Detecting Vulnerable Android Inter-App Communication in Dynamically Loaded Code

Mohannad Alhanahnah*, Qiben Yan*, Hamid Bagheri*, Hao Zhou†, Yutaka Tsutano*, Witawas Srisa-an*, Xiapu Luo†
*The Department of Computer Science and Engineering, University of Nebraska-Lincoln, USA
†The Department of Computing, The Hong Kong Polytechnic University, China

***Abstract*—Java reflection and dynamic class loading (DCL) are effective features for enhancing the functionalities of Android apps. However, these features can be abused by sophisticated malware to bypass detection schemes. Advanced malware can utilize reflection and DCL in conjunction with Android Inter-App Communication (IAC) to launch collusion attacks using two or more apps. Such dynamically revealed malicious behaviors enable a new type of stealthy, collusive attacks, bypassing all existing detection mechanisms. In this paper, we present DINA, a novel hybrid analysis approach for identifying malicious IAC behaviors concealed within dynamically loaded code through reflective/DCL calls. DINA continuously appends reflection and DCL invocations to control-flow graphs; it then performs incremental dynamic analysis on such augmented graphs to detect the misuse of reflection and DCL that may lead to malicious, yet concealed, IAC activities. Our extensive evaluation on 3,000 real-world Android apps and 14,000 malicious apps corroborates the prevalent usage of reflection and DCL, and reveals previously unknown and potentially harmful, hidden IAC behaviors in real-world apps.**

***Index Terms*—Mobile security; inter-app communication; reflection; dynamically loaded code**

I. Introduction

Despite recent significant advances in malware detection, trojanized apps can still circumvent the defense and enter the official Google Play Store. Intricate Android malware, such as Obad [1], has been creating a more stealthy threat by *employing Java reflection to commit malicious acts at runtime.* Java reflection mechanism is extensively used in Android apps for maintaining backward compatibility, accessing hidden/internal application program interface (API), providing external library support, and reinforcing app security [2], [3]. But the use of the reflection mechanism renders the security analysis approaches designed to analyze and detect malicious apps ineffective [4]. Fig. 1 illustrates a reflective call where the actual reflection targets (i.e., *Classes B, C* and *D*) cannot be resolved by static analysis tools as the malicious code is not part of the apps' bytescode, rather is loaded at runtime using the dynamic class loading (DCL). As a concrete example, MYSTIQUE-S [5] is recently developed to produce new malware that incorporates reflection and DCL to deliver malicious payloads. Moreover, via *inter-app communications (IAC)*, sophisticated collusive security threats exploit multiple apps to create longer calling paths to launch malicious activities [6]. A recent example is cross-app remote infection [7], in which a remote adversary can circulate malicious contents across apps' WebView instances to obtain surreptitious, yet persistent, control of the apps.

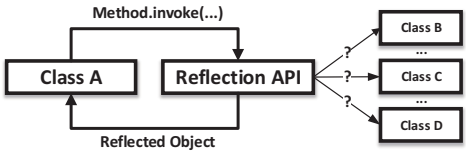


Fig. 1: A typical reflective call used to defeat static analyzers.

The current state-of-the-art security mechanisms, both static and dynamic analysis approaches, are insufficient for detecting the increasingly sophisticated security attacks. Static analysis approaches [8]–[11] can be easily bypassed by apps that covertly invoke malicious IAC using reflection or DCL. On the other hand, dynamic analysis approaches, such as TAMIFLEX [12], STADYNA [3], and DyDroid [13], suffer from false negatives largely due to the reachability challenges, where vulnerabilities are missed because of inputs that fail to reach the vulnerable code; they thus do not detect malicious IACs concealed behind reflective and DCL calls.

In this paper, we present DINA, a novel hybrid inter-app analysis technique that can detect IAC behaviors concealed by reflection and DCL. Our goal is to develop a fully automated, lightweight tool that operates in real-time to effectively identify apps that deliver vulnerable IAC components at runtime. DINA combines scalable static and dynamic analyses to incrementally augment the control-flow and data-flow graphs using dynamically loaded classes. DINA ensures every method in every class of the dynamically loaded code is analyzed. Vulnerable IAC detection is conducted continuously in real-time over the dynamically updated graphs to precisely pinpoint collusive data leaks and data manipulations. The **continuous and real-time analysis** allows DINA to be highly efficient in both identifying dynamically loaded components and detecting potential malicious behaviors thereof, making DINA ideally suited for large-scale security vetting. In addition, DINA enables concurrent multi-app analysis using a thorough graph generation and analysis approach, which has the potential to be deployed locally for real-time app auditing.

In summary, this paper presents the following contributions:

- We develop DINA, the first inter-app vulnerability detection tool with the capability of analyzing dynamically loaded code, to pinpoint the stealthy inter-app communications that are concealed using reflection and DCL. DINA combines static IAC analysis with incremental dynamic analysis to identify potential IAC vulnerabilities within dynamically loaded code at runtime.

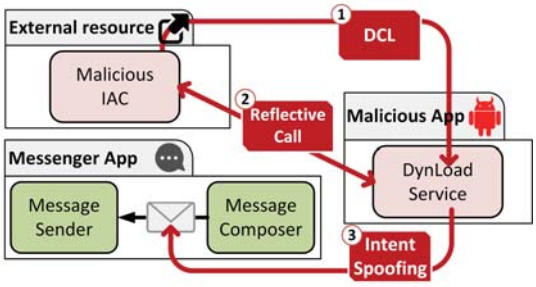


Fig. 2: Malicious app downloads code at runtime, and then uses it for leaking sensitive information.

- We analyze 3,000 popular benign apps and 14,000 malicious apps to identify their reflection usage and IAC communications via reflection/DCL. The analysis results confirm the prevalent usage of reflection and DCL in popular real-world apps, wherein surreptitious IAC behaviors concealed by reflective calls have been observed. We provide detailed case studies to assess how the vulnerable apps can be exploited to launch stealthy attacks through reflection and DCL.
- We evaluate DINA in the context of both real-world and benchmark apps, corroborating DINA's effectiveness in detecting sophisticated IAC threats that can evade the state-of-the-art security analysis tools. We perform a comprehensive comparison with the existing analysis tools. The experimental results confirm the superior performance of DINA in detecting malicious IACs in dynamically loaded code.

II. Background and Motivation

Components including *activities*, *services*, *broadcast receivers*, and *content providers*[1] are the basic building blocks of Android apps. These components communicate through a specific type of event messages called *Intent*, which can be either explicit, when its recipient component is specified, or implicit, when no specific recipient component is declared. In this section, we present motivating examples showing how Intent can be used as an attack vector to launch information leakage through hidden (dynamically loaded) code, and to conceal method invocations through reflection.

Fig. 2 presents a bundle of two apps, where a malicious IAC is initiated within a dynamically loaded component from an external source to leak sensitive information through the *Messenger* app. The *DynLoadService* component dynamically loads a malicious class from an external JAR file placed at the location specified on line 7 of Listing 1. It then instantiates a *DexClassLoader* object, and uses it to load the DEX (Dalvik Executable) file contained in the JAR file. Using Java reflection at line 12, the *mDexClassLoader* object loads a class called *MalIAC* and invokes its *getIntent* method at line 14. This method returns an implicit Intent, which *DynLoadService* uses to communicate with the *Messenger Sender* (line 15).

[1]https://developer.android.com/guide/components/fundamentals.html

```
1  public class DynLoadService extends Service {
2   public int onStartCommand(Intent intent){  [...]
3    loadCode();
4   }
5   public void loadCode(){
6    // Read a jar file that contains classes.dex file
7    String jarPath=Environment.getExternalStorageDirectory().getAbsolutePath
         ()+"/dynamicCode.jar";
8    // Load the code
9    DexClassLoader mDexClassLoader = new DexClassLoader(jarPath, getDir("
         dex", MODE_PRIVATE).getAbsolutePath());
10   // Use reflection to load a class and call its method
11   Class<?> loadedClass = mDexClassLoader.loadClass("MalIAC");
12   Method methodGetIntent = loadedClass.getMethod("getIntent", android.
         content.Context.class);
13   Object object = loadedClass.newInstance();
14   Intent intent = (Intent) methodGetIntent.invoke(object, DynamicService.
         this);
15   startService(intent); } }
```

Listing 1: *DynLoadService* component resides in the malicious app and performs DCL and reflection to hide its malicious behavior.

```
1  public class MalIAC {
2   public Intent getIntent(Context context){
3    String account = getBankAccount("Bank_Account");
4    String balance = getBankBalance("Balance_USD");
5    Intent i = new Intent("SEND_SMS");
6    i.putExtra("PHONE_NUM", phoneNumber);
7    i.putExtra("Bank_Account", account);
8    i.putExtra("Balance_USD", balance);
9   return i; } }
```

Listing 2: Malicious IAC component is concealed as external code and loaded at runtime after app installation.

Listing 2 depicts the hidden malicious class aiming at stealing users' sensitive information. On lines 3-4, *getIntent* obtains the sensitive banking information, and then creates an *implicit Intent* with a phone number and the banking information as the extra payload of the Intent (lines 5-8). This code is pre-compiled into DEX format and archived to a JAR file. The JAR file could be downloaded by the malicious app after installation. The *Messenger* app, as shown in Listing 3, receives the Intent and sends a text message using the Intent payload, effectively leaking sensitive data.

```
1  public class MessageSender extends Service {
2   public void onStartCommand(Intent intent) {
3    String number=intent.getStringExtra("PHONE_NUM");
4    String message=intent.getStringExtra("TEXT_MSG");
5    sendTextMessage(number, message);
6   }
7   void sendTextMessage(String num, String msg) {
8    SmsManager mngr = SmsManager.getDefault();
9    mngr.sendTextMessage(num,null,msg,null,null); } }
```

Listing 3: MessageSender resides in a benign app to receive Intents and send text messages.

Listing 4 presents an abbreviated code snippet from a real-world app (i.e., *com.example.qianbitou*) that uses reflection to conceal IAC behavior. The method *instantiate* in the class *Fragment* (line 2) calls the reflection method *newInstance()* (line 4). This reflective call will initialize the constructor of the class *_03_UserFragment* (line 6), and execute the method *onClick()* that invokes *toCall()*, which defines an implicit Intent for making a phone call to a hard-coded number between 8am and 10pm. The suspicious method *toCall()* is a private method concealed behind reflective calls, which is difficult to capture in the analysis.

```
1  public class Fragment {
2   public static Fragment instantiate (){
3    // Reflection call site
4    paramContext = (Fragment)localClass1 .newInstance();}
5  }
6  public class _03_UserFragment extends Fragment {
7   public onClick(){
8    toCall();
9   }
10  // The method invoked through the reflective call at line 4
11  private void toCall(){
12      int i = Calendar. getInstance() .get();
13      if ((i <= 22) || (i >= 8)) {
14          startActivity (new Intent("android . intent . action .DIAL", Uri.parse(" tel
                :4000−888−620"))); } } }
```

Listing 4: Reflection is used to conceal IAC behavior in a real world app

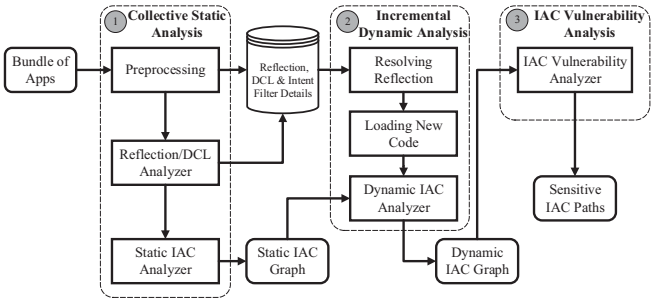


Fig. 3: Architecture of DINA.

DINA is designed to load and resolve the reflective calls in Listings 1 and 4 at runtime. DINA's dynamic analyzer automatically and incrementally augments both the control-flow and data-flow graphs with the newly loaded code and resolved reflective calls. In tandem with the graph augmentation, DINA's vulnerability analyzer identifies potential malicious IAC activities on the fly. As a result, DINA has the capability to precisely and efficiently detect the malicious IAC behavior in the motivating examples although it is concealed by reflection.

III. Detecting Hidden IAC with DINA

The goal of the attacker considered in this work is to launch stealthy inter-app attacks without being detected. Such stealthy behavior can be manifested by different types of *collusive attacks* [6], where an attacker uses the DCL and reflection mechanisms to obfuscate IAC behaviors of the sender app and launch malicious behaviors, e.g., leaking sensitive information, via another receiver app. Fig. 3 illustrates DINA's architecture. DINA is a *graph-centered hybrid analysis* system that consists of three main modules: 1) the *collective static analysis* module that simultaneously analyzes multiple apps to automatically generate a static IAC graph and elicit DCL and reflection call sites within the apps—the identified DCL and reflection call sites become the execution targets for dynamic analysis; 2) the *incremental dynamic analysis* module that systematically augments the static IAC graphs by attaching new nodes and edges that are loaded at runtime by DCL and reflection; 3) the *IAC vulnerability analysis* module utilizes real-time IAC graphs to identify potentially vulnerable paths.

A. Collective Static Analysis

The collective static analysis of DINA aims to statically identify the reflection, DCL and IAC capabilities of each app in the app bundle, by analyzing multiple apps at the same time. DINA is a classloader-based analysis system written in C++ that builds upon JITANA [8] to statically and dynamically analyze multiple apps simultaneously. This approach is more scalable than compiler-based approaches, such as the popular SOOT, which analyzes the entire code of one app at a time.

We generate two different types of graphs for each app, the *method call graph* (MCG) and *instruction graph* (IG). The method call graph maintains the call relationships among the methods defined within the analyzed apps in the bundle, while the instruction graph includes detailed control-flow and data-flow information for a certain method. DINA works on the bytecode level of the target application, and the analysis focuses on the app's Dalvik bytecode. Algorithm 1 outlines the collective static analysis process, which consists of three major steps:

Preprocessing. We first decompile APKs in the collective app bundle to generate the bytecode of each app and extract its manifest file. Intent filter information for each app is then extracted from the manifest file. This step also involves the generation of MCG for each app and the IG for each method in the method call graph. All extracted information and the generated graphs are stored in a database for fast access.

Reflection/DCL analyzer. We then identify DCL and reflective calls using the MCG of each app by detecting DCL and reflection APIs, such as `invoke()`, `newInstance()`, and `getMethod()`. The list of reflection and DCL APIs (i.e., $Ref\_DCL\_API\_List$ in Algorithm 1) is similar to the API list in STADYNA [3], which mainly includes APIs of dynamic class loading. We extend that list to include additional reflection APIs involving method invocations [2]. As a result, this step identifies the apps that need to be executed in the incremental dynamic analysis module. We further extract the class and method names (call sites) implementing these APIs.

Static IAC analyzer. We identify the IAC paths by performing string matching over the IGs of each method to match the Intent action strings that are extracted from the manifest files of all apps. If a matching is found, a new edge will be added between the components of two apps related to the matched Intent action, and the static IAC graph will be augmented via the `addEdge()` method (Algorithm 1, lines 17–19). Collectively, all the component-level IAC paths will be aggregated to construct a complete static IAC graph (i.e., *static_IAC* in Algorithm 1) for all the analyzed apps in the app bundle. The generated static IAC graph depicts the communication activities of the apps at the component-level. DINA also identifies the methods responsible for Intent-sending activities within each component of the communicating apps in the static IAC graph.

B. Incremental Dynamic Analysis

DINA performs incremental dynamic analysis for each app that contains reflective or DCL calls. The dynamic analysis is

Algorithm 1 Collective Static Analysis

INPUT: Bundle of Apps: B, $Ref\_DCL\_API\_List$
OUTPUT: $static\_IAC$, $Intent\_Filter\_App_i$, $Ref\_Details$

```
// Preprocessing
static_IAC ← CreateNodes(|B|)
Intent_Filter_App_i ← {}                          // initialize Intent filter list
for each App_i ∈ B do
    Decompile(App_i)
    parse_manifest(App_i)
    update(Intent_Filter_App_i) ← {(App_i, class-name, intent-action-string)}
end for
for each App_i ∈ B do
    Generate MCG(App_i)
  // Reflection analyzer
    for each method ∈ MCG(App_i) do
        if method_j ∈ Ref_DCL_API_List then
            update(Ref_Details) ← {(App_i, class-name, method-name)}
        end if
        Generate IG(method_j)
  // Static IAC Analyzer
        for each instruction ∈ IG(method_j) do
            for each String str in instruction_k do
                if str ∈ Intent_Filter_App_r.intent-action-string then
                    Update static_IAC ← addEdge(App_i, App_r)
                end if
            end for
        end for
    end for
end for
```

capable of capturing and loading code in various formats (i.e. APK, ZIP, JAR, DEX), resolving reflection, and performing IAC analysis incrementally with progressive augmentation of graphs. We modified Android framework for resolving reflective calls and capturing newly loaded codes at runtime. The incremental dynamic analysis consists of two major steps as described below (see Algorithm 2).

Resolving reflection and loading new codes. Every app implementing reflection and DCL will be executed on a real Android device or an emulator. This step aims to capture the dynamic behaviors of the app. To reach the components that implement reflection, we use the reflection details extracted and stored in the database, which includes the component name and the corresponding method name that implement reflection and DCL in each app. These methods/components of an app, regarded as *method of interest (MoI)*, will be exercised in the dynamic analysis for resolving reflection and DCL call sites, which will augment the control-flow and data-flow graphs dynamically. We utilize a fuzzing approach to trigger the components that contain reflection and DCL call sites.

Dynamic IAC analyzer. DINA's dynamic analyzer will execute components to generate dynamic analysis graphs. DINA immediately starts pulling and analyzing files when new dex files or classes are loaded through the DCL mechanism. It not only analyzes the triggered MoIs, but also loads the entire class that the MoI belongs to. All methods within the class will be subsequently analyzed, while some of which may initiate reflection/DCL. After generating the dynamic graphs that are augmented by resolving reflection in real-time (i.e., *dynamic_IAC* in Algorithm 2), it then performs an incremental analysis approach to detect IAC activities continuously during every augmentation process. In the end, the static IAC graph will be refined to include all the IAC detected inside the dynamically loaded codes after resolving reflection. New edges pertaining to the identified IAC are added to the graph at runtime.

To concretize our idea of DINA's dynamic analysis, consider the code snippet from the Echoer app in DroidBench[2], shown in Listing 5. Two different Intent messages will be constructed based on the Intent action that is resolved at runtime (lines 3-8). DINA's incremental analysis approach is performed over the incrementally augmented IAC graph using a *static analyzer* (similar to lines 15-18 in Algorithm 1). As a result, DINA's dynamic analyzer will be able to effectively uncover hidden codes in both cases, even if only one of them is executed at runtime.

```
1  Intent i = getIntent();
2  String action = i.getAction();
3  if (action.equals(Intent.ACTION_SEND)) {
4   Bundle extras = i.getExtras();
5   Log.i("TAG", "Data received in Echoer: " + extras.getString("secret"));
        }
6  else if (action.equals(Intent.ACTION_VIEW)){
7   Uri uri = i.getData();
8   Log.i("TAG", "URI received in Echoer:" + uri.toString()); }
```

Listing 5: Excerpts from DroidBench's Echoer app.

Algorithm 2 Incremental Dynamic Analysis

INPUT: $static\_IAC$, $Ref\_Details$, $Intent\_Filter\_App_i$
OUTPUT: $dynamic\_IAC$

```
dynamic_IAC ← static_IAC
// Resolving Reflection and Loading new code
for each App_i do
    Install(App_i)
    Launch(App_i)
    Pull newly loaded code
    for each Component ∈ Ref_Details(App_i) do
        Find method of interest (MoI)
        for each Method ∈ MoI(App_i) do
            Execute the component using Monkey (if failed, execute the whole app using
    Monkey), and incrementally generate IG(method_j)
            for each instruction ∈ IG(method_j) do
  // Dynamic IAC analyzer
                for each String str in instruction do
                    if str matches Intent_Filter_App_r.action-string then
                        dynamic_IAC ← addEdge(App_i, App_r)
                    end if
                end for
            end for
        end for
    end for
    uninstall(App_i)
end for
```

C. IAC Vulnerability Analysis

Algorithm 3 depicts the process of IAC vulnerability analysis. Essentially, it identifies whether the nodes in the dynamic IAC graph constitute a vulnerable path that reveals sensitive information. IAC vulnerability analyzer performs its analysis over all identified IAC paths in the dynamic IAC graph. Then for each path, every node is analyzed, by identifying whether it is a sender or receiver node, and then *depth-first search (DFS)* is conducted to find if this node can reach a sensitive source method in case of sender node, or can reach a sensitive sink in case the node is receiver. We leverage a sensitive API list that simplifies the widely used SuSi list [14] to identify these sensitive APIs. An inverted DFS searches from the recorded

[2]https://github.com/secure-software-engineering/DroidBench/blob/develop/eclipse-project/InterAppCommunication/Echoer

MoI (identified in Algorithm 2) to seek sensitive sources, while another DFS searches from Intent-receiving method at the receiver (line 12 in Algorithm 3) to look for sensitive sinks. Finally, it marks the complete sensitive paths from the sensitive source to the sensitive sink across multiple apps. Note that these paths are stealthy and difficult to find, as they only appear after loading dynamic codes and resolving reflection calls, but they can be captured by DINA efficiently.

Algorithm 3 IAC Vulnerability Analysis

INPUT: $dynamic\_IAC$, $Sensitive\_API\_List$
OUTPUT: $node_i.sensitive$

```
1: for each node of App_m ∈ dynamic_IAC do
   // Identify sensitive methods in the sender node
2:   if node_i is sender then
3:     for each method ∈ DFS(node_i.method-name) do
4:       if method_j ∈ Sensitive_API_List then
5:         node_i.sensitive = True
6:       else
7:         node_i.sensitive = False
8:       end if
9:     end for
   // Identify sensitive methods in the receiver node
10:  else if node_i is receiver then
11:    for each method ∈ MG(App_m) do
12:      if method_j ∈ {onCreate, onReceive, onStartCommand} &&
   (class-name of method_j == class-name of node_i) then
13:        for each method ∈ DFS(method_j) do
14:          if method_j ∈ Sensitive_API_List then
15:            node_i.sensitive = True
16:          else
17:            node_i.sensitive = False
18:          end if
19:        end for
20:      end if
21:    end for
22:  end if
23: end for
```

IV. Evaluation

This section presents our experimental evaluation of DINA. The current DINA's dynamic analysis prototype is implemented for Android 4.3. We find Android 4.3 is sufficient for our study, since we observe no differences in DCL-related APIs between Android 4.3 and Android 7.1. This observation is also confirmed by Qu et al. [13]. Currently, we have begun porting DINA to support ART, the latest Android runtime system. We then conduct our evaluation to answer the following four research questions:

- **Question 1:** How *accurate* is DINA in identifying vulnerable IAC/ICC activities compared to the state-of-the-art static and dynamic analyses?
- **Question 2:** How *robust* is DINA in analyzing the capabilities/behaviors of reflection and DCL implementations in real-world apps?
- **Question 3:** How *effective* is DINA in detecting vulnerabilities in real-world apps?
- **Question 4:** How *efficient* is DINA in performing hybrid analysis?

A. How accurate is DINA?

Evaluating the accuracy of DINA requires performing the analysis on a ground truth dataset, where the attacks are known in advance. This constitutes a major challenge due to the lack of existing colluding apps [15], specifically benchmark apps that are using reflection and DCL for performing malicious IAC. We found 12 suitable Benchmark apps (listed in Table I) from DroidBench and other resources to validate DINA's detection effectiveness and efficiency, all of which perform ICC or IAC through reflection or DCL.

Comparing with static analysis tools. Next, we consider three state-of-the-art static analysis systems: IccTA [16], SEALANT [11], and DroidRA [2] designed to identify suspicious IAC and reflection activities. DroidRA focuses on detecting reflective calls using composite constant propagation. IccTA is a static analysis tool that can detect vulnerable ICC paths using inter-component taint analysis based on FlowDroid. SEALANT combines data-flow analysis and compositional ICC pattern matching to detect vulnerable ICC paths. To construct a baseline system that shares the same capability as DINA, we attempted to integrate these two types of techniques: DroidRA was used to resolve reflective calls, while IccTA and SEALANT were used to detect vulnerable IACs in targets captured by DroidRA. Here, we compare DINA's reflection resolution and IAC detection performance with other baseline approaches.

Comparing reflection resolution performance: we compare reflection/DCL resolution capabilities of DINA and DroidRA over benchmark and real-world apps. We found that DroidRA was able to resolve reflective calls in 8 out of 12 benchmark apps in Table I. DroidRA did not detect any reflective calls in *OnlyTelephony_Reverse.apk* and *OnlyTelephony_Substring.apk*, and it crashed during the inter-component analysis of *DCL.apk*. The only app that DroidRA can successfully analyze and annotate with reflection targets is *reflection11.apk*. On the other hand, DINA has resolved all reflection and DCL calls in the benchmark apps. For real-world apps, our results show that DINA can detect more reflective calls than DroidRA. For instance, for a malware sample[3] that contains 14 reflective calls and 4 DCL calls. DroidRA detects 11 of them, while DINA captures all reflective/DCL calls. This is because DroidRA fails to detect the reflective calls within the dynamically loaded code.

Comparing IAC detection performance: we perform ICC/IAC analysis using SEALANT and IccTA over the instrumented benchmark apps by DroidRA. Although DroidRA successfully resolved the reflective calls of 8 benchmark apps, it was not able to correctly instrument the apps with those reflection targets required for IAC analysis. Our results indicate that many of these targets reside within the Android framework, and thus are not considered in the analysis conducted by DroidRA. We also found that while the annotated APK is structurally correct, it can no longer be executed. Moreover, we observed that SEALANT yields invalid results after analyzing the instrumented APKs by DroidRA, which may be caused by the incompatibility of the generated APK format with SEALANT's input. Therefore, we did not use the instrumented APKs, instead we used DroidRA's reported

[3]MD5: 00db7fff8dfbd5c7666674f350617827

reflection resolution results, and then use these results in conjunction with SEALENT and IccTA's results to identify vulnerable IAC paths within benchmark apps.

Table I shows IAC detection comparison results in terms of precision, recall and F-measure scores. Note that we did not report the results of IccTA because it can only produce results for 5 out of 12 apps (*ActivityCommunication2, OnlyIntent, OnlySMS, reflection11, and SharedPreferences1*), but fails to detect any vulnerabilities. SEALANT performs better in a number of benchmarks, yet produces several false positives that affects its precision.

TABLE I: IAC detection performance comparison between DroidRA+SEALANT and DINA. True Positive (TP), False Positive (FP), and False Negative (FN) are denoted by symbols ☑, ☒, ☐, respectively. ($X\#$) represents the number # of detected instances for the corresponding symbol X. Also note that IccTA did not detect any vulnerable paths.

Test Cases	DroidRA+SEALANT	DINA
ActivityCommunication2	☑(☒3)	☑
AllReflection	☑(☒3)	☑
OnlyIntent	☐	☑
OnlyIntentReceive	☑(☒2)	☑
OnlySMS	☑(☒3)	☑
OnlyTelephony	☑(☒3)	☑
OnlyTelephony_Dynamic	☑(☒3)	☑
OnlyTelephony_Reverse	☐	☑
OnlyTelephony_Substring	☐	☑
SharedPreferences1	☐	☐
Reflection_Reflection11	☑	☑
Dynamic class loading	☐	☑
Precision	29.2%	**100%**
Recall	58.3%	**91.6%**
F-measure	38.9%	**95.62%**

The experimental results show that DINA can handle reflective and DCL calls to de-obfuscate ICC, and reaches 100% precision and 91.6% recall in detecting vulnerable ICC. As for the app *SharedPreferences1*, DINA detects the reflective calls, but misses its ICC path, because the shared preference mechanism used in the app is not considered by DINA.

Comparing with dynamic analysis tools. As for the dynamic analysis approach, the most closely-related technique is HARVESTER [17], which uses program slicing to deobfuscate reflective calls for dynamic execution, yet we were informed by the authors that neither the source code nor the binary version of HARVESTER is available. Moreover, HARVESTER's precision was not evaluated over benchmark apps, which makes it hard to compare against. IntentDroid [18] is a dynamic analysis tool for detecting vulnerable IAC, but it cannot deal with reflection/DCL.

B. How robust is DINA?

In this section, we evaluate DINA's capabilities to reveal the behavior of reflection/DCL classes in complex, real-world apps. We used three datasets with 49,000 real-world apps, including: 1) 31,894 apps from AndroZoo project[4], 2) 3,000 most popular apps from Google Play store, and 3) 14,294 uncategorized malware samples from VirusShare[5].

TABLE II: Dynamic analysis of real-world apps.

Dataset	# of installed Apps	# of apps contain reflection/DCL	# of activated reflection/DCL	% of 3rd party classes	% of app-owned classes
Benign	1,957	1,271	17,170	85.6%	7.66%
Malicious	2,378	1,033	7,336	54.6%	5.18%

TABLE III: Intent sending and receiving capabilities of activated ref/DCL classes.

Dataset	# of Intent sending APIs (reachable)	# of Intent receiving APIs
Benign	1022 (936)	146
Malicious	600 (390)	79

Reflection/DCL Usage Landscape. We first performed the collective static analysis of DINA using the three previously-mentioned sets of apps to identify reflection and DCL call sites. The experimental results show that 92.0% (i.e., 26,361/31,894) of AndroZoo apps implement reflection calls, and 51.1% (i.e., 16,313/31,894) of them implement DCL calls. This shows the wide adoption of DCL and reflection mechanisms in Android apps. More remarkably, 99.4% of 3,000 popular apps implement reflection calls, and 90.1% of them implement DCL calls. Therefore, reflection and DCL mechanisms are even more widely adopted in popular apps. For the malware apps, 85.0% implement reflection mechanism, while only 24.3% of them adopt DCL mechanism. Solely based on our evaluation, it seems that fewer malware apps use the DCL mechanism. Note that DINA counts the number of APIs by traversing the whole method graph, which produces an accurate representation of the apps under analysis.

We then run the dynamic analysis on the popular apps and randomly-picked malicious apps. Table II presents the results, from which we can see that the number of activated classes in the benign apps significantly exceeds that of the malicious apps. We also perform further analysis to identify the entity, either the app itself or a third-party library, behind the activated reflection and DCL classes. Note that we ignore the Android framework classes. We can see most of the activated reflection/DCL classes are included in third-party APIs in both malicious and benign apps, as has been confirmed by prior research [13].

Intent sending/receiving capabilities of DCL/reflection classes. Next, we evaluate DINA's incremental dynamic analysis to detect Intents in dynamically loaded code. We analyze the activated reflection/DCL classes of popular and malicious apps to identify the Intent sending and receiving APIs presided within the reflection/DCL classes. Table III presents the number of Intent sending APIs and receiving APIs. We use DFS as a reachability test to find whether MoI can reach the Intent sending APIs as shown in Table III.

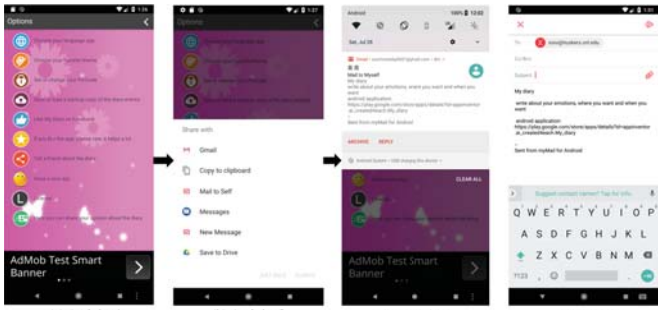
Sensitive sources within DCL/reflection classes. Table IV presents the top 10 sensitive sources in the activated reflection

[4]androzoo.uni.lu

[5]virusshare.com

TABLE IV: Top sensitive sources in the activated reflection and DCL classes.

Benign		Malicious	
Sensitive APIs	Freq.	Sensitive APIs	Freq.
getInstalledApplications()	113	getSubscriberId()	35
getMacAddress()	100	getSSID()	29
getCountry()	42	getMacAddress()	7
getActiveNetworkInfo()	33	getDeviceId()	6
getInstalledPackages()	21	getCountry()	6
openConnection()	20	getInstalledPackages()	6
getDeviceId()	7	openConnection()	4
getSubscriberId()	1	getSimSerialNumber()	2



(a) Activity 1 (b) Activity 2 (c) Activity 3 (d) Exploit

Fig. 4: (a). Activity 1 of sender app; (b). Activity 2 of sender app initiated through reflection; (c). Activity 3 represents the activity of receiver app invoked by IAC; (d). Inject malicious email address in the sender app to launch attack via IAC.

and DCL classes in both benign and malicious apps. These sensitive sources can reach Intent-sending APIs concealed by reflection/DCL to leak sensitive information, including device ID, subscriber ID, etc.

C. How effective is DINA?

In our experiment, we found some concealed IAC vulnerabilities that have been effectively detected by DINA, as presented in Table V. We have manually triggered these vulnerable IAC paths to verify that they can be activated at runtime, as described below.

Intent spoofing vulnerability is observed between *appinventor.ai_created4each.My_Diary* and *com.my.mail*. The receiver app manages the users' emails, which contains two components that can receive the implicit Intent *android.intent.action.SEND*. The sender app contains the method *ShareMessage()* that can be triggered through reflection, which initializes the Intent sending activity to configure the email setting for the receiver app. We modified this method to inject a specific email address, which can be used for phishing attacks. The complete attack process is shown in Fig. 4. The stealthy IAC initialized by the sender app cannot be detected by existing static and dynamic analyses. Moreover, the class of *ShareMessage()* requests to access the external storage, leading to serious privacy leakage. We scan the sender app (downloaded from official Google Play store) on VirusTotal, and is only detected by 2 engines out of 63.

Android component activation is observed between *com.example.qianbitou* and *com.axis.mobile*. The sender is an app providing services for used cars, including financial services. The app also implements reflection for invoking a method that activates an implicit Intent *android.intent.action.DIAL* to make a phone call to a hard-coded phone number. Therefore, any installed app (i.e. the receiver app) with components that have the matched Intent filter will be activated. The receiver app is a mobile banking app, whose component (*com.gtp.framework.UninstallShortcutReceiver*) will be activated to make random phone calls.

Information leakage vulnerability is observed between *com.hbg.coloring.fish* and *cn.jingling.motu.photowonder*. The sender is a gaming app, and the receiver app is an image editing app. The sender app contains a reflective call that instantiates an implicit Intent for sending pictures from the mobile storage. We also observed this concealed method invokes a sensitive API (*queryIntentActivities*) to obtain the running activities on the mobile device. The implicit Intent can be received by any app that contains *android.intent.action.SEND*. This vulnerable IAC path may lead to the leakage of sensitive images, and it can be very harmful when both apps are managed by one party. Another case is observed between *com.sogou.novel* and *com.gtp.nextlauncher.trial*. The sender app is a reader app, while the receiver app is used for 3D image processing. The reflection implemented in this app executes an implicit Intent *android.intent.action.SEND* for activating a broadcast receiver component of the receiver app. The implicit Intent sends information about a book, which can be easily replaced by sensitive information (e.g., bank accounts, location). This vulnerability can also be exploited to perform denial of service attack on the receiver app, by repeatedly invoking the implicit Intent to send the broadcast messages.

D. How efficient is DINA?

App stores including Google Play receive thousands of new apps every day, all of which require comprehensive security analysis. Therefore, we need efficient tools that can scale to the size of a large app market. We next report the running time of DINA's app analysis. We report the analysis time for both static analysis phase and dynamic analysis phase. The performance reported in this section was run on a Nexus 7 tablet connecting to a MacBook Pro laptop with Intel Core 2 Duo 2.4 GHz CPU and 8 GB memory. The static analysis was conducted on the laptop, while the dynamic analysis was performed on the tablet.

Fig. 5 shows the static analysis time with respect to the app bundle size in megabytes (MBs). With the growing app bundle sizes, the analysis time appears to be stable. For most of the bundles, DINA can finish the static analysis within 1 minute, demonstrating the efficiency of DINA's static analyzer. Fig. 6 presents the running time of dynamic analysis for $1,000$ real-world popular apps. The result shows that over 90% of apps can finish the dynamic analysis within 5 minutes. The majority of the dynamic analysis time is spent on running the apps to boost the coverage, and this time cost is inevitable for dynamic analysis tools. For complex apps with an average app size of 42 MBs, DINA can accomplish the dynamic analysis

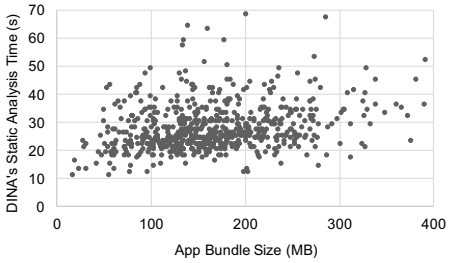


Fig. 5: Static analysis time.

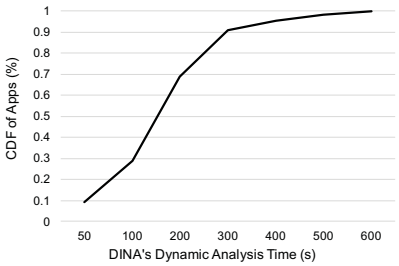


Fig. 6: Dynamic analysis time.

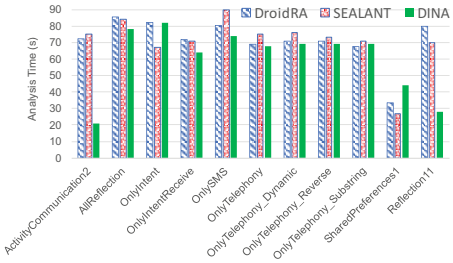


Fig. 7: Runtime performance comparison.

TABLE V: Risky vulnerabilities have been uncovered by DINA in real-world apps.

Sender app	# of Installs	Concealed method by reflection/DCL	Receiver app	Sensitive sink	Triggered component	Consequences
appinventor.ai_created4each.My_Diary	5,000,000+	ShareMessage()	com.my.mail	android.util.log	Activity	Intent Spoofing
com.example.qianbitou	N/A	toCall()	com.axis.mobile	java.lang.ProcessBuilder	Activity	Android Component Activation
com.hbg.coloring.fish	5,000+	shareImageOnTwitter()	cn.jingling.motu.photowonder	android.util.log	Activity	Information Leakage
com.sogou.novel	100,000+	ui.activity.MainNovelShelf.a()	com.gtp.nextlauncher.trial	android.util.log	Broadcast Receiver	Information Leakage

within 2.5 minutes/app on average as similar to the analysis time of HARVESTER, showing that DINA can be used for large-scale security analysis.

We further compared DINA's runtime performance with the state-of-the-art IAC analysis tools, i.e., SEALANT and DroidRA, using the set of benchmark apps (cf. Section IV-A). Fig. 7 shows the results of performance comparison. DINA achieves the best performance for the majority of the apps (8 out of 12), with an average analysis time of 1 minute/app.

V. Discussion and Limitations

Dynamic analysis coverage. Improving coverage has been a major challenge for dynamic analysis approaches [19]. In DINA, we currently utilize Monkey for input generation to exercise the targeted components. Although we achieve excellent IAC detection performance, we may still suffer from potential false negatives. We inherit some of the input generation limitations of Monkey. However, this fuzzing approach is still widely used by recent approaches that target DCL (e.g., DyDroid [13]), which relies on the observation that third-party libraries launch their DCL events when starting the app, which is sufficient for our analysis. Furthermore, the empirical study performed in [20] shows Monkey has achieved the best coverage among all analyzed input generation tools including DynoDroid [21] and PUMA [22].

IntelliDroid [23] is a recently-proposed input generation tool using event-chain detection and input injection with constraint solver. We integrate Intellidroid with DINA to replace Monkey. However, it fails to trigger most of the reflective/DCL calls. We found that most of the statically-identified paths related to reflective calls cannot be successfully triggered by Intellidroid, mainly due to the limited input type support, the limitations in the constrain solver, and its lack of support in dealing with environmental contexts/variables. Furthermore, a malicious app may perform emulator detection to halt its malicious activities during the analysis. DINA addresses this issue by performing the analysis on real devices.

IAC detection accuracy. Compared with static and dynamic tainting analysis based approaches, DINA does not perform precise data flow tracking analysis, which may lead to the imprecision of our detection results. Thus, the static analysis results may contain false positives. However, we use dynamic analysis to narrow down the scope of analysis on the methods that are dynamically executed. Therefore, we can effectively alleviate the imprecision brought by the lack of tainting analysis. One major benefit of our approach, however, is the improvement on runtime performance as shown in Section IV-D compared to other approaches.

Furthermore, existing static IAC analysis cannot handle an Intent that has been obfuscated in a manifest file. In such scenario, static analysis cannot identify app pairs by simply matching the Intents. This type of obfuscation makes static analysis to identify app pairs ineffective. By using dynamic analysis, DINA, on the other hand, can be extended to uncover app pairs during execution by performing runtime analysis to observe sent and received Intents by apps in the same bundle. This extension can be achieved by monitoring the *ActivityManager*, which we plan to implement in future work.

VI. Related Work

With respect to reflection and DCL, several research efforts attempt to improve soundness of static analysis in the presence of dynamically loaded code through Java reflection. DroidRA [2] adapts TamiFlex [12] to statically analyze Android apps for dynamically loaded code. Unlike TamiFlex, DroidRA does not execute apps; instead, it uses a constraint solver to resolve reflection targets. However, these static analysis approaches only work when reflection targets

can be identified from the source code, which is typically not the case in sophisticated security attacks, where a malicious component may be externally downloaded from hidden APKs. Our approach however detects reflection targets and captures dynamically loaded code using dynamic analysis.

With respect to ICC Analysis, numerous techniques have been proposed to focus on Intent and component analysis within the boundary of one app. IccTA [24] and its successor [25] leverage an Intent resolution scheme to identify inter-component privacy leaks. However, their approach relies on a preprocessing step connecting Android components through code instrumentation, which can lead to scalability issues [8], [11]. SEPAR [9] and SEALANT [11] perform compositional security analysis at a higher level of abstraction. While these research efforts are concerned with security analysis of component interactions between Android apps, DINA's analysis, however, goes far beyond single application analysis, and enables reflection and DCL-aware assessment of the overall security posture of a system, greatly increasing the scope of potential ICC-based misbehavior analysis.

VII. CONCLUSION

In this paper, we present DINA, a hybrid analysis approach for detecting malicious IAC activities in dynamically loaded code. DINA utilizes a systematic approach based on control-flow, data-flow, and method call graphs to identify malicious IAC activities across multiple apps. We have shown DINA can effectively resolve reflective and DCL calls at runtime for real-world apps. We demonstrate that multi-app, colluding attacks concealed by reflection and DCL can be launched to perform stealthy attacks, and evading existing detection approaches. In particular, we discover several popular real-world apps, which can trigger vulnerable IAC activities through reflection and DCL, leading to surreptitious privacy leakage. We have compared DINA with existing IAC vulnerability and reflection analysis tools. DINA analyzes most of apps in less than five minutes, and can identify malicious IAC behaviors concealed by reflective calls that no previous approach was able to detect. We believe further effort is required to better regulate the usage of reflection and DCL calls to close the attack avenues without undermining their utilities.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable comments and feedback. This work was supported in part by an NSF EPSCoR FIRST award, and the NSF grants CNS-1566388, CNS-1717898, CCF-1755890 and CCF-1618132. This work was also supported in part by RGC Project No. PolyU 152279/16E, CityU C1008-16G.

REFERENCES

[1] E. Tinaztepe, D. Kurt, and A. Gulec, "Android obad," COMODO, Tech. Rep., July 2013.

[2] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "DroidRA: Taming reflection to support whole-program analysis of android apps," in *Proc. of ISSTA*, 2016, pp. 318–329.

[3] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci, "Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications," in *Proc. of CODASPY '15*, 2015, pp. 37–48.

[4] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection - literature review and empirical study," in *Proc. of ICSE*, May 2017, pp. 507–518.

[5] Y. Xue, G. Meng, Y. Liu, T. H. Tan, H. Chen, J. Sun, and J. Zhang, "Auditing anti-malware tools by evolving android malware and dynamic loading technique," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 7, pp. 1529–1544, July 2017.

[6] A. Bosu, F. Liu, D. D. Yao, and G. Wang, "Collusive data leak and more: Large-scale threat analysis of inter-app communications," in *Proc. of ASIA CCS'17*, 2017, pp. 71–85.

[7] T. Li, X. Wang, M. Zha, K. Chen, X. Wang, L. Xing, X. Bai, N. Zhang, and X. Han, "Unleashing the walking dead: Understanding cross-app remote infections on mobile webviews," in *Proc. of CCS*, ser. CCS '17, 2017, pp. 829–844.

[8] Y. Tsutano, S. Bachala, W. Srisa-an, G. Rothermel, and J. Dinh, "An efficient, robust, and scalable approach for analyzing interacting android apps," in *Proc. of ICSE*, May 2017, pp. 324–334.

[9] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek, "Practical, formal synthesis and automatic enforcement of security policies for android," in *Proc. of DSN*, June 2016, pp. 514–525.

[10] M. Hammad, H. Bagheri, and S. Malek, "Determination and enforcement of least-privilege architecture in android," in *Proc. of ICSA*, April 2017, pp. 59–68.

[11] Y. K. Lee, J. y. Bang, G. Safi, A. Shahbazian, Y. Zhao, and N. Medvidovic, "A sealant for inter-app security holes in android," in *Proc. of ICSE*, May 2017, pp. 312–323.

[12] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proc. of ICSE*, 2011, pp. 241–250.

[13] Z. Qu, S. Alam, Y. Chen, X. Zhou, W. Hong, and R. Riley, "Dydroid: Measuring dynamic code loading and its security implications in android applications," in *Proc. of DSN*, June 2017, pp. 415–426.

[14] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks." in *Proc. of NDSS*, 2014.

[15] J. Blasco and T. M. Chen, "Automated generation of colluding apps for experimental research," *Journal of Computer Virology and Hacking Techniques*, pp. 1–12, 2017.

[16] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting inter-component privacy leaks in Android apps," in *Proc. of ICSE*, 2015, pp. 280–291.

[17] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques," in *Proc. of NDSS*, 2016.

[18] R. Hay, O. Tripp, and M. Pistoia, "Dynamic detection of inter-application communication vulnerabilities in android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 118–128.

[19] Y. Chen, W. You, Y. Lee, K. Chen, X. Wang, and W. Zou, "Mass discovery of android traffic imprints through instantiated partial execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, 2017, pp. 815–828.

[20] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *Proc. of ASE*. IEEE Computer Society, 2015, pp. 429–440.

[21] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 224–234.

[22] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, 2014, pp. 204–217.

[23] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware," in *Proc. of NDSS*, 2016.

[24] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "I know what leaked in your pocket: Uncovering privacy leaks on Android apps with static taint analysis," *CoRR*, vol. abs/1404.7431, 2014.

[25] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Apkcombiner: Combining multiple android apps to support inter-app analysis," in *Proceedings of the IFIP TC 11 International Conference*, 2015, pp. 513–527.