur.*, 2016, pp. 422–439.
[44] Y. Hong, Y. Wang, and J. Yin, "NativeProtector: Protecting android applications by isolating and intercepting third-party native libraries," in *Proc. 31st IFIP TC 11 Int. Conf. Syst. Secur. Privacy Protection*, 2016, pp. 337–351.
[45] M. Spreitzenbarth, T. Schreck, F. Echtler, D. Arp, and J. Hoffmann, "Mobile-sandbox: Combining static and dynamic analysis with machine-learning techniques," *Int. J. Informat. Secur.*, vol. 14, no. 2, pp. 141–153, 2015.
[46] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of android malware behaviors," in *Proc. 22st Annu. Netw. Distrib. Syst. Secur. Symp.*, 2015.
[47] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for android," in *Proc. 1st ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2011, pp. 15–26.
[48] M. Sun, T. Wei, and J. C. S. Lui, "TaintART: A practical multi-level information-flow tracking system for android runtime," in *Proc. SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 331–342.
[49] D. Schoepe, M. Balliu, F. Piessens, and A. Sabelfeld, "Let's face it: Faceted values for taint tracking," in *Proc. 21st Eur. Symp. Res. Comput. Secur.*, 2016, pp. 561–580.
[50] L. Xue, C. Qian, and X. Luo, "AndroidPerf: A cross-layer profiling system for android applications," in *Proc. 23rd IEEE Int. Symp. Qual. Serv.*, 2015, pp. 115–124.
[51] S. H. Kim, C. Sun, D. Zeng, and G. Tan, "Refining indirect call targets at the binary level," in *Proc. 28th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2021.
[52] D. Zeng and G. Tan, "From debugging-information based binary-level type inference to CFG generation," in *Proc. 8th ACM Conf. Data Appl. Secur. Privacy*, 2018, pp. 366–376.
[53] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_05A-4_Bauman_paper.pdf
[54] A. Flores-Montoya and E. M. Schulte, "Datalog disassembly," in *Proc. 29th USENIX Secur. Symp., USENIX Secur.*, 2020, pp. 1075–1092. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya



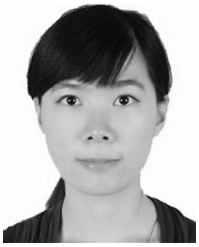
Cong Sun (Member, IEEE) received the BS degree in computer science from Zhejiang University, Zhejiang, China, in 2005 and the PhD degree in computer science from Peking University, China, in 2011. He is currently a full professor with the School of Cyber Engineering, Xidian University, China. His research interests include information flow security, software security, and program analysis.

Yuwan Ma received the MS degree majoring in computer science from Xidian University, Xi'an, China, in 2020. Her research interests include Android security and binary analysis.

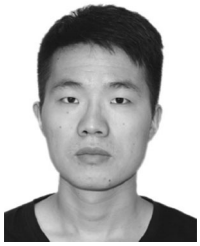


Dongrui Zeng received the BS degree in computational mathematics from Nanjing University, Nanjing, China, in 2014, and the PhD degree in computer science and engineering from Pennsylvania State University, University Park, Pennsylvania, in 2021. Currently, he is working with Palo Alto Networks as a security research engineer. His major research interests include software security and program analysis.

Gang Tan (Member, IEEE) received the bachelor's (with Hons.) degree in computer science from Tsinghua University, China, in 1999, and the PhD degree from Princeton University, Princeton, New Jersey, in 2005. He is currently a full professor with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, Pennsylvania. He leads the Security of Software (SOS) Lab. His research interests include the interface between computer security, programming languages, and formal methods. He has received an NSF CAREER award, two Google Research Awards, and a Francis Upton Graduate Fellowship. He is a member of the ACM.



Siqi Ma received the BS degree in computer science and technology from Xidian University, China, and the PhD degree in information system from Singapore Management University, Singapore. She is currently a senior lecturer with the University of New South Wales, Canberra Campus. Her research interests include software security, mobile apps and IoT firmware.



Yafei Wu is currently working toward the MS degree in the School of Cyber Engineering, Xidian University, Xi'an, China, His research interests include Android security and program analysis.