Action-Based Test Carving for Android Apps

Anonymous Author(s)

***Abstract*—Effective software testing of Android apps should combine different types of tests, including end-to-end tests to validate user flows and unit tests that provide focused executions. However, the tight coupling between the Android framework and the apps makes unit testing challenging: unit tests are executed on the device against the actual Android framework, which is cumbersome and time-consuming, or they cannot access its full implementation, which might prevent correctly testing the units' behavior. To address this issue, we propose a novel technique called ARTISAN that (i) traces the app execution during end-to-end testing on Android devices, (ii) identifies *focal* methods to test, (iii) carves the necessary preconditions for testing those methods from the collected traces, and (iv) synthesizes executable unit tests that can run *outside* Android devices. As our evaluation involving open-source Android apps with existing end-to-end tests shows, ARTISAN can quickly generate unit tests that cover a significant portion (i.e., 45% on average) of the code exercised by the end-to-end tests.**

I. Introduction and Motivation

Testing is an essential aspect of the software development life cycle and is crucial in improving software quality. For Android applications (or apps in short), testing is critical to avoid failures that can lead to the disruption of mission-critical activities, loss of reputation, and customer loss.

A good testing strategy needs to find an appropriate balance between the fidelity of the tests, testing speed, and testing reliability [1]. To achieve this balance, app developers can create tests at different granularity levels [1]. *End-to-end tests* exercise large parts of an app throughout its User Interface or Graphical User Interface (GUI) and help developers check user flows. *Unit tests* focus on a small portion of an app and help developers debug and test regression. Finally, *medium tests* check the integration of units. In the Android realm, developers also need to decide on which platform tests shall run [1]. *Instrumented tests* execute on an Android device, either physical or emulated, whereas *Local tests*, instead, execute on a Java virtual machine (JVM).

Although it is possible to create tests by combining granularity levels and execution environments (e.g., unit tests that run in the Android device), related work on app testing [2], [3] observed that instrumented end-to-end GUI tests and local unit tests are the most frequently used.

Researchers and practitioners proposed several automatic and semi-automatic techniques to help app developers create end-to-end instrumented tests (e.g., [4]–[16]). However, only a few existing techniques [17], [18] for automatically creating local unit tests have been proposed even though developers would benefit from having those tests which run fast and simplify debugging activities [2]. Existing techniques have limited applicability as they require substantial manual effort [17] and are not based on the tester's intent [18] (i.e., they are based on an input generation strategy). Additionally, they cannot handle the technical challenges of testing typical Android apps that (i) operate through event-based and inversion-of-control paradigms (i.e., everything, including the instantiation of components required as a precondition by the unit tests, is orchestrated by the Android framework), (ii) are executed in a different environment than unit tests, and (iii) require replacing Android components with test doubles [3], [19], [20] using libraries such as Robolectric [21] that are not comprehensive [22].

Test carving [23] has been proposed for generating unit tests from end-to-end tests by either checkpointing portions of an application state and reloading it during unit testing to set the test preconditions (i.e., state-based carving) or by extracting units of execution from end-to-end executions and replaying them as part of the generated unit tests (i.e., action-based carving). The potential benefits of test carving include improving regression testing effectiveness by enabling standard regression test selection techniques, reducing the sensitivity to interference bugs by better isolating tests, and enabling developers to perform more focused debugging activities. Consequently, various techniques for performing test carving have been proposed (e.g., [23]–[32]). However, to the best of our knowledge, no technique allows carving unit tests across different platforms while addressing the challenges of locally testing Android apps.

To address those challenges, we propose ARTISAN, an *action-based* test carving technique that takes as input the app under test (AUT) and its end-to-end GUI tests and automatically produces a set of locally executable unit tests as output. In a nutshell, ARTISAN (1) instruments the AUT to enable tracing of method invocations at runtime; (2) it executes the end-to-end GUI tests against the instrumented app on an Android (possibly emulated) device; (3) it identifies and carves the relevant method invocations for testing a unit from the collected traces; and, (4) it synthesizes locally executable unit tests that feature suitable test doubles.

We evaluated ARTISAN using five apps and 152 end-to-end GUI tests. Since this work is the first step towards effectively unit testing Android apps, we focus our investigation on how ARTISAN can quickly generate carved executions from GUI tests and how different strategies to identify focal methods compare to each other. Additionally, we report on some of the generated tests' characteristics (e.g., length, number of stubs) to foster future studies involving Android app developers and further research on improving unit test qualities (e.g., readability, maintainability) and test oracles generation. Using the best-performing carving strategy, in 43 minutes (on average) ARTISAN generated $2,087$ local unit tests that cover

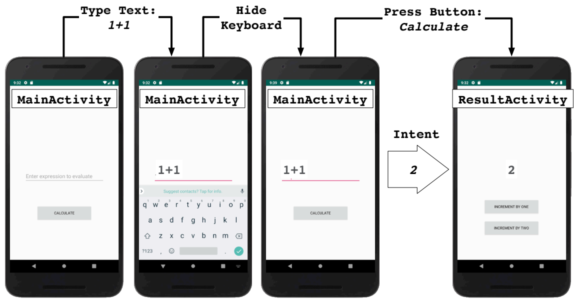

Fig. 1: Main user flow of BASICCALCULATOR.

```java
@RunWith(AndroidJUnit4.class)
@LargeTest
public class MainActivityTest {

  @Test
  public void testCalculate() {
    onView(withId(R.id.input)).perform(typeText("1+1"));
    Espresso.closeSoftKeyboard();
    onView(withId(R.id.calculateButton)).perform(click());
    onView(withId(R.id.resultView)).check(matches(withText(
     "2")));
  }
}
```

Listing 1: GUI test stressing BASICCALCULATOR main user flow.

45% (on average) of statements exercised by the original end-to-end GUI tests. Our technique did not obtain 100% of the original end-to-end tests' coverage as ARTISAN (i) focuses on executions originated in the apps' main thread (the user-facing thread [33]) from Android activities (a key user-facing Android component [34]) and (ii) does not refactor the original source code of the apps (which would be required to handle executions originating from callbacks defined in anonymous classes).

In summary, this paper makes the following contributions:

- An automated technique that performs action-based test carving for Android apps.
- An publicly available implementation of the technique (see replication package [35].
- An empirical evaluation that provides initial evidence of the effectiveness and efficiency of our technique.

Based on the current achieved results, we believe that ARTISAN can provide local unit tests with significant coverage to developers and help them better test Android apps.

II. BACKGROUND AND RUNNING EXAMPLE

To introduce background concepts and present our technique, we created an illustrative Android app called BASICCALCULATOR. The app solves user-provided mathematical expressions.[1]

Android apps are composed of `Activities`, software components whose life cycle (e.g., creation, destruction) and communications are handled by the Android framework. For example, BASICCALCULATOR consists of two activities: `MainActivity`, which initializes the app, parses input expressions and solves them, and `ResultActivity`, which shows the results to the users in a new screen (see Figure 1).

Activities are loosely-coupled and communicate via message passing. Messages are called `Intents` and can contain arbitrary, serializable payload.

Activities are also event-based: their logic is encapsulated into *listeners* and *callbacks*, and the dispatch of predefined events by the Android framework triggers their execution. Typical examples of such event listeners are life-cycle events (e.g., *create*, *destroy*) and events generated by GUI elements (e.g., *buttonPressed*). For instance, `MainActivity` sets up the app GUI, i.e., the text input field and the "Calculate" button shown in Figure 1, and registers the various callbacks to promptly handle user interactions (e.g., the pressing of the "Calculate" button) when the *create* event is triggered. Its `sendResult` method, instead, is invoked when the user presses the "Calculate" button, fetches the content of the text input field, and invokes the `eval` methods to compute the results that, finally, `ResultActivity` shows. Notably, GUI elements extend `android.view.View` and can be retrieved by invoking the method `findViewById` using the unique IDs defined by Android. Starting another activity, instead, requires calling `startActivity` and passing an intent that contains a reference to the activity to start (e.g., `ResultActivity`) and an optional payload (e.g., `eval`'s result).

Testing BASICCALCULATOR main user flow can be done using tests similar to the GUI test in Listing 1; doing so requires developers to (i) build the app, (ii) install it inside an Android device or emulator, and (iii) execute the test interacting with the app's GUI *on* that device.

An alternative for testing BASICCALCULATOR is to use local unit tests that are focalized on specific units of code and can be run on any standard JVM. Listing 2 reports a local unit test generated by ARTISAN for checking the behavior of `MainActivity`'s `sendResult` under the same conditions observed while running the GUI test in Listing 1. From this code snippet, one can note that `test_MainActivity_sendResult_001` implements a complex setup to ensure that objects such as "Calculate" button and the text input field, which are normally provided by the Android framework, are also available during unit testing.

Since those Android-managed objects cannot exist outside Android devices, ARTISAN replaces them with stubs and mocks. Specifically, ARTISAN relies on state-of-practice libraries such as Mockito [36] and Robolectric [21] that provide basic stubbing capabilities and the ability to (partially) simulate the Android framework on the JVM. In our example, ARTISAN created a simple mock of the "Calculate" button (Lines 8–11), nested mocks that simulate accessing the text input field (Lines 18–25), and "injected' these mocks into Robolectric to enable their execution (Lines 29–30).

In summary, this motivating example shows how complex Android apps' unit tests can be and illustrates some of the technical challenges involved in generating them.

[1]Although we did not consider BASICCALCULATOR in our empirical evaluation, we included it in the replication package.

```java
@RunWith(RobolectricTestRunner.class)
@Config(shadows = { EditTextShadow28.class })
public class Test028 {

  @Test(timeout = 4000)
  public void test_MainActivity_sendResult_001() {
    // Mock the Calculate button
    Button button2 = Mockito.mock(Button.class);
    Stubber stubber3 = Mockito.doReturn(R.id.
     calculateButton);
    Button button3 = stubber3.when(button2);
    button3.getId();
    // Instantiate MainActivity using Robolectric
    ActivityController controller  = Robolectric.
     buildActivity(MainActivity.class);
    MainActivity mainactivity = controller.get();
    // Simulate triggering "create" event
    controller.create();
    // Mock the text input field
    SpannableStringBuilder stringbuilder3 = Mockito.mock(
     SpannableStringBuilder.class);
    Stubber stubber4 = Mockito.doReturn("1+1");
    SpannableStringBuilder stringbuilder4 = stubber4.when(
     spannablestringbuilder3);
    spannablestringbuilder4.toString();
    EditText edittext3 = Mockito.mock(EditText.class);
    Stubber stubber5 = Mockito.doReturn(
     spannablestringbuilder3);
    EditText edittext4 = stubber5.when(edittext3);
    edittext4.getText();
    // Inject the mocks in Robolectric
    EditText edittext2 = mainactivity.findViewById(R.id.
     input);
    EditTextShadow28 edittextshadow = Shadow.extract(
     edittext2);
    edittextshadow.setMockFor("android.widget.EditText:
     android.text.Editable getText()", edittext3);
    edittextshadow281.setStrictShadow();
    // Invoke the Method Under Test
    mainactivity.sendResult(button3);
  }
}
```

Listing 2: A unit test carved by ARTISAN for BASICCALCULATOR

III. TECHNIQUE

ARTISAN is an end-to-end approach composed of several steps (Figure 2). It starts by instrumenting a non-obfuscated *Original App* written in Java to enable tracing method invocations. Next, it executes the *Instrumented App* on an Android device against *GUI Tests* to collect *Execution Traces*. Then, it parses the traces into a form amenable to automatic analysis (i.e., graphs) and carves the original executions. Finally, it augments the *Carved Executions* with code that mocks dependencies provided by third-party libraries and the Android framework and synthesizes the source code of the *Carved Unit Tests* from the *Extended Carved Executions*.

Notably, ARTISAN generates carved unit tests in the static single-assignment (SSA) form [37], an intermediate program representation widely used in compilers in which each variable has only one definition site. Although programs written in SSA form are generally longer than programs written in other forms, which might affect their readability, we decided to generate test code in SSA form for two main reasons: on the one hand, it is easy to translate the carved executions in this form; on the other hand, SSA enables the application of

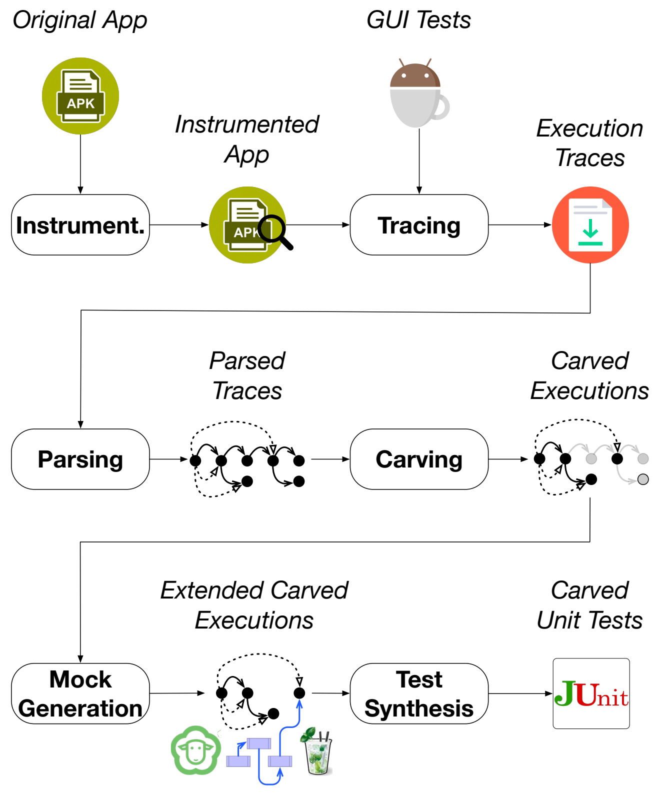

Fig. 2: An overview of ARTISAN's end-to-end approach to carving local unit tests from instrumented GUI tests.

standard program analyses (e.g., live variable analysis) and optimization techniques (e.g., test minimization [38]).

A. Instrumentation and Tracing

Action-based test carving is a dynamic analysis technique that requires execution traces to identify the units of execution, i.e., method invocations, that are relevant for testing code units. Therefore, ARTISAN instruments the original AUT using the byte-code modification library Soot [39].

ARTISAN adopts a light-weighted approach to trace Android apps and generates traces in plain text, making it possible for developers to easily inspect them. Specifically, ARTISAN injects code that logs for each method invocation, the method signature, the actual parameters, and any returned values or thrown exceptions.

Tracing focuses only on AUT's operations, therefore, it instruments the method bodies of all the methods that belong to the AUT, whereas it treats third-party libraries, standard language libraries, and the Android runtime as black boxes. Consequently, ARTISAN can only trace the invocations of methods that belong to those operations that are made by the AUT. Notably, ARTISAN differentiates between calls to instance and static methods, traces method calls at different visibility levels and distinguishes whether methods return normally or exceptionally. In the latter case, the trace contains also the indication of whether the exceptional behavior was caused by a checked or an unchecked exception.

Differently than state-based carving, ARTISAN does not check-point the application state nor serialize the objects used as parameters or return values. Instead, it manages all the object instances, including exceptions, *by-reference* and stores in the trace only their *object id*. ARTISAN obtains the object id of non-null instances by calling `System.getObjectIdentity()` to capture the actual object types along with their hash codes. To avoid cluttering the execution traces, ARTISAN manages primitive types, boxed primitive types, and "stringly" types[2] *by-value* and reports only their string representation.

Action-based carving implicitly assumes that all interactions between objects happen exclusively via method invocations; unfortunately, units of executions like array stores, array accesses, and field assignments are not implemented as method calls in Java and would be missed if not properly handled. To avoid missing such fundamental units of execution, ARTISAN implement a custom instrumentation code that traces them as *synthetic* methods. For example, it traces an array store like `array[0] = 10` as the generic method invocation `abc.ArrayOperation.set(array, 0, 10)`.

As discussed in Section II, activities communicate by passing `Intent` objects that the Android framework serializes in a stream of bytes and deserializes into actual objects. Thus, the object ids of serialized and deserialized objects differ, which effectively breaks the (logical) connection between them. To avoid losing this connection, ARTISAN leverages *application level taint tracking* [40], [41]. Instead of modifying the `Intent`'s bytecode to accommodate the tainting value, i.e., the object id of the intents to be sent, ARTISAN stores the tainting values directly in the intents as a regular payload using a special key before Android sends them. To read the tainting value, instead, ARTISAN injects custom code that is invoked before the receiving activity accesses the payload. This custom code extracts from the payload the tainted value by invoking a standard `Intent` method using the special key as a parameter. Doing so enables ARTISAN to trace that method invocation, effectively exposing the logical connection between sent and received intents.

B. Trace Parsing

After tracing the execution of the GUI tests, ARTISAN parses the generated execution traces into graph data structures that capture various types of dependencies between method invocations and object instances. Specifically, ARTISAN captures chronological, data, and (method) call dependencies. Chronological dependencies capture the order in which method invocations have been traced; hence, they are useful to identify (past) method invocations that may be useful to set test preconditions. Data dependencies, instead, identify method invocations that act on the same objects; hence, they are useful to identify test preconditions. Finally, call dependencies capture the (nesting) relations among the method invocations; hence, they are useful to ensure that carved method invocations are executed the right amount of times (e.g., no duplicate executions). ARTISAN parses each execution trace file into the following three graphs:

Execution Flow Graph. This is a (doubly) linked list whose nodes represent method invocations and edges the (strict) precedence/follow relations (i.e., IS_BEFORE and IS_AFTER).

Data Dependency Graph. This is a directed graph that links method invocations to data nodes and data nodes to method invocations. Data nodes can be either object instances or primitive values. Object instances can be linked to multiple method invocations, whereas primitive values are always linked to one and only one method invocation. In this graph, two nodes are linked when (i) an object instance OWNS a method invocation; (ii) a data node is used as a PARAMETER of a method invocation; (iii) a STATIC data node is used inside a method invocation; (iv-a) a method invocation RETURNS a data node or (iv-b) THROWS an exception.

Call Dependency Graph. This is a directed, acyclic, disconnected graph whose nodes represent method invocations and edges the INVOKE relation. Notably, this graph is a forest because Android apps can have multiple entry points.

After parsing is completed, ARTISAN *decorates* the graphs by including additional information that will be used later during carving and test synthesis. This step includes (i) identifying method invocation nodes that are owned by Android components, like Activities, and tagging the nodes corresponding life cycle events callbacks (e.g., `onCreate`); (ii) aliasing data nodes that correspond to sources and sinks of tainted intents and their payload; and, (iii) injecting static dependencies.

C. Action-based Carving

The carving process begins after parsing the execution traces and considers only one trace at a time. At first, ARTISAN selects "carvable" targets, i.e., the method invocations for which developers want to generate unit tests. In general, there are no restrictions on which method invocations can be carved; however, carving some method invocations, such as private methods and methods that do not belong to the AUT might produce non-compilable or irrelevant unit tests. Therefore, ARTISAN automatically filters out invocations of private methods and invocations whose owner type (e.g., `basiccalculator.MainActivity`) does not match the AUT's package name (e.g., `basiccalculator`). Among the remaining method invocations, ARTISAN selects the ones that match user-defined criteria on the methods' signature, actual arguments, and return values. In case an end-to-end test makes multiple calls to the same target method ARTISAN provides a special option to either select one (i.e., the first) or all the invocations of that method as "carvable" targets (i.e., ARTISAN can use different carving strategies).

After selecting the target method invocations, ARTISAN finds all the method invocations that are relevant to them,

[2]Strings and CharSequences are an example of types that ARTISAN treats as primitives.

either directly or indirectly, using a backward slicing algorithm: Starting from a target method invocation, this algorithm identifies the past method invocations that match one of the following three conditions: (1) the method invocation shares the same owner with the target method invocation; (2) the method invocation is owned by a parameter used by the target method invocation; or, (3) the method invocation belongs to a static class used within the target method body.

Since the selected method invocations might introduce additional dependencies (e.g., parameters to be set), this algorithm iteratively carves each of them. Nevertheless, the algorithm converges because, at every iteration, it considers a smaller set of dependencies. Moreover, as the "carvable" targets are considered sequentially and might share dependencies, ARTISAN caches intermediate results and speeds up the algorithm.

Once the algorithm selects all the method invocations relevant to a "carvable" target, ARTISAN uses the call dependency graph to retrieve all the additional method invocations that would be executed because the relevant method invocations are. For instance, in our working example, invoking `sendResult` with a valid mathematical expression in a unit test would also cause it to automatically execute `eval`. Given this "extended" set of method invocations, ARTISAN creates carved executions by extrapolating the connected components they form in the execution flow graph, the data dependency graph, and the call dependency graph.

As the last step, ARTISAN "cleans up" the carved executions by removing those units of computation, such as Lambdas, which naturally occur in Android apps but cannot be directly instantiated in the unit tests, and re-carves the target method invocations within the carved trace executions to ensure they remain consistent.

D. Mock Generation

Carved executions contain a list of executed method invocations and their data dependencies; hence, they are not yet *executable* code. Additionally, they might lack the method invocations necessary to instantiate objects that are managed by Android or instantiated by third-party libraries. In this case, the carved execution contains uses of those objects, but not their definitions. We label those objects that are used but never instantiated in the scope of the carved executions as *dangling*.

ARTISAN deals with dangling objects by means of *mocking*, a standard technique that improves the reliability of unit tests by replacing complex dependencies, i.e., other objects, with pre-programmed test doubles. Specifically, ARTISAN tracks how each dangling object is used within the carved executions and automatically programs a mock object that can replicate the (observed) behavior within the unit tests. If a stubbed method invocation on a mocked dangling object returns another dangling object, then ARTISAN reproduces its behavior utilizing another automatically configured mock object; the process continues until there are no more dangling objects left. Thus, synthesizing mocks is an iterative activity based on forward slicing (i.e., the opposite of carving). Since the number of data dependencies to consider is finite and shrinks at every iteration to the algorithm, also synthesizing mocks is guaranteed to always terminate.

Having programmed the mock objects is not enough because Robolectric, which simulates the Android framework on the JVM by wrapping Android's classes using *shadow* objects that can modify or extend their behavior [21], is not aware of them. Hence, it will not use them during the execution of the unit tests. Therefore, ARTISAN extends Robolectric by automatically synthesizing new shadow objects that take the just-programmed mocks as input and use them to replicate the expected behavior of the Android-managed object.

E. Synthesis of Carved Unit Tests

At this point, all the test preconditions are either instantiated or mocked, and ARTISAN can finally synthesize the code implementing the unit tests by transforming the (extended) carved execution's call dependency graph root-level nodes, i.e., the directly "visible" method invocations, into their corresponding source-code method invocations. While doing so, ARTISAN relies on the data dependency graph to generate all the variables needed to host the references or values that correspond to methods' owners, parameters, and return values. Notably, the (extended) carved executions do not contain complex control flows; thus, the generated tests consists of a number of variable declarations and a sequence of method invocations, as one would expect from unit tests.

To generate the mocking code which stubs the methods contained in the extended carved executions, ARTISAN uses a predefined template: For each dangling object `do`, (1) it declares `do` as a mock object using Mockito (Line 8); (2) it specifies which object (if any) the mock should return (Line 9) and (3) after its stubbed method `sm` is invoked(Line 10); finally, (4) it invokes the stubbed method `sm` on `do` to activate the mocking (Line 11). To inject the pre-programmed mock objects inside the Android GUI elements simulated by Robolectric, instead, ARTISAN (1) retrieves the GUI elements from the activity using their (known) unique ID by invoking the method `findViewById` (Line 27); (2) it extracts the shadow objects simulating them (Line 28), and (3) passes the mock objects to the shadows (Line 29). Since Robolectric and Mockito lack a proper integration, ARTISAN synthesizes custom shadows that act as a bridge between the two libraries dynamically. In turn, these shadows allow the mock objects to be executed inside Robolectrics' Android simulations.

IV. Limitations

In its current implementation, ARTISAN can synthesize executable unit tests that lack test oracles, do not involve any Android components besides (simulated) activities, intents, and GUI elements, and consider only traces generated by the main Android thread. We argue that automatically generating test oracles, which is currently an open research problem, is outside the scope of this first work on carving unit tests for Android apps, and it could be partially addressed by existing approaches based on mutation analysis [42] or Deep Learning

TABLE I: Benchmarks used in the empirical evaluation.

ID	*Name*	*Category*	*Version*	*LOC (K)*	*GUI Tests*
A1	BLABBERTABBER	Tools	1.0.10	2.4	9
A2	FIFTHELEMENT	Music	2.2.5	69.8	17
A3	OWL FLASH CARDS	Education	1.1	6.4	12
A4	PRISMACALLBLOCKER	Tools	1.2.3	12.0	67
A5	UK-GM	Education	1.2.1	5.3	47

Language models [43], [44]. Extending ARTISAN's implementation to handle more Android components and multiple threads, instead, is mostly an engineering effort.

Regarding our evaluation, we did not involve developers to evaluate the quality of carved tests. We plan to perform such an evaluation in future work. Specifically, we plan to perform studies with developers to understand which carved tests best help developers and explore alternative carving strategies based on the results of the studies.

V. EMPIRICAL EVALUATION

We evaluated ARTISAN's effectiveness and efficiency empirically on a set of open-source, well tested Android apps. Specifically, we targeted the following research questions (RQs):

RQ1: Can ARTISAN carve unit tests from GUI tests?
RQ2: What is the cost of running ARTISAN?
RQ3: What are the characteristics of carved tests?
RQ4: How do different carving strategies compare?

A. Experimental Benchmarks

For the empirical evaluation, we used a set of five open-source Android apps (Table I). We used open-source apps because through their public repositories they make available both their source code, required by ARTISAN, and the existing GUI tests, required by our evaluation. To identify relevant apps, we used a dataset of $1,002$ apps with tests from related work [2]. The apps in the dataset are publicly available on GitHub [45] and, to the best of our knowledge, the dataset was the largest set of apps with tests at the time we started evaluating ARTISAN.

We selected the five apps from the dataset as follows: First, we identified apps that contain GUI tests written in Espresso; this step identified 245 relevant apps. We identified GUI tests written in Espresso by analyzing whether the abstract syntax trees (ASTs) of the test files contained invocations of the Espresso API. Second, we filtered out apps that use programming languages other than Java (e.g., Kotlin), as ARTISAN does not currently support them; this step left us with 180 apps. Third, we sorted the 180 apps in descending order based on the number of GUI tests associated with the apps and processed the list of apps starting from the one having the highest number of tests. We discarded any app that we could not build and for which the available GUI tests did not pass or showed flakiness [46]. We stopped as soon as we identified five apps. We could not build or run some of the top-ranked apps as those had outdated dependencies, required an API key for a third-party service used by the apps, or interacted with Web servers not reachable anymore. We determined whether all tests were passing and were not flaky by checking whether all tests passed in ten runs of the tests.

Table I summarizes the main elements of the apps we considered in the evaluation. For each app, the table reports an identifier for the app (*ID*), its name (*Name*), the category of the app (*Category*), its version (*Version*), the number of source and test code lines (in thousands) (*LOC (K)*), and the number of existing GUI tests (*GUI*).

B. Experimental Settings

To answer the RQs, we ran ARTISAN on the five apps considered on a dedicated workstation with 128GB of memory, an Intel i9-9900K 3.60GHz processor, and running Ubuntu 18.04. To execute the GUI tests, we used an Android Nexus 5X emulator running API 28. We used API 28 as it was compatible with all the selected apps according to their supported Android API versions [47].

In the RQs, we evaluated ARTISAN using different *carving strategies*, i.e., different approaches for selecting the target method invocations to carve from the traces generated by GUI tests' execution. RQ1, RQ2, and RQ3 are based on a strategy (**Strategy-I**) that, for each trace, selects as carving targets only the first occurrence of method invocations with the same fully qualified method signature. The idea behind using this strategy is that it allows for covering the behavior of multiple methods while limiting the cost of running the technique. This strategy might carve the same method multiple times across different traces. Still, we think that this is reasonable as the traces originate from different GUI tests that likely have different objectives. RQ4, instead, compares three carving strategies. The first strategy (**Strategy-I**) is the same as the one used to answer RQ1, RQ2, and RQ3. The second strategy (**Strategy-II**) selects as carving targets all the method invocations of non-private methods defined in the AUT. In this strategy, we use a timeout of 10 hours for carving tests to evaluate the case in which the technique runs overnight. The third strategy (**Strategy-III**) selects carving targets as the second one but changes the timeout to 5 hours. We decided to use this strategy to investigate the effectiveness of ARTISAN in a context where the results of the technique can be used on the same day as the technique is started (i.e., the results can be used for the development activities of the day).

C. Results

1) RQ1: Can ARTISAN *carve tests from GUI tests?:* Table II reports the results of running ARTISAN on the benchmark apps using **Strategy-I**. For each app, the table reports the identifier of the app (*ID*), the number of traces collected from running the GUI tests associated with the app (*Traces*), the number of method invocations in the traces (*Method Invocations*), the number of traces that ARTISAN could parse while carving tests (*Parsed Traces*), the number of method invocations selected by the carving strategy (*Targets*), the number of tests carved (*Carved Tests*), and the statement and branch coverage achieved by GUI and carved tests (*GUI*

TABLE II: Results of running ARTISAN on the benchmarks using Strategy-I.

ID	*Traces*	*Method Invocations*	*Parsed Traces*	*Targets*	*Carved Tests*	*Statement Coverage (%)*			*Branch Coverage (%)*		
						GUI Tests	*Carved Tests*	*Included*	*GUI Tests*	*Carved Tests*	*Included*
A1	9	1,654	9	55	48	45	12	100	35	11	100
A2	17	35,495	17	1,559	1,004	49	18	100	35	15	100
A3	12	13,306	12	221	126	64	30	98	36	14	100
A4	67	31,151	67	1,479	821	64	29	99	52	23	100
A5	47	230,055	3	295	88	90	50	100	65	16	100

Tests and *Carved Tests* columns under the *Statement Coverage* and *Branch Coverage* headers). Additionally, the table relates the coverage of carved tests with the one of GUI tests (columns labeled with *Included* under the *Statement Coverage* and *Branch Coverage* headers) by reporting the percentage of coverage from carved tests that also appears in GUI tests.

Overall, ARTISAN carved 2,087 tests from 152 GUI tests. The carved tests cover 45.28% and 41.33% of the statements and branches that are covered by the GUI tests. The overall number of targets is 3,609, and the number of method invocations in the traces is 311,661. The difference between the number of method invocations in the traces and the number of targets is due to the fact that the traces contain a large number of method invocations whose definition is not inside the AUT (i.e., the methods are defined in the Java standard library, third-party libraries, or the Android framework) and due to the carving strategy we adopted.

The difference between the number of targets and the number of carved tests is caused by some limitations in the implementation of the technique (see Section IV) and our design choice to reject unit tests in which the method under tests are not *directly* called. Additionally, some targets cannot be carved by ARTISAN as those are methods in anonymous classes (e.g., callback handler definitions for GUI elements) that cannot be directly invoked in Java. To carve those targets, ARTISAN could use a preprocessing step that refactors the code of the app such that those methods are not part of anonymous classes. However, we did not implement such a solution as it could lead to unwanted changes by the developers. There is a need for studies and interviews with developers to investigate this aspect, and we leave those studies as a possible direction for future work.

The design choices, limitations in the implementation of the technique (Section IV), the focus on ARTISAN on executions originating in the main thread [33], and the focus on Android activities are the reasons why the coverage of carved tests is not 100%.

Table II also compares (columns *Included* under the *Statement Coverage* and *Branch Coverage* headers) the statement and branch coverage achieved by GUI and carved tests. Specifically, we look at how much of the coverage in carved tests also appears in the GUI tests used for generating them. In other words, these columns reveal whether the carved tests cover portions of the apps that are not covered by the GUI tests (i.e., lead to spurious coverage). We computed this information by extending JaCoCo [48], the coverage tool we used in the experiments. All the branches covered in the carved tests are also covered by the GUI tests. In terms of statement coverage, instead, there are a few statements (in two of the five apps) that are covered by the carved tests but not by the GUI tests. We analyzed the tests leading to the discrepancies and identified that the cause behind the discrepancy is a different behavior of some of the Android API methods when they execute on an Android device and the JVM (via Robolectric). This situation can appear because Robolectric is a partial model of the Android framework.

TABLE III: Time cost of running ARTISAN using Strategy-I.

ID	*GUI Tests Execution Time Before Instrumentation*	ARTISAN		
		Instrumentation Time	*GUI Tests Execution Time After Instrumentation*	*Carving Time*
A1	34s	10s	36s	7s
A2	2m56s	28s	3m26s	38m07s
A3	1m49s	18s	1m41s	31s
A4	6m10s	15s	6m24s	3m12s
A5	4m51s	19s	6m53s	6m16s

RQ1 answer: Yes, ARTISAN can carve tests from GUI tests. Additionally, carved tests cover 45.28% and 41.33% of the statements and branches that are covered by the GUI tests and rarely have spurious coverage.

2) *RQ2: What is the cost of running* ARTISAN*?:* Table III provides details on the execution time for running ARTISAN on the benchmark apps when using **Strategy-I**. For each app, the table provides the identifier for the app (*ID*), the time to execute the GUI tests before instrumenting the app (*GUI Tests Execution Time Before Instrumentation*), the time needed to instrument the app (*Instrumentation Time*), the time to execute the GUI tests after instrumenting the app (*GUI Tests Execution Time After Instrumentation*), and the time to carve the tests (*Carving Time*). The time values reported in Table III are averages across 10 runs of ARTISAN.

ARTISAN was able to generate carved tests in less than one hour for each app considered. The technique was the fastest when analyzing A1 and, in this case, ARTISAN took only 53 seconds. The technique took the longest when analyzing A2, roughly 42 minutes. The time to instrument the apps is negligible for the selected apps, especially considering that instrumentation is a one-time activity. The overhead introduced

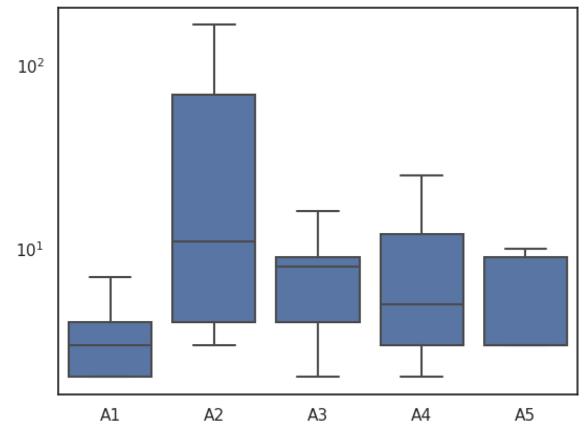

Fig. 3: Test size (y-axis) for carved tests generated by ARTISAN with Strategy-I. The graph uses a logaritmic scale.

by the instrumentation when running the GUI tests is 16.34%. We believe that the overhead is reasonable as it is low and did not affect the test execution behavior (i.e., all that passed before the instrumentation also passed after instrumenting the apps). The time to carve tests, instead, varies significantly between apps. We analyzed the causes behind the variance and identified that the main contributing factors are the presence of static method invocations in the traces and the size of the carved executions. Static method invocations affect the carving time as all the invocations to the same static classes need to be (conservatively) taken into account when analyzing the targets those invocations precede. Larger sets of required method invocations also affect carving time as it takes longer to extrapolate connected components from the graphs to generate the corresponding carved executions.

RQ2 answer: Based on the results of our evaluation, we believe that the time cost of running ARTISAN is low. For the apps considered, the technique always terminated within an hour and, in some cases, within a few minutes.

3) RQ3: What are the characteristics of carved tests?: To characterize carved tests, we consider the size of the unit tests and the number of mocks contained in them. We focus on test size as larger tests can be harder to maintain and can lead to test smells [49]. We also focus on the number of mocks as they are not always straightforward to set up [3], [50]; hence, having tests with them can potentially help app developers.

For each app considered, Figure 3 reports the size of the carved tests. The boxplot chart reports the size of the tests on the y-axis and uses the log scale. We computed the size of each test by counting the number of statements in the tests. The results reported in Figure 3 are promising. For four out of the five apps considered, the median number of statements in the tests is less than 10. A2 is the app with the largest number of statements per test. We observed that this app requires setting some values in its database for a larger number of tests which, in turn, led to an increase in the size of the tests. Considering that most tests have a reasonable size, we believe that ARTISAN can provide developers with tests that might be useful in debugging activities.

The total number of mocks in the carved tests, instead, is 241. The ratio between the number of tests and the number of mocks follows the ratio of developer-written tests in some of the apps analyzed by related work on test doubles in Android.

Based on our experimental results, we argue that the tests generated by ARTISAN, both in terms of test size and the mocks they provided, could be actionable for developers. However, to confirm this hypothesis, studies and interviews with developers are necessary; hence, we suggest them as possible future work.

RQ3 answer: the tests carved by ARTISAN tend to be concise and provide mocks that are required for testing certain parts of the apps. Considering the tests' characteristics, we believe that the tests can be actionable for developers.

4) RQ4: How do different carving strategies compare?: We compare the three carving strategies considered (**Strategy-I**, **Strategy-II**, and **Strategy-III**) in terms of execution time, coverage achieved by the generated tests, and generated tests' size.

The total execution time for **Strategy-I** across all apps is 48 minutes, for **Strategy-II** is 16 hours, and for **Strategy-III** is 10 hours. As we expected, **Strategy-I** is significantly faster than the other two strategies as these take into account a higher number of carvable targets. **Strategy-II** and **Strategy-III** are characterized by a timeout, and the timeout was hit once in **Strategy-II** and twice in **Strategy-III**. (This result is the reason why the total time for **Strategy-II** and **Strategy-III** is not 50 and 25 hours, respectively.) In the three cases that the timeout was hit, ARTISAN could not finish processing all the targets selected by these strategies. Note that increasing the timeout values would go against the idea behind these strategies, i.e., we selected the timeouts to study how the strategies would perform when used in a practical app development context.

Figure 4 and Figure 5 compare the statement and branch coverage of the tests generated by the three strategies. Interestingly, despite **Strategy-II** and **Strategy-III** considering more targets and taking longer to execute, the coverage achieved by the carved tests between **Strategy-I** and the other strategies is comparable. This observation suggests that **Strategy-I** might be more appealing in terms of coverage than the other ones. **Strategy-II** and **Strategy-III** generated a higher number of tests than **Strategy-I** ($10,756$ and $7,867$ versus $2,087$) and might have better fault-finding ability. We leave the investigation of the fault-finding ability of carved tests as future work. In the case of A5, **Strategy-I** had better coverage than

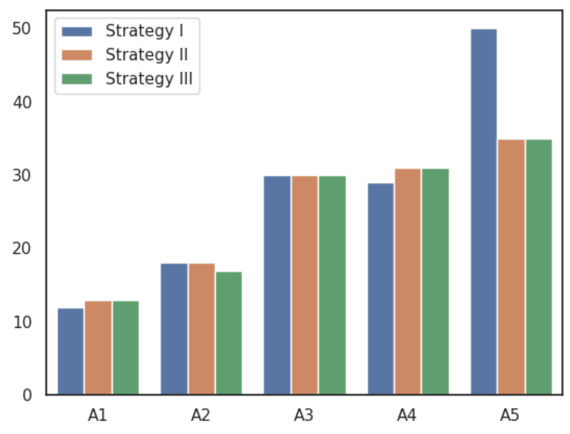

Fig. 4: Statement coverage comparison between different carving strategies.

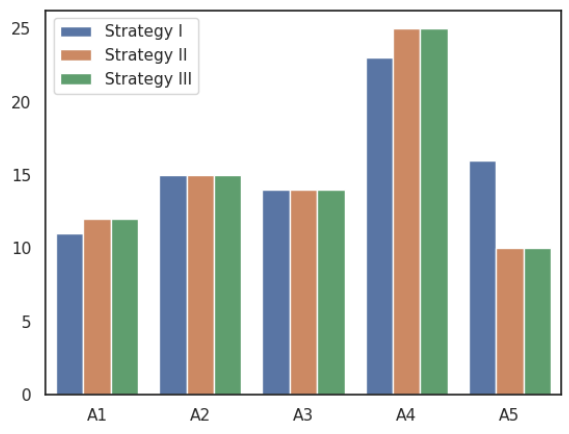

Fig. 5: Branch coverage comparison between different carving strategies.

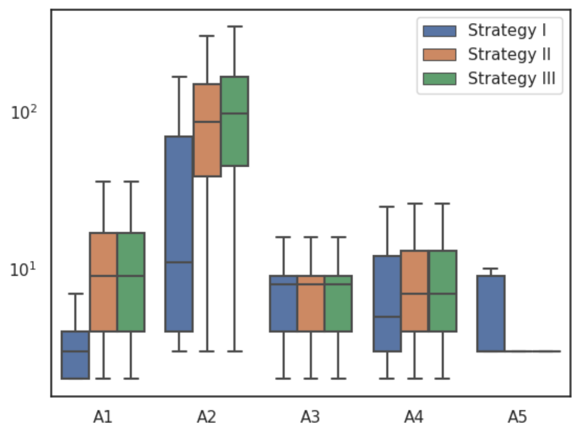

Fig. 6: Test size comparison between different carving strategies.

the other two strategies. The reason behind this result is that **Strategy-II** and **Strategy-III** took a very long time to process the targets in one of the traces, and the timeouts were triggered, thus leading to the generation of a small number of tests.

Figure 6 compares the size of the tests generated using the different strategies. The y-axis reports the size of the tests, while the chart uses a log scale. Overall, the test size associated with **Strategy-I** is smaller than the other two strategies. This result is justified by the fact that **Strategy-II** and **Strategy-III** considered multiple targets for the same method, and targets appearing later in the traces required a larger number of statements to recreate the preconditions for invoking the targets.

RQ4 answer: Overall, we believe that **Strategy-I** is the most cost-effective strategy as it achieved similar coverage compared to the other two strategies while being more efficient.

D. Discussion

ARTISAN is an end-to-end technique to perform tests carving for Android apps. Implementing such a technique required a significant technical effort, the mastery of several technologies, and knowledge from different domains besides Android. For instance, we employed byte code modification for instrumenting and tracing the execution, automated build systems for setting up the evaluation benchmarks, graph theory and program analysis to carve the execution traces, and Java code generation to synthesize the unit tests.

There are parts of ARTISAN that provide a solid base for future work and others that can be further refined. For instance, instrumentation and tracing of apps, as well as experiment automation, worked smoothly. Similarly, the carving algorithm and the generation of mocks produced excellent results. However, the carving algorithm also has some cost when carving apps that heavily employ loops or implement overly complex methods. In these cases, carving might take too long and produce excessively long unit tests that can quickly become hard to inspect and manage. We postulate that in these cases, it might be beneficial to combine action-based and state-based carving such that unit tests can directly load large objects created with long method sequences, or loops, from memory. An alternative way to reduce the size of the carved unit tests might be using other dynamic techniques, such as delta debugging [51], or employing purity analysis to identify and filter out method invocations that do not introduce any relevant dependency. Additionally, different carving strategies could be explored to make action-based carving more efficient and effective.

Since ARTISAN selects invocations of non-private methods as carvable targets, many of the generated tests might test the AUT under similar or the same execution conditions. Consequently, the resulting carved test suites might contain duplicated tests that should be removed. In future work, we plan to investigate the application of test suite reduction techniques [52] to identify and remove duplicate tests.

E. Threats to Validity

As it is the case for most empirical evaluations, there are both external and construct threats to validity associated with the results we presented. Our results might not generalize to other apps in terms of external validity. In particular, we only considered five apps. This limitation is an artifact of the complexity involved in setting up the infrastructure to run the apps, which might require customized build configurations and manually inspecting the results of our analysis. We selected apps of different sizes that belong to different app categories and are already considered in related work to mitigate this threat. In terms of construct validity, there might be errors in the implementation of our technique. To mitigate this threat, we extensively inspected the evaluation results manually.

VI. Related Work

The idea of test carving was originally proposed by Elbaum et al. [23] as a means to generate unit tests, dubbed Differential Unit Tests, to spot regression errors in Java programs. Elbaum and co-authors identified three main test carving paradigms: state-based carving, in which carving takes place on the system under test's state recorded during execution; action-based carving, in which carving takes place on the sequence of method invocations recorded during execution; and hybrid carving, which combines the previous two. However, they implemented only state-based carving. In this area, remarkable results have been achieved by Krikava and Vitek [24], who proposed GENTHAT for generating unit tests from execution traces of R libraries, Kampmann and Zeller [25], who proposed BASILISK for state-based carving of parameterized unit tests targeting C programs, and, Juvekar et al. [27], who created a program executing all public methods on a given object the same way as in a given program trace. Compared to those works, ARTISAN implements a different form of test carving, works on Android apps, generates unit tests across different platforms, and augments carved tests with automatically synthesized mocks.

ARTISAN generates focused unit tests from execution traces. Pasternak et al. [32], Saff et al. [29], and Thummalapenta et al. [31] achieved the same goal but with different techniques that, respectively, selected the interactions to recreate the state of objects until a certain point in time, automatically created focused unit tests by test factoring and environmental mocks generation, and mined an extensive collection of execution traces to generate generic parameterized tests using dynamic symbolic execution to cover paths not contained in the traces. Unlike these techniques, ARTISAN is not limited to inter-object interactions implemented as method calls, considers method invocations invoked by frameworks, and generates focused unit tests.

Action-based carving sets unit tests' preconditions, i.e., objects' state, by re-executing specific sequences of method calls that have been observed during end-to-end testing. Therefore, action-based carving is a sensible solution to the object creation problem defined by Bach et al. [53]. The main differences between ARTISAN and the work done by Bach and co-authors lie in the fact that they first identified feasible method-call sequences for object creation statically and then selected the most desirable sequence using a search algorithm. Another difference is that Bach et al.'s approach is designed for C++ programs.

Alternative approaches make use of selective capture and replay techniques. For instance, Orso et al. proposed SCARPE [28] and JINSI [26] to capture parts of program execution for later replay and isolate the instructions that lead to the failure to produce a minimal example that reliably replays it. Compared to those works, ARTISAN has a broader scope as it does not consider interactions involving a single component, can generate tests from both normal and exceptional system executions and is capable of carving unit tests for Android apps.

VII. Conclusions and Future Work

This paper presented ARTISAN, a technique to perform test carving for Android apps. ARTISAN carves unit tests from GUI tests by collecting method invocations in the AUT while running the GUI tests on Android devices, extracting target method invocations and their preconditions from the collected information, and synthesizing preconditions and target method invocations into portable unit tests. We evaluated ARTISAN on five apps using different carving strategies and identified that the technique is able to carve tests that achieve 45% of the original GUI tests' coverage and does so in an amount of time compatible with standard development practices.

In future work, we plan to perform studies and interviews with developers to understand which carved tests best help developers and explore alternative carving strategies based on the gathered insights. We also plan to investigate test suite reduction techniques to identify duplicate tests among carved tests. We plan to extend ARTISAN to support apps written in different programming languages (e.g., Kotlin), and, more on the engineering side, we also plan to extend the implementation of ARTISAN to support additional Android features. Finally, we plan to investigate techniques to carve oracles from end-to-end tests into oracles suitable for unit tests.

REFERENCES

[1] Google, "Fundamentals of testing android apps." [Online]. Available: https://developer.android.com/training/testing/fundamentals

[2] J.-W. Lin, N. Salehnamadi, and S. Malek, "Test automation in open-source android apps: A large-scale empirical study," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1078–1089.

[3] M. Fazzini, C. Choi, J. M. Copia, G. Lee, Y. Kakehi, A. Gorla, and A. Orso, "Use of test doubles in android testing: An in-depth investigation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022.

[4] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," *Acm Sigplan Notices*, vol. 48, no. 10, pp. 623–640, 2013.

[5] 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*. New York, NY, USA: Association for Computing Machinery, 2014, pp. 204–217.

[6] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 94–105.

[7] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 245–256.

[8] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, 2017, pp. 23–26.

[9] Google, "Ui/application exerciser monkey." [Online]. Available: https://developer.android.com/studio/test/other-testing-tools/monkey

[10] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering*. Piscataway, NJ, USA / New York, NY, USA: Institute of Electrical and Electronics Engineers / Association for Computing Machinery, 2019, pp. 269–280.

[11] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, 2019, pp. 1070–1073.

[12] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, "Combodroid: generating high-quality test inputs for android apps via use case combinations," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 469–480.

[13] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 481–492.

[14] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 153–164.

[15] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso, "Barista: A technique for recording, encoding, and running platform independent android tests," in *2017 IEEE International Conference on Software Testing, Verification and Validation*. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, 2017, pp. 149–160.

[16] Google, "Create ui tests with espresso test recorder." [Online]. Available: https://developer.android.com/studio/test/other-testing-tools/espresso-test-recorder

[17] Y. Liu, Y. Lu, and Y. Li, "An android-based approach for automatic unit test," in *International Conference on Cyberspace Technology (CCT 2014)*, 2014, pp. 1–4.

[18] J. Cao, H. Huang, and F. Liu, "Android unit test case generation based on the strategy of multi-dimensional coverage," in *7th International Conference on Cloud Computing and Intelligent Systems*, 2021, pp. 114–121.

[19] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[20] M. Fowler, "Testdouble." [Online]. Available: https://martinfowler.com/bliki/TestDouble.html

[21] Robolectric, "Robolectric." [Online]. Available: http://robolectric.org

[22] ——, "Shadows." [Online]. Available: http://robolectric.org/extending

[23] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2006, p. 253–264.

[24] F. Krikava and J. Vitek, "Tests from traces: automated unit test extraction for R," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 232–241.

[25] A. Kampmann and A. Zeller, "Carving parameterized unit tests," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*. Piscataway, NJ, USA / New York, NY, USA: Institute of Electrical and Electronics Engineers / Association for Computing Machinery, 2019, pp. 248–249.

[26] A. Orso, S. Joshi, M. Burger, and A. Zeller, "Isolating relevant component interactions with jinsi," in *Proceedings of the 2006 international workshop on Dynamic systems analysis*. New York, NY, USA: Association for Computing Machinery, 2006, pp. 3–10.

[27] S. Juvekar, J. Burnim, and K. Sen, "Path slicing per object for better testing, debugging, and usage discovery," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-132, Sep 2009. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-132.html

[28] A. Orso and B. Kennedy, "Selective capture and replay of program executions," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, 2005.

[29] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for java," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA: Association for Computing Machinery, 2005, pp. 114–123.

[30] D. Saff and M. D. Ernst, "Mock object creation for test factoring," in *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. New York, NY, USA: Association for Computing Machinery, 2004, pp. 49–51.

[31] S. Thummalapenta, J. de Halleux, N. Tillmann, and S. Wadsworth, "Dygen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces," in *Tests and Proofs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 77–93.

[32] B. Pasternak, S. Tyszberowicz, and A. Yehudai, "Genutest: a unit test and mock aspect generation tool," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 4, pp. 273–290, 2009.

[33] Google, "Processes and threads overview." [Online]. Available: https://developer.android.com/guide/components/processes-and-threads

[34] ——, "Application fundamentals." [Online]. Available: https://developer.android.com/guide/components/fundamentals

[35] A. Authors, "Artifact for action-based test carving for android apps." [Online]. Available: https://zenodo.org/record/7285409

[36] S. Faber, B. Dutheil, R. Winterhalter, and T. van der Lippe, "Mockito." [Online]. Available: https://site.mockito.org/

[37] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, 1991. [Online]. Available: https://doi.org/10.1145/115372.115320

[38] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, R. E. K. Stirewalt, A. Egyed, and B. Fischer, Eds. ACM, 2007, pp. 417–420. [Online]. Available: https://doi.org/10.1145/1321631.1321698

[39] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First*

Decade High Impact Papers. Armonk, NY, USA: IBM Corp., 2010, p. 214–224.

[40] J. Clause, W. Li, and A. Orso, “Dytan: a generic dynamic taint analysis framework,” in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 196–206.

[41] J. Newsome and D. X. Song, “Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software.” in *NDSS*, vol. 5, 2005, pp. 3–4.

[42] G. Fraser and A. Zeller, “Mutation-driven generation of unit tests and oracles,” *IEEE Trans. Software Eng.*, vol. 38, no. 2, pp. 278–292, 2012. [Online]. Available: https://doi.org/10.1109/TSE.2011.93

[43] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, “TOGA: A neural method for test oracle generation,” in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 2130–2141. [Online]. Available: https://doi.org/10.1145/3510003.3510141

[44] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, “On learning meaningful assert statements for unit test cases,” in *ICSE ’20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 1398–1409. [Online]. Available: https://doi.org/10.1145/3377811.3380429

[45] GitHub, “Github.” [Online]. Available: https://github.com

[46] S. Thorve, C. Sreshtha, and N. Meng, “An empirical study of flaky tests in android apps,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 534–538.

[47] Google, “¡uses-sdk¿.” [Online]. Available: https://developer.android.com/guide/topics/manifest/uses-sdk-element

[48] JaCoCo, “Jacoco.” [Online]. Available: https://www.jacoco.org

[49] G. Grano, F. Palomba, D. Di Nucci, A. De Lucia, and H. C. Gall, “Scented since the beginning: On the diffuseness of test smells in automatically generated test code,” *Journal of Systems and Software*, vol. 156, pp. 312–327, 2019.

[50] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, “To mock or not to mock? an empirical study on mocking practices,” in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 402–412.

[51] A. Zeller, “Isolating cause-effect chains from computer programs,” in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2002, pp. 1–10.

[52] S. ur Rehman Khan, S. P. Lee, N. Javaid, and W. Abdul, “A systematic review on test suite reduction: Approaches, experiment’s quality evaluation, and guidelines,” *IEEE Access*, vol. 6, pp. 11 816–11 841, 2018.

[53] T. Bach, R. Pannemans, and A. Andrzejak, “Determining method-call sequences for object creation in C++,” in *Proceedings of the 13th IEEE International Conference on Software Testing, Validation and Verification*. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, 2020, pp. 108–119.