2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

DexLego: Reassembleable Bytecode Extraction for Aiding Static Analysis

Zhenyu Ning and Fengwei Zhang
COMPASS Lab
Wayne State University
{zhenyu.ning, fengwei}@wayne.edu

***Abstract*—The scale of Android applications in the market is growing rapidly. To efficiently detect the malicious behavior in these applications, an array of static analysis tools are proposed. However, static analysis tools suffer from code hiding techniques like packing, dynamic loading, self modifying, and reflection. In this paper, we thus present DexLego, a novel system that performs a reassembleable bytecode extraction for aiding static analysis tools to reveal the malicious behavior of Android applications. DexLego leverages just-in-time collection to extract data and bytecode from an application at runtime, and reassembles them to a new Dalvik Executable (DEX) file offline. The experiments on DroidBench and real-world applications show that DexLego correctly reconstructs the behavior of an application in the reassembled DEX file, and significantly improves analysis result of the existing static analysis systems.**

I. Introduction

With the rapid proliferation of malware attacks on mobile devices, understanding their malicious behavior plays a critical role in crafting effective defense. Static analysis tools are used to analyze malware and investigate their malicious activities [1]–[5]. However, malware writers can hide the malicious behavior by using an array of obfuscation techniques. The annual report from AVL team [6] shows that the number of Android packed applications has increased more than nine times, while about one third of them are packed malware. Typically, static analysis tools identify the malicious behavior of an application by investigating bytecode in Dalvik Executable (DEX) files. The packing technology replaces the original DEX file with a shell DEX file and dynamically releases the original DEX file at runtime. Additionally, the original DEX file is encrypted until its execution. While the free use of public packing platforms [7]–[12] provides a convenient and reliable protection for applications, the challenge of facing packed malware is rising. Static analysis tools are completely unarmed to the packed malware as they can only fetch the shell DEX file but not the encrypted original DEX file.

To address this problem, several unpacking systems are introduced recently [13], [14]. However, these systems are far from solving the problem completely. For instance, they assume that there is a point when all original code is unpacked in memory (i.e., a clear boundary or transition between the packer's code and the original code). However, the malware writers can pack code with advanced techniques that interleave the packing and unpacking processes. Moreover, recent studies show that sophisticated adversaries, known as self-modifying malware [15], [16], can modify the bytecode and other contents in a DEX file at runtime.

To further understand the self-modifying malware, consider Code 1 as an example. In Line 14, the native method, `bytecodeTamper`, modifies the bytecode of Lines 11 and 13. Note that the method `bytecodeTamper` is executed twice and performs different modifications to the two Lines during each iteration. There is a taint flow in Code 1, but the state-of-the-art static analysis tools [1]–[5] cannot detect it. Moreover, existing method-level unpacking systems [13], [14] are unable to reveal this taint flow because they cannot differentiate the actual executed code from the fake code (i.e., modified code like Lines 11 and 13 to hide taint flows), and we will discuss the details in Section IV-A.

Unlike the static analysis tools, the dynamic analysis tools [17]–[21] do not suffer from packing techniques. However, they have their own drawbacks. The automatic dynamic taint flow analysis tools [17], [18], [21] cannot handle implicit taint flows while static analysis tools [4], [5] can solve them. Moreover, the huge performance overhead makes it difficult to implement a complicated analysis mechanism, so there is a trade-off between the accuracy and performance. Meantime, the code coverage problem also threatens the accuracy of the dynamic analysis tools [19]–[21].

In this paper, we present DexLego, a novel program transformation system that reveals the hidden code in Android applications to analyzable pattern via instruction-level extracting and reassembling. DexLego collects bytecode and data when they are executed and accessed, and reassembles the collected result into a valid DEX file for static analysis tools. Since we extract all executed instructions, our system is able to uncover the malicious behavior of the packed applications or malware with self-modifying code. One of the key challenges in DexLego is to reassemble the instructions into a valid and accurate DEX file. Hence, we design a novel reassembling approach to construct the entire executed control flows including self-modifying code. Additionally, we implement the first prototype of force execution on Android and use it as our code coverage improvement module.

Moreover, our system helps static analysis tools improve the analysis accuracy on reflection-involved samples. The Java reflection obscures the control flows of the application by replacing the direct function call or field access with a call to the reflection library functions which take the name string

2158-3927/18/$31.00 Â©2018 IEEE
DOI 10.1109/DSN.2018.00075

```
1  package com.test;
2
3  public class Main extends Activity {
4    private static final String PHONE = "800-123-456";
5    protected void onCreate(Bundle savedInstanceState) {
6      // ...
7      advancedLeak();
8    }
9
10   public void advancedLeak() {
11     String a = getSensitiveData(); // source
12     for (int i = 0; i < 2; ++i) {
13       normal(a);
14       bytecodeTamper(i);
15     }
16   }
17
18   public void normal(String param) {
19     // do something normal
20   }
21
22   public void sink(String param) {
23     // send param through text message.
24    SmsManager.getDefault().sendTextMessage(PHONE, null,
          param, null, null); // sink
25   }
26
27   /* While i = 0:
28    *  modify Line 11 to String a = "non-sensitive data"
29    *  modify Line 13 to sink(a)
30    * While i = 1:
31    *  modify Line 11 to String a = getSensitiveData()
32    *  modify Line 13 to normal(a) */
33   public void native bytecodeTamper(int i);
34 }
```

Code 1: An Example of Self-Modifying Code.

of the function or field as parameter. Previous reflection solutions [22] and static analysis tools [1]–[3] on Android assume that the name strings of the reflectively invoked method and its declaring class are reachable. However, the name string can be encrypted in some cases [23] and the advanced malware could even use reflective method calls without involving any string parameter [24]. A solution on traditional Java platform [25] requires load-time instrumentation which is not supported in Android [1]. Thus, DEXLEGO implements a similar idea in Android and replaces the reflective call with direct call.

We evaluate DEXLEGO on real-world packed applications and DroidBench [24]. The evaluation result shows DEXLEGO successfully unpack and reconstruct the behavior of the applications. The F-measures (i.e., analysis accuracy) of FlowDroid [1], DroidSafe [3], and HornDroid [2] on DroidBench increase 33.3%, 31.1%, and 23.6%, respectively. Moreover, static analysis tools with the help of DEXLEGO provide a better accuracy than existing dynamic analysis systems TaindDroid [17] and TaindART [18]. The code coverage experiments on open source samples from F-Droid [26] show that our force execution module helps to improve the coverage of dynamic analysis and increases the coverage of state-of-the-art fuzzing tool, Sapienz [27], from 32% to 82%. The main contributions of this work include:

- We present DEXLEGO, a novel system that automatically transforms the hidden code in the Android applications to analyzable pattern. Our novel approach leverages tree structures to collect data/bytecode at runtime, and reassemble collected information back to DEX files, which makes the hidden code including packed or self-modifying one analyzable for current static analysis tools. To the best of our knowledge, this is the first system to reassemble the instruction-level tracing result of Java bytecode back to an executable file, and we consider this is the **key contribution** of this work.
- DEXLEGO mitigates the inaccuracy of static analysis tools on the reflection-involved samples by transforming the reflective method call to direct call regardless how the adversary uses it; it also improves the code coverage of dynamic analysis via our force execution module and. Moreover, DEXLEGO can be easily applied to Java application on x86 platforms and advances the traditional taint flow analysis.
- We implement a prototype of DEXLEGO in Android Runtime and evaluate the system in a real Android device. The experiment result shows that DEXLEGO successfully unpacks and reconstructs the hidden behavior of the real-world packed applications. By testing our system with state-of-the-art static analysis tools on DroidBench, we demonstrate that DEXLEGO improves the F-Measures of static analysis tools by more than 23%. Moreover, the comparison with existing dynamic analysis tools shows that DEXLEGO-assisted approach provides a more accurate result.
- The source code of DEXLEGO is publicly available at goo.gl/jpRvqu.

II. BACKGROUND

A. Dalvik and Android Runtime

Dalvik is a special Java virtual machine running in the Android system. It is used to interpret Android specified bytecode format since the first release of Android. To improve the performance, Google has introduced Just-In-Time (JIT) compilation and Ahead-Of-Time (AOT) compilation since Android 2.2 and Android 4.4, respectively. The JIT compilation continually compiles frequently executed bytecode slices into the machine code. As an upgrade, the AOT compilation compiles most bytecode in the application into the machine code during the installation. Dalvik equipped with AOT compilation is renamed to Android Runtime (ART). Since Android 5.0, Dalvik has been completely replaced by ART.

In both Dalvik and ART, the bytecode is organized in units of methods. The minimum code unit for JIT and AOT compilation is a method, indicating that a single method cannot contain both bytecode and machine code. Methods such as constructors and abstract methods require the bytecode interpreter even in ART. Moreover, a single method or the entire ART can be configured to run in the interpreter mode.

B. Android Java Bytecode

The Java bytecode in Android is chained by instructions. Each instruction contains an opcode and arguments related to the opcode. The opcodes are different from the ones in regular Java bytecode and the bit-length of an instruction varies according to the opcode. In the interpreter, instructions are listed in an array of 16-bit (2 bytes) units. An instruction

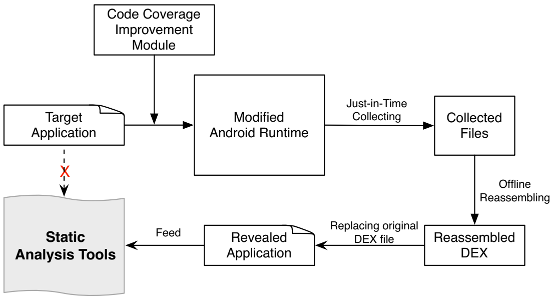

Fig. 1: Overview of DEXLEGO.

occupies at least one unit with a maximum number of units up to five.

III. SYSTEM OVERVIEW

As Figure 1 shows, instead of directly feed the target application to static analysis tools, we firstly execute the target application with DEXLEGO. In executing, we use Just-in-Time (JIT) collection to extract data/instructions and output them to files right before used by ART. In the meantime, we use a code coverage improvement module to increase the code coverage. Next, we reassemble the collected files to a DEX file and use the reassembled DEX file to replace the one in the original APK. Finally, the new APK file is fed to the static analysis tools. The architecture of DEXLEGO contains three main components: 1) the collecting component that collects bytecode and data, 2) the offline reassembling component that reassembles a new DEX file based on the collection result, and 3) the code coverage improvement module that helps DEXLEGO to achieve a high code coverage. Next, we will discuss the three components respectively.

A. Bytecode and Data Collection

Figure 2 shows the JIT collection we used in DEXLEGO. During the execution of an application, ART firstly extracts the DEX file from the original APK file and passes it to the class linker. The class linker then loads and initializes the classes in the DEX file, and our JIT collection method collects the metadata of the class (e.g., super class) at this point. Next, when a method is invoked, ART extracts its bytecode from the DEX file, and leverages the interpreter to execute them. The interpreter fetches the entire bytecode (organizing in a 16-bit array) of the method and executes the bytecode instructions one by one. Thus, according to our JIT policy, we collect the executed instructions of the method and their related objects (e.g., `string`) via instruction-level extracting. Note that the execution of the code in the dynamic loaded DEX file also follows the same flow.

The state-of-the-art static analysis tools do not accept machine code as their input. However, ART executes most methods based on the machine code, and the translation from the machine code to the bytecode is a challenging task. To simplify the task, DEXLEGO configures all methods in the application to be executed by the interpreter.

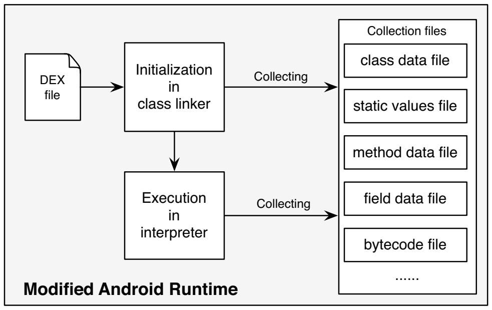

Fig. 2: Just-in-Time Collection.

B. DEX File Reassembling

After the collecting, all the output files are reassembled to a new DEX file offline following the format of a DEX file, and we replace the DEX file in the original APK file with the reassembled one. The modified APK file is finally fed to static analysis tools to study the malicious behavior.

This reassembling is not trivial, and we consider this is the key contribution of this work. In the DEX file format, each method contains only one instruction array. However, due to different control flows (e.g., execution is led to different branches of a branch statement) or self-modifying code, one method may contain different instruction arrays in the collection stage. To correctly combine the collected instructions, we thus design a tree model and a novel collecting and reassembling mechanism. More details are discussed in Section IV-A and Section IV-B.

C. Code Coverage Improvement Module

To improve the code coverage of dynamic analysis systems, there already exists a series of tools or theories like: 1) Input generators or fuzzing tools [28]–[32], 2) Symbolic or concolic execution [23], [33]–[37] based systems, 3) Force execution [38]–[40] based systems. Our code coverage improvement module can be one of them or a combination of them. Note that most of the systems mentioned in 1) and 2) are implemented in Android, and we can directly use them to conduct the execution of the target application with little engineering effort. However, to the best of our knowledge, the idea of force execution has not been applied on Android platform. Thus, we implement a prototype of force execution as a supplement of our code coverage improvement module.

To use force execution in DEXLEGO, we identify the Uncovered Conditional Branches (UCB) and calculate the path to each UCB. By monitoring and manipulating the branch instructions in the interpreter, we force the control flow to go along the calculated path to reach each UCB.

IV. DESIGN AND IMPLEMENTATION

We implement DEXLEGO in an LG Nexus 5X with Android 6.0. Based on the Android Open Source Project [41] (AOSP), we build a customized system image and flash it into the device by leveraging a third-party recovery system [42].

A DEX file consists of data structures that represent different data types used by the interpreter [43]. DexLego collects these data structures directly from memory while they are used by ART at the runtime. Moreover, we leverage instruction-level tracing to collect executed instructions and reassemble them back to a `method` structure. In this section, we discuss 1) bytecode collection, 2) bytecode reassembling, 3) data collection, and 4) DEX file reassembling separately. The approaches to handle reflection and force execution are also discussed in this section.

A. Bytecode Collection

In ART, after the instruction array of a method is passed to the interpreter, the interpreter executes the instructions one by one following the control flow indicated by them. To expose the behavior of the method, DexLego aims to collect all instructions executed in the method. However, existing systems [13], [14] that use method-level collection cannot defend against dynamic bytecode modification, and the detailed limitation is described as below.

Inadequacy of Method-level Collection. Consider Code 1 as an example. While entering the method `advancedLeak`, the smali code [1] of the method is represented by Code 2. After the first execution of the native method `bytecodeTamper`, the code of the method `advancedLeak` is modified to Code 3. In Code 3, the native method has modified the bytecode to hide the source (Lines 2-4 are changed from Code 2 to Code 3), but the sensitive data is already stored in the register `v0`. During the second execution of the `for` loop, the sensitive data in the register `v0` is leaked through the method `sink` (Lines 9-10 in Code 3). Then, the native method resumes the code back to Code 2. The instruction array of the method `advancedLeak` in memory is either Code 2 or 3 at any time point (e.g., before and after JNI code), which means that the method-level collection (e.g., DexHunter [14] and AppSpear [13]) can only collect Code 2 or 3 even when multiple collections are involved. However, in the static taint flow analysis, the red lines in Code 2 (Lines 2-4) represent a source, but the data fetched from the source are sent to the blue lines (Lines 9-10) which are not a sink. In Code 3, the red lines (Lines 9-10) are a sink, but the received data are obtained from the blue lines (Lines 2-4) which are not a source. Thus, the leak of the sensitive data can be identified from neither Code 2 nor Code 3, and the key reason is that the code representing the source and sink are modified on purpose to hide the taint flow. AppSpear claims that it implements an instruction-level tracing mechanism, however, as we will explain below, simply tracing the instructions does not satisfy the requirement of static analysis tools.

Instruction-level Collection and Tree Model. In light of the shortcoming of method-level collection as described above, the DexLego leverages instruction-level collection to defend against self-modifying code such as Code 1. One simple approach for instruction-level collection is to list all the executed instructions one by one; however, this approach leads to a code scale issue. Take the loop as an example, since the instructions in a loop are executed for multiple times, the simple approach would lead to a large number of repeating instructions. Moreover, the branch statements and self-modifying code make it possible that different executions of a single method lead to different instruction sequences. However, the format of the DEX file [43] allows only one instruction sequence for a single method.

```
1  .method public advancedLeak()V
2    invoke-virtual  p0 , \
3      Lcom/test/Main;->getSensitiveData()Ljava/lang/String;
4    move-result-object v0
5    const/4 v1, 0
6    :L0
7    const/4 v2, 2
8    if-ge v1, v2, :L1
9    invoke-virtual  p0, v0 , \
10     Lcom/test/Main;->normal(Ljava/lang/String;)V
11   invoke-virtual { p0, v1 }, \
12     Lcom/ecspride/Main;->bytecodeTamper(I)V
13   add-int/lit8 v1, v1, 1
14   goto :L0
15   :L1
16   return-void
17 .end method
```

Code 2: Smali representation of the method `advancedLeak` while entering and leaving it.

```
1  .method public advancedLeak()V
2    const-string v0, "non-sensitive data"
3    nop
4    nop
5    const/4 v1, 0
6    :L0
7    const/4 v2, 2
8    if-ge v1, v2, :L1
9    invoke-virtual  p0, v0 , \
10     Lcom/test/Main;->sink(Ljava/lang/String;)V
11   invoke-virtual { p0, v1 }, \
12     Lcom/ecspride/Main;->bytecodeTamper(I)V
13   add-int/lit8 v1, v1, 1
14   goto :L0
15   :L1
16   return-void
17 .end method
```

Code 3: Smali representation of the method `advancedLeak` after the first execution of the method `bytecodeTamper`.

To address the code scale issue, DexLego eliminates repeating instructions by comparing the instructions with same indices. As mentioned above, the bytecode of a method is organized in a 16-bit unit array and passed to the interpretation functions (`ExecuteSwitchImpl` and `ExecuteGotoImpl` functions). In these functions, the interpreter uses a variable `dex_pc` to represent the index of the executing instruction in the array. In light of this, we identify repeating instructions by comparing the executing instructions with the same `dex_pc` values. Moreover, the self-modifying code can also be identified by the comparison. Different instructions with the same `dex_pc` value actually indicate a runtime modification.

Algorithm 1 illustrates the comparison-based instruction collection algorithm, and Figure 3 shows the related data structures. We consider the first execution of an instruction

[1] The smali code is a more readable format of the bytecode.

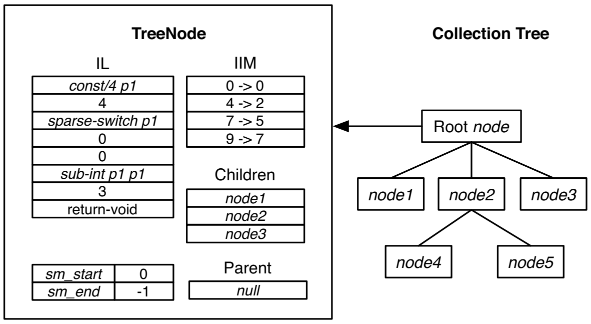

Fig. 3: Data Structure Storing All Instructions in a Method During a Single Execution. The right tree structure shows the collection result for a method during a single execution. The left rectangle describes the data structure of each tree node. For each execution of a method, we generate a collection tree.

Algorithm 1 Bytecode Collection Algorithm

```
procedure BytecodeCollection
    create node $root$
    $current = root$
    for each executing instruction $ins$ do
        let index of $ins$ be $dex\_pc$
        if $dex\_pc$ exists in $current.IIM$ then
            $pos\_in\_IL = current.IIM.get(dex\_pc)$
            $old\_ins = current.IL.get(pos\_in\_IL)$
            if !SameIns($ins$, $old\_ins$) then
                create a child node $child$
                $child.parent = current$
                $child.start\_pos = dex\_pc$
                $current = child$
            else
                $continue$
            end if
        else if $current$ has a parent then
            $parent = current.parent$
            if $dex\_pc$ exists in $parent.IIM$ then
                $pos\_in\_IL = parent.IIM.get(dex\_pc)$
                $old\_ins = parent.IL.get(pos\_in\_IL)$
                if SameIns($ins$, $old\_ins$) then
                    $current.end\_pos = dex\_pc$
                    $current = parent$
                    $continue$
                end if
            end if
        end if
        $pos\_in\_IL = current.IL.size()$
        $current.IL.add(ins)$
        $current.IIM.push(pair(dex\_pc, pos\_in\_IL))$
    end for
end procedure
```

as a baseline and any different instructions with the same dex_pc value as a divergence branch. Thus, each divergence branch indicates a piece of self-modifying code. Note that self-modifying code might also exist in the divergence branch (like multiple layers of self-modifying). The divergence branches in a method then form a tree structure. The right part of Figure 3 shows an example of the final collecting result. Nodes 1-3 represent three pieces of self-modifying code on the root node, and Nodes 4-5 represent two pieces of self-modifying code on Node 2. The left rectangle in Figure 3 shows the TreeNode structure which represents a node in the tree structure. The Instruction List (IL) in the structure includes the list of executed instruction and their metadata. The instructions in IL are recorded by the order of their first execution and the IL plays the role of baseline in the node. The dex_pc value of an instruction may be different from its index in IL due to branch statements, and we use an Instruction Index Map (IIM) to maintain the mapping between the instruction's dex_pc value and its index in IL for further comparison. sm_start and sm_end indicate the starting and ending dex_pc value of the divergence branch, while parent and children represent the parent and all children of the node, respectively. With the tree structure, DexLego records all executed instructions in a single execution of a method and maintains the code size similar to the original instruction array.

In Algorithm 1, we only update one node during the execution of a single instruction, and this node is considered as the current node. DexLego creates an empty root node as the current node while entering a method. Once an instruction is executed, we check IIM of the current node to find whether the dex_pc value of this instruction has been recorded. If it does not exist in IIM, DexLego pushes the instruction into IL and updates IIM. If the dex_pc value already exists in IIM, we add a check procedure to find whether the instruction is the same as the one we recorded before. A positive result means that the same instruction in the same position is executed again, and DexLego does not record it. In contrast, the negative result indicates that modification has occurred to this instruction since its last execution. Then, we create a child node of the current node to represent the divergence branch, and the new node becomes the current node. After that, DexLego treats the instruction as a new instruction and pushes it into IL of the current node. In a divergence branch, another check procedure is added to each instruction, and this check procedure aims to identify whether the current divergence branch converges to its parent. If the same instruction with the same dex_pc value has been found in the parent's IL, we consider that the divergence branch converges back to its parent (e.g., current layer of self-modifying code ends) and make the parent node to be the new current node.

Listing 1 shows a high-level semantic view of the collection result of the method advancedLeak in Code 1. When Line 13 in Code 1 is executed for the first time, an invocation of the method normal is recorded. Then, in the second run, an invocation of the method sink is detected. However, by comparing with the recorded instructions, DexLego finds that it is a divergence point. A child node is forked and the instruction is pushed into the IL of the child node. Furthermore, a convergence point is found when Line 14 is executing. Thus, the collection tree contains a root node and a child node, and the child node contains only one instruction. With the tree, the executed instructions and the control flows in the method are well maintained. Note that the modification to the Line 11 is ignored since the modified instructions are never executed.

For the issue of multiple instruction sequences for a single

```
Root Node:
  String a = getSensitiveData();
  for (int i = 0; i < 2; ++i) {
    normal(a);
    bytecodeTamper(i);
  }

Child Node: (Line 13 in Code 1)
  sink(a);
```

Listing 1: High-level Semantic View of the Collection Result of the Method `advancedLeak` in Code 1.

method, we generate multiple collection trees for multiple executions of the method and keep only the unique trees. The trees are further combined together with the approach detailed in Section IV-B.

B. Bytecode Reassembling

The offline reassembling-phase merges the collected trees into a DEX file while holding all the executed instructions and control flows. There are two steps in this phase: 1) converting each tree into an instruction array. 2) merging instruction arrays into the DEX file.

Converting a Tree into an Instruction Array. Each node in the collection tree generated from the collection phase contains an independent Instruction List (IL), and the goal of this phase is to combine the ILs in the nodes together without losing any control flows or instructions. To simplify the combination process, we traverse the nodes with the bottom-up fashion since the leaf nodes contain no child node.

To merge a single leaf to its parent, DexLego inserts an additional branch instruction in the divergence point (indicated by `sm_start`, self-modifying start, as defined in the above subsection IV-A), with one branch of the instruction pointing to the leaf. To make both conditional branches reachable, the conditional expression of the added branch instruction is calculated based on a static field of an instrument class with random values. Note that the random value produces indeterminacy problem on the additional branch instruction, and we consider it acceptable since the static analysis tool will take both branches of the instruction as reachable.

Once the leaf nodes are recursively merged into their parents, the root node becomes a complete set of the collected instructions including different control flows triggered during the execution.

Code 4 demonstrates the reassembled result of Listing 1. The static field `com_test_Main_advancedLeak_0` in our instrument class `Modification` indicates the divergence point in Line 13 of Code 1. When this result is fed to static analysis tools, they treat both `normal` and `sink` as reachable and detect the taint flow from sensitive data to text message in Code 1.

Merging Instructions Arrays. For each executed method, the previous phase outputs unique instruction arrays which indicate different executions of the method. Similar to the approach discussed above, we create a method variant for each instruction array and use additional branch instructions to cover different method variants.

```
String a = getSensitiveData();
for (int i = 0; i < 2; ++i) {
  if (Modification.com_test_Main_advancedLeak_0) {
    normal(a);
  } else {
    sink(a)
  }
  bytecodeTamper(i);
}
```

Code 4: Reassembled Result of the Method `advancedLeak` in Code 1.

C. Data Collection and DEX Reassembling

As mentioned in Section III-A, besides bytecode instructions, DexLego uses JIT collection to collect the metadata of DEX file. The collected data is written into collection files and further used to reassemble a new DEX file offline.

In Code 1, before any method or field in `Main` is accessed, the class `Lcom/example/Main;` is loaded and initialized. During the process, we firstly store string `Lcom/example/Main;` into a `string` structure and record the index of this `string` structure. Then with the index, a `type` structure is constructed and stored. Finally, a corresponding `class` structure related to the `type` is extracted. The collection occurs again when the class is initialized. The initialization procedure links the methods and fields to the class, and initializes the static fields. In Code 1, methods `onCreate`, `advancedLeak`, `normal`, and `sink` are linked to the class. While the static field `PHONE` is initialized, DexLego stores its name `PHONE`, type `Ljava/lang/String;` and initial value `800-123-456`. Lastly, a `field` structure is created and recorded. The `method` structures and the bytecode inside them are collected before and during the execution of the methods, respectively.

After the collection process, all collection files including bytecode are combined offline according to the format of the DEX file. Finally, we leverage the Android Asset Packaging Tool integrated with Android SDK to replace the DEX file in the original APK file with the reassembled one. To verify the soundness of our extracting and reassembling algorithm, we perform extensive tests against real-world applications, and the evaluation results in Section V-A, Section V-B, and Section V-D show that the reassembled DEX file retains the semantics of the real-world application and can be correctly processed by the state-of-the-art static analysis tools.

D. Handling Reflection

Currently, reflection is a serious obstacle for static analysis tools, and even the state-of-the-art static analysis tools [1]–[3], [23] cannot provide a precise result when reflection is involved in an application. FlowDroid [1], DroidSafe [3], and HornDroid [2] can solve the reflection only when the parameters are constant strings. However, the name string can be encrypted in some cases [23], and advanced malware can use reflection without involving any string parameter [24].

The TamiFlex [25] system on traditional Java platform uses load-time instrumentation to log reflective method calls and transform them to direct calls at offline. However, the required

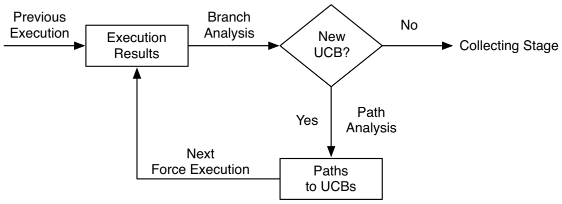

Fig. 4: Iterative Force Execution.

load-time instrumentation class `java.lang.instrument` is not supported in Android [1]. Meanwhile, since the target of the reflective method calls is parsed in ART at runtime, DEXLEGO actually knows the target of each reflection. Thus, we apply the similar idea in ART by replacing the reflection calls with direct calls in the collecting stage.

E. Force Execution

As a supplement of the code coverage improvement module, we implement a prototype of force execution which executes the target application in an iterative fashion. Note that our force execution starts from the execution result of the previous execution, and the previous execution could be any kind of execution like fuzzing, symbolic execution, another force execution, or simply open the application and close. Figure 4 shows the workflow of the iterative force execution. In each iteration, we first use branch analysis to identify the Uncovered Conditional Branch (UCB) from the result of the previous execution. Next, we calculate the control flow path to each UCB. A path to an UCB consists of branch instructions and the offsets of the conditional branches leading to the UCB. We save each path into a file and use these files as the input of the next iteration together with the original application. Finally, in the interpretation functions, the outcome of the corresponding conditional expression is automatically manipulated at runtime following the path files. With this approach, DEXLEGO ensures that the runtime control flow goes along the path to the UCB. If no more new UCB are generated after the iteration, we terminate the execution and continue the collecting stage. Otherwise, the next iteration is scheduled.

Since the idea of force execution breaks the normal control flow of the original application, the application may crash due to the control flow falls to an infeasible path [39], [40]. To avoid crash triggered by force execution, we monitor the unhandled exception in the interpreter and tolerate it by directly clear the exception. This strategy helps us to avoid terminations due to infeasible paths while does not affect our runtime bytecode and data collection.

V. EVALUATION

In this section, we evaluate DEXLEGO with DroidBench [24] and real-world applications downloaded from Google Play and other application markets. In particular, we aim to answer five research questions:

RQ1. Can we correctly reconstruct the behavior of apps?
RQ3. How is DEXLEGO compared with other tools?
RQ4. Can DEXLEGO work with real-world packed apps?

TABLE I: Test Result of Different Packers.

Applications	HTMLViewer	Calculator	Calendar	Contacts
# of Instructions	217	2,507	78,598	103,602
360 [11]	✓	✓	✓	✓
Alibaba [7]	✓	✓	✓	✓
Tencent [12]	✓	✓	✓	✓
Baidu [8]	✓	✓	✓	✓
Bangcle [45]	✓	✓	✓	✓
NetQin [46]	The service is offline now			
APKProtect [47]	Unresponsive to packing requests			
Ijiami [9]	Samples are rejected by human agents			

RQ5. What is the coverage of our force execution prototype?
RQ6. What is the runtime performance overhead?

A. RQ1: Test with Open-source Apps and Public Packers

To verify the correctness of the reassembled result, we pick up four open source applications (i.e., `HTMLViewer`, `Calculator`, `Calendar`, and `Contacts`) from AOSP [41] and use DEXLEGO to reveal them. By manually comparing the instructions and control flows in each method, we ensure that the instructions and control flows in the source code are completely included in the reassembled result. In regard to `Calendar` and `Contacts`, we use Soot framework [44] to build a complete call graph since the numbers of instructions (78,598 and 103,602 instructions, respectively) are too large for a manual analysis. By examining the call graph, we confirm that the control flows in these two applications are properly maintained in the reassembled DEX.

Next, to check the functionality against packers, we use different public packing platforms to pack these applications and then use DEXLEGO to reveal them again. Table I shows the result of the experiments. For the packers including 360 [11], Alibaba [7], Tencent [12], Baidu [8], and Bangcle [45], DEXLEGO succeeds in both collection and reassembling stages. By using the same approach described above, we ensure that DEXLEGO correctly rebuilds the behavior of each application. Note that NetQin packer [46] mentioned in AppSpear [13] is no longer available. The APKProtect [47] is unresponsive to the packing requests, and there are no logs of the occurred errors. The packing service provided by Ijiami [9] requires manual audits by their agents, and they reject our applications for the reason that the applications are not actually developed by us.

B. RQ2: Test with Existing Tools

1) Static Analysis Tools: DroidBench [24] is a set of open-source samples that leak sensitive data in various ways. It is considered as a benchmark for Android application analysis and widely used among recent analysis tools [1]–[5]. The latest release of DroidBench contains 119 applications, including both leaky and benign samples. The leaky samples leak a variety of sensitive data fetched from sources (API calls that fetch sensitive information) to sinks (API calls that may leak information), and the benign samples contain no such information flows. As a supplement, we contribute another 15

TABLE II: Analysis Result of Static Analysis Tools. The columns in ”Original” represent the analysis result of the original samples, and the columns in ”DexLego” represent that of the samples reassembled by DexLego. The column ”TP” and ”FP” indicate the number of true positives and false positives of the analysis result, respectively.

	# of Samples	# of Malware	Original		DexLego	
			TP	FP	TP	FP
FlowDroid [1]	134	111	81	10	95	4
DroidSafe [3]	134	111	95	12	105	7
HornDroid [2]	134	111	98	9	106	4

samples involving usage of advanced reflection (5 samples), dynamic loading (3 samples), self-modifying (4 samples), and unreachable taint flows (3 samples). Current static analysis tools [1]–[3] cannot precisely analyze these newly added samples. Besides this benchmark, we choose three representative static analysis tools (FlowDroid [1], DroidSafe [3], and HornDroid [2]) to conduct the experiments.

Our experiment involves 134 samples (119 samples in the newest release plus 15 samples we contributed) in DroidBench. Since the lines of code in DroidBench samples are small, we simply choose the state-of-the-art fuzzing tool Sapienz [27] to generate the inputs for the execution. We first use the static analysis tools to analysis the original samples and the samples processed by DexLego, and the result is shown in Table II. The table shows that DexLego increases more than 8 true positives by resolving advanced reflections, extracting self-modifying code and dynamic loading code. Moreover, The JIT collection ensures that the extracted data reflects the performed behavior of the target application. Thus, at least 5 false positives introduced by dead code blocks are removed. Next, without losing generality, we use one of the most popular packers tested in Section V-A, 360 packer, to pack the original samples and process the packed samples with DexLego, DexHunter [14], and AppSpear [13], respectively. The analysis result of the processed samples is shown in Table III. Note that DexHunter and AppSpear lead to the same result since they can extract the original DEX files and the result is same as analyzing the original DEX. Compared to DexLego, they fail to deal with self-modifying code and reflection. As shown in the table, DexLego provides more than 5 true positives and reduces more than 5 false positives than DexHunter and AppSpear. We note that DexLego fails to cover taint flow in only one application among all samples. In this sample, sensitive data only leaks in the tablet, and it cannot be detected as we execute it in a mobile phone.

$$\begin{aligned} Sensitivity = \frac{tp}{tp+fn},\ Specificity = \frac{tn}{tn+fp}, \\ F\text{-}Measure = 2 \times \frac{Sensitivity \times Specificity}{Sensitivity + Specificity} \end{aligned} \tag{1}$$

TABLE III: Analysis Result of Packed Samples. The columns in ”DH”, ”AS”, and ”DexLego” represent the analysis result of the samples processed by DexHunter [14], AppSpear [13], and DexLego, respectively. The column ”TP” and ”FP” indicate the number of true positives and false positives of the analysis result, respectively.

	# of Samples	# of Malware	DH [14] / AS [13]		DexLego	
			TP	FP	TP	FP
FlowDroid [1]	134	111	84	10	95	4
DroidSafe [3]	134	111	98	12	105	7
HornDroid [2]	134	111	101	9	106	4

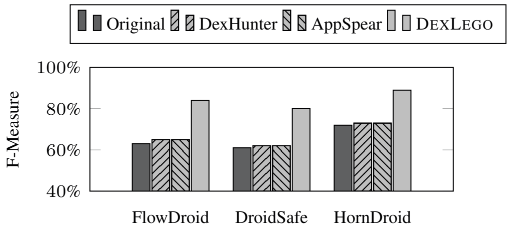

Fig. 5: F-measures of Static Analysis Tools.

The F-Measure [2] is a standard measure of the performance of a classification, and it is calculated by Formula (1). Figure 5 illustrates the changes of F-Measures after involving DexHunter, AppSpear, and DexLego. Once DexLego is involved, the F-Measure of FlowDroid increases from 63% to 84% on DroidBench, and that of DroidSafe increases from 61% to 80%. In regard to the most recent static analysis tool, HornDroid, the F-Measure increases from 72% to 89%. The percentages of incremental values are 33.3%, 31.1%, and 23.6%, respectively. In the meantime, the improvement introduced by DexHunter and AppSpear is less than 3%.

2) Dynamic Analysis Tools: As mentioned in Section I, dynamic analysis tools can be circumvented through implicit taint flows, and a recent work [23] shows that a representitive dynamic analysis tool, TaintDroid [17], misses leakage on some samples of DroidBench. We pick these samples and analyze them with both TaintDroid and another recent dynamic analysis tool TaintART [18]. Next, we use DexLego to analyze it again. The reassembled result is fed to HornDroid, the most recent static analysis tool, for comparison.

Table IV shows the taint flow analysis results of TaintDorid, TaintART, and combing DexLego and HornDroid. As shown in Table IV, the static analysis result of reassembled APK file by DexLego detects the taint flows and is more precise than dynamic analysis tools. In `Button1` and `Button3`, the sensitive data are leaked via callback methods, and we solve it properly while the dynamic analysis tools miss it. As TaintDroid executes applications on emulator, the sample `EmulatorDetection1` evades the analysis. Both TaintDroid and TaintART cannot detect the implicit taint flows in `ImplicitFlow1`, and using HornDroid with DexLego provides a precise analysis result. One of the taint flows in `PrivateDataLeak3` leaks the sensitive data through writing/reading an external file, and all tested tools fail to detect this flow since they do not take this case into account. Note that these missed taint flows are not caused by code coverage issue, but due to the weakness of dynamic analysis

TABLE IV: Analysis Result of Dynamic Analysis Tools and DexLego. The columns ”TD” and ”TA” represent the taint flows detected by TaintDroid [17] and TaintART [18], respectively. The last column shows the detected taint flows by feeding the revealed result of DexLego to HornDroid [2].

Samples	Leak #	# of Leak Detected		
		TD [17]	TA [18]	DexLego + HD [2]
Button1	1	0	0	1
Button3	2	0	0	2
EmulatorDetection1	1	0	1	1
ImplicitFlow1	2	0	0	2
PrivateDataLeak3	2	1	1	1

tools on implicit taint flows.

Note that DexLego is not a dynamic analysis tool. We believe we should not directly compare DexLego with dynamic analysis tools, and the dynamic analysis tools have their advantages. However, the experiment conducted in this subsection is to show that DexLego can help static analysis tools make up some deficiencies of dynamic analysis tools.

C. RQ3: Test with Real-world Packed Applications

A previous work [48] has downloaded more than one million applications from Google Play by a crawler in 2014, and we select the packed applications from this set. Since the DEX file in an application packed by the public packing platforms contains only the classes needed to unpack the original DEX file, the number of the classes in the DEX file is less compared to normal applications. In light of this, we perform a coarse-grain analysis to screen the applications which contains less than 50 classes from the top rated $10,000$ applications. Next, we select the first 9 applications from the screened result by manually checking and reverse engineering. Without loss of generality, we download the latest version of these applications from three different popular application markets: 1) Google Play [49] (denoted as set A), 2) 360 Application Market [50] (denoted as set B), and 3) Wandoujia Application Market [51] (denoted as set C).

For these real-world packed applications, we use FlowDroid to provide a quick scan on the original applications, and then execute them with DexLego for 5 minutes. Next, the reassembled APK file is analyzed again by FlowDroid. Table V shows the result of our experiment. Although no taint flow can be detected from the original samples, FlowDroid detects several taint flows from these revealed applications. From the analysis result, we find that all of these applications send device ID (IMEI number) to remote servers. Moreover, three of them leak location information and two of them leak SSID. This result also shows that DexLego successfully reveals the latest packed real-world applications.

TABLE V: Analysis Result of Packed Real-world Applications. The column ”Sample Set” is defined in Section V-C, which indicates the source of the application. The column ”# of Installs” shows the installation number provided by the application markets. The column ”Original” represents the number of detected taint flows in the original application while the column ”Revealed” is the number of detected taint flows in the revealed APK file.

Package Name	Version	Sample Set	# of Installs	Original	Revealed
com.lenovo.anyshare	3.6.68	A	100 million	0	4
com.moji.mjweather	6.0102.02	A	1 million	0	5
com.rongcai.show	3.4.9	A	100 thousand	0	3
com.wawoo.snipershootwar	2.6	B	10 million	0	4
com.wawoo.gunshootwar	2.6	B	10 million	0	5
com.alex.lookwifipassword	2.9.6	B	100 thousand	0	2
com.gome.eshopnew	4.3.5	C	15.63 million	0	3
com.szzc.ucar.pilot	3.4.0	C	3.59 million	0	5
com.pingan.pabank.activity	2.6.9	C	7.9 million	0	14

D. RQ4: Code Coverage

To evaluate the code coverage of our force execution engine, we pick up five open source applications from the random page [52] of F-Droid [26] project. For each application, we first execute it with Sapienz [27] and use Java Code Coverage Library (JaCoCo) [53] for Android Studio to calculate the coverage. Next, based on the result of Sapienz, we execute it again using the force execution engine as the code coverage improvement module.

Table VI shows the details of the samples including package name, version number, the number of instructions, and the total size of the dump files after fuzzing by Sapienz. Note that the size of the dump files is not only related to the number of the instructions in the application, but also related to the size of other data structures in the DEX file (e.g., number of classes, number of methods, size of strings, and so on.) and the code coverage of the fuzzing. Table VII shows the average coverage of these samples with different granularities. The results show that the force execution significantly improves the coverage and achieves an average instruction coverage of 82%. By manually check the source code, we group the cause of missed instructions into three main categories: 1) Dead code blocks. As an example, the `CmdTemplate` class is never involved in the application `be.ppareit.swiftp`, thus the entire instructions in this class are not included while calculating coverage. 2) Native crashes. Although DexLego clears the unhandled exceptions in the interpreter, the abnormal control flows may lead the native code to crash. This may be mitigated by the on demand runtime memory allocation mechanism applied in [40]. 3) Instructions in exception handlers. During force execution, the expected exceptions in the `try-catch` blocks may not be thrown due to abnormal control flow, and it may be solved by treating these blocks as branch instructions in the branch analysis. We leave it as a future work.

E. RQ5: Performance

As DexLego traces and extracts instructions at runtime, it slows the ART during instruction execution. To learn the performance overhead introduced by DexLego, we use CF-Bench [54] to compare the performance of the unmodified ART and ART with DexLego. For each environment, we run CF-Bench for 30 times, and the results are presented in

TABLE VI: Samples from F-Droid [26].

Package Name	Version	# of Instructions	Dump File Size
be.ppareit.swiftp	2.14.2	8,812	47.26 KB
fr.gaulupeau.apps.InThePoche	2.0.0b1	29,231	771.81 KB
org.gnucash.android	2.1.7	56,565	2.40 MB
org.liberty.android.fantastischmemopro	10.9.993	57,575	1.55 MB
com.fastaccess.github	2.1.0	93,913	3.18 MB

TABLE VII: Code Coverage with F-Droid Applications.

	Class	Method	Line	Branch	Instruction
Sapienz [27]	44%	37%	32%	20%	32%
Sapienz + DexLego	87%	88%	82%	78%	82%

Figure 6. A higher score indicates a better performance. It shows that DexLego brings 7.5x, 1.4x, 2.3x overhead on Java score, native score, and overall score, respectively.

Moreover, we evaluate the launch time of three popular applications (i.e., Snapchat, Instagram, and WhatsApp) downloaded from Google Play. While an activity in an application is launching, the `ActivityManager` reports the time usage for initializing and displaying. We launch each application for 30 times and the result is summarized in Table VIII. The result shows that DexLego introduces about two times slowdown on the launch time, and this result matches the overall overhead tested by CF-Bench.

Since our system is designed for security analyst instead of traditional users, we do not take performance as a critical factor. In summary, we consider the overhead is acceptable and leave the further improvement as our future work.

VI. Related Work

A. Static Analysis Tools

FlowDroid [1] is a static taint-analysis tool for Android applications, and it achieves a high accuracy by mitigating the gaps between lifecycle methods and callback methods. Amandroid [5] and IccTA [4] aim to resolve the implicit control flows during inter-component communication. EdgeMiner [55] links the callback methods with their registration methods to facilitate the static analysis tools in gaining more precise results. DroidSafe [3] implements a simplified model of the Android system and solves native code in the Android framework by manually analyzing the source code and writing stubs for them in Java. HornDroid [2] generates Horn clauses from the bytecode of application and performs both value-sensitive and flow-sensitive analysis on the clauses. HSOMiner [56] uses machine learning algorithms to discover the hidden sensitive operations by analyzing the branch instructions and their related conditional branches.

B. Dynamic Analysis Tools

DroidScope [20] provides an instrumentation tool to monitor the executed bytecode and native instructions to help analysts learn the malware manually. VetDroid [21] executes the Android applications by a custom application driver and performs a permission usage behavior analysis. CopperDroid [19] traces the system calls and reconstructs the behavior of the target application. TaintDroid [17] and TaintART [18] are taint flow analysis system on different Android Java virtual machines. They track the information flow of the target application at runtime and report the data leakage from sink methods. DexHunter [14] focuses on how to dump the whole DEX file from memory at a “right timing”. AppSpear [13] leverages the key data structures in Dalvik to reassemble the DEX file and claims that these data structures are reliable. Both DexHunter and AppSpear assume that there is a clear boundary between the unpacking code and the original code. However, the unpacking code and malicious code may intersperse with each other. Moreover, advanced malware can modify bytecode and data in the DEX file at runtime, and thus the previous dump-based unpacking systems will miss the content modified after the dump procedure.

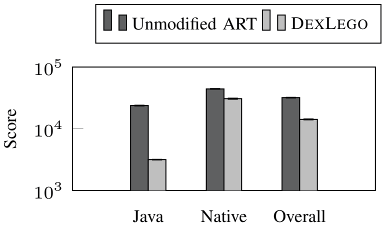

Fig. 6: Performance Measured by CF-Bench [54].

TABLE VIII: Time Consumption of DexLego. The column ”Original” represents the mean and standard deviation (STD) of the launch time with unmodified ART, while the last column represents launch time with DexLego.

Application	Version	Original		With DexLego	
		Mean	STD	Mean	STD
Snapchat	9.43.0.0	826.9ms	52.11ms	1,664.7ms	16.08ms
Instagram	9.7.0	608.5ms	45.6ms	1,275.8ms	25.37ms
WhatsApp	2.16.310	236.4ms	12.24ms	480.2ms	84.3ms

C. Hybrid Analysis Tools

Harvester [23] collects runtime values and injects these values into the DEX file for the accuracy improvement of analysis tools. However, some limitations still exist. Firstly, marking logging points and backward slicing are based on the original DEX file. If packing is considered, Harvester loses its target like other static analysis tools. In contrast, DexLego does not analyze the original DEX file. Additionally, Harvester greatly facilitates static analysis tools on solving reflections as they reduce the parameters back into constant strings. However, malware can use advanced reflection code to evade the analysis. Since DexLego replaces the reflective call with direct call, we do not care about how the adversaries use reflection.

D. Unpacking and Reassembling in Traditional Platforms

Ugarte et al. [57] present a summary of recent unpacking tools and develop an analysis framework for measuring the complexity of a large variety of packers. CoDisasm [58] is a dissembler tool that takes memory snapshot during execution and disassembles the captured memory. Uroboros [59] aims to disassemble binaries with a reassembleable approach. Their reassembling method is based on the disassembling output of Uroboros. DexLego is different from these systems as we do not disassemble the binary or monitor memory. [60] collects the instruction trace at runtime and performs taint analysis on the trace. Unlike [60], DexLego aims to facilitate the other static analysis tools and outputs a standardized DEX file, which could be used for state-of-the-art static analysis tools to perform different kinds of analysis including taint analysis.

VII. Limitations and Future Work

Although the bytecode collection in DexLego is not based on the machine code in ART, the experience of TaintART shows that we can also implement our collecting algorithm in the compliers of ART [18] to achieve the same goal. As we implement DexLego in a real mobile device, we consider that it is transparent to applications with anti-emulation techniques. However, advanced malware may be aware of its existence by code footprints or checksum values of Android libraries. One potential solution is to leverage hardware isolated execution environment mentioned in [61] to reduce the artifacts of the system and improve the transparency. The code coverage improvement modules in DexLego may introduce additional false positives on the unreachable code paths caused by unrealistic input. It is a trade-off between the code coverage and the analysis precision. As DexLego collects instructions in ART, our procedure may also be compromised by native code. To prevent attackers tampering DexLego, we can randomize the memory address of DexLego [62], [63] to make it difficult to be located. Additionally, using sandbox [64], [65] or hardware-assisted isolated execution environments such as TrustZone technology [61], [66]–[68] can secure the execution of DexLego. Note that applying these techniques to the entire ART may introduce a heavy performance overhead or compatibility issues, and we need to restrictively use them on DexLego only. Currently, DexLego only reveals the behavior performed by Java code. However, JNI technique allows sophisticated malware to perform malicious behavior through native code. We consider tracing the native instructions and reassemble them as our future work.

VIII. Conclusions

In this paper, we present DexLego, a novel system that performs bytecode extraction and reassembling for aiding static analysis. It adopts instruction-level JIT collection to record the data and control flows of applications, and reassembles the extracted information back into a new DEX file. The evaluation results on packed DroidBench samples and real-world applications with state-of-the-art static analysis tools show that DexLego correctly reveals the behavior in packed applications even with self-modifying code. The F-Measures of FlowDroid, DroidSafe, and HornDroid increase by 33.3%, 31.1%, and 23.6%, respectively. We also show that DexLego provides a better accuracy than pure dynamic analysis, and our force execution module efficiently increases the code coverage of the dynamic analysis.

IX. Acknowledgement

This work is supported by the National Science Foundation Grant No. CICI-1738929 and IIS-1724227. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government.

References

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, 2014.

[2] S. Calzavara, I. Grishchenko, and M. Maffei, "HornDroid: Practical and sound static analysis of Android applications by SMT solving," in *Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P'16)*, 2016.

[3] 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 *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[4] 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 *Proceedings of the 37th International Conference on Software Engineering-Volume 1 (ICSE'15)*, 2015.

[5] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*, 2014.

[6] AVL Team, "AVL malware report 2015," https://www.avlsec.com/, 2016.

[7] Alibaba Inc., "AliProtector," http://jaq.alibaba.com/, 2014.

[8] Baidu Inc., "BaiduProtector," http://app.baidu.com/jiagu/, 2014.

[9] Ijiami Inc., "IJiamiProtector," http://www.ijiami.cn/AppProtect, 2013.

[10] Licel Inc., "DexProtector," https://dexprotector.com/, 2013.

[11] Qihoo 360 Inc., "360Protector," http://jiagu.360.cn/protection, 2014.

[12] Tencent Inc., "TencentProtector," http://legu.qcloud.com/, 2014.

[13] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu, "AppSpear: Bytecode decrypting and DEX reassembling for packed Android malware," in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'15)*, 2015.

[14] Y. Zhang, X. Luo, and H. Yin, "DexHunter: Toward extracting hidden code from packed Android applications," in *Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS'15).*, 2015.

[15] Bluebox Security Inc., "Android security analysis challenge: Tampering Dalvik bytecode during runtime," https://bluebox.com/android-security-analysis-challenge-tampering-dalvik-bytecode-during-runtime/, 2013.

[16] J. hyuk Jung and J. Lee, "DABID: The powerful interactive Android debugger for Android malware analysis," Asia Black Hat, 2015.

[17] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, 2010.

[18] M. Sun, T. Wei, and J. Lui, "TaintART: a practical multi-level information-flow tracking system for Android RunTime," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*, 2016.

[19] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of Android malware behaviors," in *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[20] L. K. Yan and H. Yin, "Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21st USENIX Security Symposium (USENIX Security'12)*, 2012.

[21] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in Android apps with permission use analysis," in *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*, 2013.

[22] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst, "Static analysis of implicit control flow: Resolving Java reflection and Android intents," in *Proceedings of the 30th Annual International Conference on Automated Software Engineering (ASE'15)*, 2015.

[23] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in Android applications that feature anti-analysis techniques," in *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[24] EC SPRIDE Secure Software Engineering Group, "DroidBench," https://github.com/secure-software-engineering/DroidBench, 2013.

[25] 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 *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[26] F-Droid, "F-Droid," https://f-droid.org/, 2011.

[27] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for Android applications," in *Proceedings of the 25th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'16)*, 2016.

[28] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*, 2012.

[29] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *Proceedings of the 19th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'13)*, 2013.

[30] Google Inc., "UI/Application Exerciser Monkey," https://developer.android.com/studio/test/monkey.html, 2008.

[31] 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 (MobiSys'14)*, 2014.

[32] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC'13/FSE'13)*, 2013.

[33] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'12)*, 2012.

[34] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[35] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing Android apps through symbolic execution," *ACM SIGSOFT Software Engineering Notes*, 2012.

[36] M. Y. Wong and D. Lie, "IntelliDroid: A targeted input generator for the dynamic analysis of Android malware," in *Proceedings of the 23nd Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[37] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: analyzing sensitive data transmission in Android for privacy leakage detection," in *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*, 2013.

[38] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu, "iRiS: Vetting private api abuse in iOS applications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[39] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu, "J-Force: Forced Execution on JavaScript," in *Proceedings of the 26th International Conference on World Wide Web (WWW'17)*, 2017.

[40] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-Force: Force-executing binary programs for security applications," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*, 2014.

[41] Google Inc., "Android open source project," https://source.android.com/, 2008.

[42] Team Win, "Team win recovery project," https://twrp.me/, 2014.

[43] Google Inc., "Dalvik executable format," https://source.android.com/devices/tech/dalvik/dex-format.html, 2008.

[44] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for Java program analysis: A retrospective," in *Proceedings of the Cetus Users and Compiler Infastructure Workshop (CETUS'11)*, 2011.

[45] Bangcle Ltd., "BangcleProtector," https://www.bangcle.com/, 2013.

[46] NQ Mobile, "NetQinProtector," https://shield.nq.com, 2014.

[47] Android APK Encryption and Protection, "APKProtector," https://sourceforge.net/projects/apkprotect/, 2013.

[48] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of Google Play," in *Proceedings of the ACM SIGMETRICS*, 2014.

[49] Google, "Google Play," https://play.google.com/store?hl=en, 2017.

[50] Qihoo 360 Inc., "360 Market," http://zhushou.360.cn/, 2017.

[51] Wandoujia, "Wandoujia Market," http://www.wandoujia.com/apps, 2017.

[52] F-Droid, "F-Droid Random Page," https://f-droid.org/wiki/page/Special:Random, 2011.

[53] JaCoCo, "Java Code Coverage Library," http://www.eclemma.org/jacoco/, 2009.

[54] Chainfire, "CF-Bench," https://play.google.com/store/apps/details?id=eu.chainfire.cfbench, 2013.

[55] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework," in *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[56] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, "Dark Hazard: Learning-based, large-scale discovery of hidden sensitive operations in Android apps," in *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS'17)*, 2017.

[57] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "SoK: Deep packer inspection: A longitudinal study of the complexity of runtime packers," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[58] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "CoDisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[59] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*, 2015.

[60] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[61] Z. Ning and F. Zhang, "Ninja: Towards transparent tracing and debugging on arm," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ning

[62] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee, "From zygote to morula: Fortifying weakened ASLR on Android," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P'14)*, 2014.

[63] M. Sun, J. C. Lui, and Y. Zhou, "Blender: Self-randomizing address space layout for Android apps," in *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'16)*, 2016.

[64] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna, "Going Native: Using a large-scale analysis of Android apps to create a practical native-code sandboxing policy," in *Proceedings of the 23nd Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[65] M. Sun and G. Tan, "NativeGuard: Protecting android applications from third-party native libraries," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks (WiSec'14)*, 2014.

[66] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "TrustShadow: Secure execution of unmodified applications with ARM trustzone," in *Proceedings of the 15th Annual International Conference on Mobile systems, applications, and services (MobiSys'17)*, 2017.

[67] F. Zhang and H. Zhang, "SoK: A study of using hardware-assisted isolated execution environments for security," in *Proceedings of Hardware and Architectural Support for Security and Privacy (HASP'16)*, 2016.

[68] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "CaSE: Cache-Assisted Secure Execution on ARM Processors," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P'16)*, 2016.