Observation-based unit test generation at Meta

Nadia Alshahwan*
Mark Harman
Alexandru Marginean
Rotem Tal
Eddy Wang
Meta Platforms Inc.,
Menlo Park, California, USA

ABSTRACT

TestGen automatically generates unit tests, carved from serialized observations of complex objects, observed during app execution. We describe the development and deployment of TestGen at Meta. In particular, we focus on the scalability challenges overcome during development in order to deploy observation-based test carving at scale in industry. So far, TestGen has landed 518 tests into production, which have been executed 9,617,349 times in continuous integration, finding 5,702 faults. Meta is currently in the process of more widespread deployment. Our evaluation reveals that, when carving its observations from 4,361 reliable end-to-end tests, TestGen was able to generate tests for at least 86% of the classes covered by end-to-end tests. Testing on 16 Kotlin Instagram app-launch-blocking tasks demonstrated that the TestGen tests would have trapped 13 of these before they became launch blocking.

KEYWORDS

Automated test generation, unit testing, test carving

1 INTRODUCTION

This paper describes our experience developing and deploying observation-based TestGen ('TestGen-obs' or 'TestGen' for short[1]). TestGen is a tool that automatically generates unit and regression tests from scratch using scalable serialized observations of salient objects created during app executions; the tests are generated in Kotlin in order to test Meta's Java and Kotlin code bases.

---

*Author order is alphabetical. The corresponding author is Mark Harman.

[1]We also have an entirely separate tool, called 'TestGen-LLM' [3], which *extends existing human-written* test cases using Assured LLM-Based Software Engineering [6]. By contrast, TestGen-obs generates new tests from scratch using observations.

Our intention with TestGen is not to fully replace human-written unit and integration testing. Nevertheless, at the scale Meta operates (hundreds of millions of lines of code and over 3 billion users) it is infeasible to rely *solely* on human engineering effort for test case design.

Therefore, we have embarked on an automation program for over a decade, using both well-established and novel software engineering research to deploy automated test design at scale. This process started with the deployment of end-to-end automated test design tools such as Sapienz [4], and static analysis tools such as Infer [16]. We went on to deploy automated simulation-based test generation for higher level testing of whole communities of inter-acting users [1, 19, 40].

This previous work covered the system level testing requirements such as end-to-end and social testing. More recently, in order to shift testing leftwards in the development process [5], Meta has been targeting unit test design automation. Consequently, we are now tackling the problem of automatic generation of integration and unit tests, hence TestGen.

A TestGen test is a machine-generated unit test that captures the current behavior of the function under test, and thus it is capable of detecting regressions. The tool works by first automatically instrumenting the app to record runtime values of functions for which we want to generate unit tests. It records current object instances, return values and parameter values. We call these recorded runtime values 'observations'. TestGen automatically saves the observations when the instrumented version of the app executes the target function under test. This can be done either by running the app manually or using existing app execution tooling, such as Jest End-to-End (E2E) testing or the Sapienz test infrastructure, which has been deployed since 2018 [4, 32].

A TestGen test asserts that, when called on the observed object instance with the observed parameter(s), the function under test produces the expected previously-observed return value. For the case of functions without a `return`, TestGen tests assert that the function executes without exception.

TestGen was originally implemented to generate unit tests for Meta's Instagram app, so the results we report in this paper are based on TestGen's deployment for Instagram. However, there is nothing specific to Instagram in how TestGen operates and it is now being deployed across other Meta platforms and products.

TestGen tests are regression tests: they assert that the method under test in a diff[2] behaves the same as it did when executed on the current main branch (called 'master'). This has 2 primary benefits:

(1) It fully automates unit test generation, without the need for human intervention to define the assertions (there is no need for an oracle [9]).

(2) It produces realistic test cases, thereby avoiding false positives. The tested values can occur at runtime because they have been observed in a previous execution of the app under exactly the same circumstances as those tested.

The primary contributions of this paper are:

(1) A scalable industrial-strength observation-based unit test generation system: TestGen.

(2) A description of the principal novel technical features required to achieve sufficient scalability that allowed us to deploy observation-based testing at Meta: depth-aware serialization/deserialization, and an observation–aware Android memory manager.

(3) A report of our experience of TestGen deployment, where it has thus far revealed 5,702 faults, and demonstrated its potential to trap at least 81% of the important issues that might otherwise impact the launch of new versions of the Instagram app.

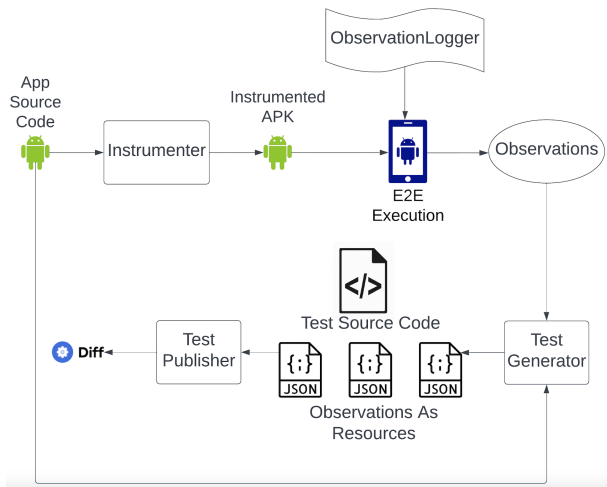## 2 META'S OBSERVATION-BASED TESTGEN SYSTEM

Figure 1: The Architecture of TestGen. TestGen instruments the app source code to record observations at runtime using the ObservationLogger. The TestGenerator uses the observations and the app source code to produce fully-runnable unit tests. Finally, the TestPublisher integrates with the build system and publishes a diff with the tests.

Given a method under test, foo, on an object of class A, TestGen produces (potentially many) unit tests, each of which reflects a

single execution trace and asserts that: When called on an observed object instance of type A, with the parameters that TestGen has observed in a previous execution, foo returns the same value as that observed in this previous app execution. This is a form of Test Carving [20], as we explain in more detail in the Related Work section (Section 4).

TestGen designs its tests, based on observations from previous app executions. It collects these observations by building a specially instrumented version of the app. The instrumented app deploys unobtrusive and fast probes that collect the serialized values subsequently used to build tests. The instrumentation logic does not affect the functional behavior of the app. It simply collects observations silently in the background. It is also designed to be lightweight, so as not to unduly affect app execution time, which might otherwise influence test behavior.

The instrumented app dumps observations into a data store. Currently, we use the standard Android app DB as the data store that we optimized for a highly concurrent environment with a high volume of writes. Because the executions are real E2E executions of the whole app, the values observed are highly realistic and faithfully reflect those that could occur in production.

Figure 1 depicts the architecture of TestGen. The source code of the app is compiled with the 'Instrumenter' compiler plugin that produces the 'Instrumented APK'. At runtime, the 'Instrumented APK' calls the 'ObservationLogger' to record observations. The observations together with the initial app source code are compiled with the 'Test Generator' plugin that constructs the tests, using the deserialized json representation of the observations. Finally, the 'Test Publisher' constructs the build system files and dependencies for the test cases, running them 5 times to confirm they are not flaky. For all tests that pass and are not flaky the TestPublisher produces a source control revision (i.e., a 'diff') that can be landed. Once this happens, Meta's Continuous Integration (CI) platform automatically runs these tests, both continuously and also every time an engineer submits a diff.

Test Maintenance: Like any other Android unit test deployed at Meta, TestGen tests will block diffs from landing when they fail. It is thus important that signals that TestGen tests provide are highly actionable and that test maintenance does not become a burden for Meta engineers. It is also important that TestGen provides automatic ways to unblock engineers when a test fails. TestGen relies on the assertion failure message to provide actionable information on how to unblock when a test fails. The assertion failure exactly describes the difference that caused the failure:

```
This is an automatically generated unit test from TestGen.
TestGen equality assertion failed! When comparing the return from
the method under test with the expectation for the field `X` of
the class `C` we saw the value `true` while we expected the
value `false`. The expected return for this test is in the
resource file `path/to/resource` at line: x column: y.
If this change is not expected after the change in your diff, you
can debug this test with the debugger in Android Studio
```

From this point, the diff author has the following options:

• If the test failure is not expected after the change, the diff author will debug their code to fix the issue.

• If the test failure is the result of an expected change in behavior then TestGen provides the following automated single-click resolutions:

---

[2]At Meta, a pull request is known as a 'diff' (short for 'differential'), following the Mercurial repository management nomenclature.

– Update the assertion to reflect the new behavior.
– Call TestGen again to regenerate the test (if the change is more fundamental than a simple assertion failure).
– Delete the test if the diff author deems it to be no longer needed.

The rest of this section describes the implementation of the main TestGen components.

2.1 Instrumenter Plugin

The Instrumenters are distinct compiler plugins that we implemented for Java and Kotlin. In order to collect observations at runtime, an 'Instrumenter' adds extra code into the app at compile time, to explicitly log the observations.

The Kotlin and Java compilers use an Intermediate Representation (IR). The TestGen Instrumenter plugin modifies the IR of the Kotlin/Java code rather than the source code. The compilers generate standard bytecode from their IR representations. The compiler-generated bytecode thereby includes the instrumentation logic. The Instrumented bytecode makes calls to the 'ObservationLogger' (discussed in subsection 2.2) to do the heavy lifting of actually logging the observations. Figure 2 shows an example of this IR instrumentation added for a Kotlin function.

The Instrumenter constructs an instance of the *ObservationLogger* at line 2 in Figure 2, the first parameter of which specifies that observations should be saved to the app DB ('APP_DB'). The second parameter specifies that a random seed should not be used in this instance[3]. The third parameter specifies that the observations should be logged on a different thread (to improve scalability).

The Instrumenter also adds instrumentation code that, when executed, will record observations for the:

- **Current Object Instance**: The instrumenter adds a call to logObs at the start of the method body, to log the value of the current object instance. Line 3 in Figure 2 shows an example.
- **Method Parameters**: The instrumenter adds calls to logObs after the logging call for current object instance. It will add a call for each function parameter in turn. Lines 4–5 in Figure 2 show two such examples.
- **Method Returns**: At each `return` statement, the Instrumenter adds a call to the logObs as a meaning–preserving transformation. It creates a temporary variable, in which it holds the returned value, then passes this to logObs, and finally returns it. Lines 8–17 in Figure 2 show an example of this. The block that appears between lines 8–15 is the way in which TestGen is able to log returns with the semantics-preserving transformation.

Although it is possible to instrument the whole app, it can be very time consuming to completely rebuild a whole app, such as Instagram [26]. As a result, TestGen limits the observation logging to the set of files for which tests must be generated.

2.2 ObservationLogger

To tackle the observation logging scalability challenge, we introduce DASAD (Depth-Aware Serialization And Deserialization). DASAD

introduces depth-aware serialization/deserialization together with a pointer-based observation sharing mechanism that uses a pointer-aware Android memory management layer. This scales the collection of observations, their serialization and serialization protest generation. With DASAD observations can be made from the app, while running, without and affecting its performance.

By contrast, a simple reflection-based approach to fully serializing each relevant object used by an app would not scale: the instrumented app would soon stop responding (i.e., AnR[4]) since such a simple approach would end up serializing large amounts of entire Android heap store. Furthermore, the space required would also make this infeasible. For example, at the scale of the Instagram App, which is characteristic of the larger of the commercially available apps, it takes hours to serialize a single object instance without DASAD, which would clearly impact the performance of the app under test, rendering any attempt at observation-based test generation approach infeasible.

In the remainder of this section, we explain how DASAD addresses the two scalability challenges:

(1) Serializing large complex objects,
(2) Supporting concurrent runtime environments.

2.2.1 Serializing Large And Complex Objects. In order to serialize large objects, DASAD uses a combination of depth awareness and a pointer-based observation representation approach, as explained below.

Depth-Aware Serialization: Fortunately, for the TestGen use case, it is usually *unnecessary* to record the entire object for the purpose of generating tests. Instead, a test case may only need to retrieve values observed in nested objects up to a given depth. This may also generalize to many other use cases, where scalability can be achieved in a similar depth-aware manner, but here we focus exclusively on the TestGen use case. To illustrate how depth awareness can help tackle scale for the test generation use case, consider the following Java classes:

```
1   class Foo{A a;}
2   class A {B b;}
3   class B {C c;}
```

Suppose we want to serialize an instance of Foo. A full serialization will recursively contain all 3 fields: a, b, and c. If we serialize with a depth limit of 1, then the serialization will contain only the field a, in which field b will be null. Depth-aware serialization simply replaces all fields that are nested deeper than the given depth with null.

Pointer-Aware Serialization in a Android Memory Manager: Even with the depth-aware serialization, the number of objects produced is too numerous and large for it to be feasible to record them all. The core intuition that we used to tackle this issue is that most of the objects are sharable by an E2E run (consider, for example, the Android App context, used in many functions with an identical instance). To avoid duplication we constructed a memory management system on top of Android that the serialization uses to determine when objects are the identical, and can thus be shared in memory.

Algorithm 1 describes DASAD serialisation. The serialization-function *serialize_object* receives as input an object to be serialized.

---

[3]For reproducability of the runs, TestGen also allows the use of a predefined random seed.

```
1   fun <elided method name>(session: Session?, intent: Intent?): Boolean {
2       val tmp0_obs: ObservationLogger = ObservationLogger(logTo = "APP_DB", randomSeed = null, logOnDifferentThread = true)
3       tmp0_obs.logObs</* null */>(observedValue = <this>, lNr = -1L, colNr = -1L, fileName = "file/path/Class.kt", obsType = "CURRENT_OBJ_INST")
4       tmp0_obs.logObs</* null */>(observedValue = session, lNr = 142L, colNr = 31L, fileName = "file/path/Class.kt", obsType = "PARAMETER")
5       tmp0_obs.logObs</* null */>(observedValue = intent, lNr = 142L, colNr = 54L, fileName = "file/path/Class.kt", obsType = "PARAMETER")
6       var intent: Intent? = intent
7       ...
8       { // BLOCK
9         val tmp0: Boolean = { // BLOCK
10          val tmp2_elvis_lhs: Boolean? = <this>.<get-isTestedCondition>()
11          when {
12            EQEQ(arg0 = tmp2_elvis_lhs, arg1 = null) -> false
13            else -> tmp2_elvis_lhs
14          }
15        }
16        tmp0_obs.logObs</* null */>(observedValue = tmp0, lNr = 153L, colNr = 4L, fileName = "/file/path/Class.kt", obsType = "RETURN")
17        return tmp0
18      }
19  }
```

Figure 2: Kotlin Intermediate Representation (IR) of an example instrumentation. tmp0_obs is an instance of the observation logger. logObs is its method that logs the observations to the app DB. The grayed code is code that the instrumenter has added.

The variable *isFirstSerialization* keeps track whether or not we are serializing the current object for the first time: line 17 gets the object id and checks whether it is in *seenObjects*, a hash set of seen objects ids. The object id is a hash over the object state, such that it uniquely identifies an object. If it is the first serialization, lines 18 and 19 set the value of *isFirstSerialization* to true and add it into the 'seenObject'. We do this in a syncronized block since this code will be called in a highly concurrent environment. If the object was previously serialized, TestGen simply marks its serialization as *POINTS_TO_OBJECT_ID* and stops the serialization process.

At deserialization time, the algorithm looks for a full serialization for the same object id to reconstruct it. If the object is encountered for the first time, we proceed with each serialization by recursively calling the *serialize_object* function on each of its fields. The variable *objSerialization* is a string that contains the json representation of the serialized object.

TestGen has to handle pointers in deserialization. TestGen applies Algorithm 2 to instantiate pointers, prior to the test's execution. In Algorithm 2, the entry point is the function *get_full_serialization_map* (line 41). This function is applied across all observations, for all tests generated from the same E2E run. It returns a set of all full initializations across all these observations (this is recursive, becuase each field in the recursive traversal of an object can contain other objects that are either full serializations or pointer serializations). The first step, is to construct the set of *pointerKeys* (lines 42–44) that contains all the keys of serializations that are pointers.

Lines 45-47 construct the list of all full serializations that have at least one pointer serialization. Having this list will later allow us to replace all pointer serializations, such that we obtain observations containing only full serialization (as the test runtime requires). Because of the recursive nature of Algorithm 1, observations might still contain pointer serializations for various fields

Algorithm 1 Pointer-Aware Object Serialization: the algorithm that TestGen uses to log observations at scale: it keeps track of already serialized objects to avoid multiple serializations of the same objects.

```
1:  function IS_PRIMITIVE(object)
2:         ▷ This function return true if the object is primitive and false otherwise.
3:
4:  function PRIMITIVE_SERIALIZATION(object)
5:         ▷ This function dumps the representation of a primitive object.
6:
7:  function APPEND_FIELD_SERIALIZATION(objSerialization, fName, fSerialization)
8:         ▷ This function appends to the json representation of the object (objSerialization) with the
    serialization of the field named fName: fSerialization.
9:
10: seenObjects ← HashSet<String>
11:
12: function SERIALIZE_OBJECT(object)
13:     if is_primitive(object) then
14:         return primitive_serialization(object)
15:     isFirstSerialization ← false
16:     synchronized {
17:     if object.get_id() ∉ seenObjects then
18:         isFirstSerialization ← true
19:         seenObjects.add(object.get_id())
20:     }
21:     if !isFirstSerialization then
22:         return "POINTS_TO_" + object.get_id()
23:     objSerialization ← ""
24:     for each field in object.get_fields() do
25:         objSerialization ← append_field_serialization (objSerialization, field.name(),
    serialize_object(field))
26:     return objSserialization
```

in their recursive traversal. This is what the rest of the logic in *get_full_serialization_map* tackles.

Lines 48-49 call the method *init_serialization*. This function parses each *fullSerialization* in turn and replaces all its pointer serializations (e.g., across its fields) with full serializations. The underlying data structure for *fullSerialization* preserves the pointer relationships after replacement. That is, if we replace a field that

is a pointer serialization P2 in a full serialization S1 with its corresponding full serialization S2, the field will keep pointing to the full serialization S2 rather than copying it. Therefore, the algorithm need only parse this list once: further initializations will include the previously processed full serializations.

fullSerializations might contain fields that are still stored as pointer serializations until lines 48-49 finish execution. The algorithm replaces all pointer serializations in *init_serialization* across all the observations and these replacements are also reflected in earlier replacements (when a particular observation might not be fully initialized). Therefore, the algorithm guarantees that, from line 50, there will be no pointer serializations remaining.

Additionally, *init_serialization* does not process deeper than the *MAX_DEPTH* limit because those values would be unneeded in any case.

At this point, the *full_serialization*s will contain recursion (since data types can be recursive). Additionally, although we restrict observations to *MAX_DEPTH* with the depth-aware serialization, *init_serialization* might overshoot this depth, because of initializing the pointer serialization. For example, suppose the app has a nested class structure like this: A { b: B {c: C {d: D { ...}} }}. Suppose we have an object, ObjA, of type A that has pointers to nested elements. These nested elements may become instantiated, with the effect that the effective depth for ObjA will be as deep as its nesting, irrespective of the maximum depth to which DASAD serializes. In this way, *MAX_DEPTH*, is merely the maximum depth to which serialization will process in a single traversal, but not the maximum depth that will be achieved overall.

fullSerializations is a dictionary with values that can be pointers (line 16 in 2 adds these pointers). Thus, any processing of these pointers will affect their values in all the *fullSerializations* in which they appear. Since next we want to remove elements that bypass *MAX_DEPTH*, we first need to deep copy (in line 50). For example, suppose a full serialization, A, with pointer P is already at *MAX_DEPTH - 1*. In this case, we want only depth 1 in pointer P. But P can also be in another full serialization, B, where it is at depth 1. Here we need to keep P with *MAX_DEPTH - 1*. The deep copy allows us to do this by making each occurrence of P in *fullSerializations* a distinct value, rather than then pointers to the same value.

The final stage of *get_full_serialization_map* removes recursion and keeps elements only up to *MAX_DEPTH* in the call to *remove_rec* at lines 51–52. *remove_rec* is a recursive function that passes each full serialization. When it reaches *MAX_DEPTH*, it simply removes the field values. To remove recursion, it keeps track (in *seen*) of the objects encountered when parsing a top level full serialization. If the recursive traversal reaches an already seen object, we must be in a recursive case and thus replace the value with a 'recursion' annotation. The outputs of *get_full_serialization_map* are then simply used as a string replacement over the set of all observations, to obtain the set of all fully initialized observations that the test cases can use at runtime.

2.2.2 Support For A Very High Concurrent Runtime Environment.
Many observations will be serialized and logged concurrently. This leads to a third scalability challenge for the 'ObservationLogger': support for highly concurrent writes to an external storage medium.

Algorithm 2 Pointer-Aware Object Deserialization: the algorithm that instantiates observations prior to generating tests. The pointer-aware serialization leaves pointers in observations as an optimizations. Before the test uses the observations, they need to be fully-initialized such that the objects can be constructed at runtime.

```
1: function POINTER_SERIALIZATIONS(observation)
2:                    ▷ This function returns all the keys of the pointer serialization.
3:
4: function FULL_SERIALIZATIONS(observation)
5:                    ▷ This function returns all the full serializations .
6:
7: function INIT_SERIALIZATION(p, key, obs, initialized, seen, depth)
8:     if depth > MAX_DEPTH then return
9:     if is_instance(obs, list) then
10:        for i, it in enumerate(obs) do
11:            init_serialization(obs, i, it, initialized, seen, depth + 1)
12:    if !is_instance(obs, dict) then return
13:    hash ← compute_hash_key(obs)
14:    if hash ∈ seen_keys then return
15:    if is_pointer_serialization(obs) then
16:        p[key] ← initialized[hash]
17:    seenKeys ← seenKeys|hash
18:    for k, val ∈ obs.items do
19:        if is_instance(val, dict) then
20:            init_serialization(obs, k, val, initialized, seen, depth + 1)
21:        if is_instance(val, list) then
22:            for i, it ∈ enumerate(val) do
23:                init_serialization(val, i, it, initialized, seen, depth + 1)
24:
25: function REMOVE_REC(p, key, obs, initialized, seen, depth)
26:     if depth > MAX_DEPTH then
27:         if is_instance(obs, dict) then parent[key] ← {}
28:         else if is_instance(obs, list) then parent[key] ← []
29:     if is_instance(obs, list) then
30:         for i, it in enumerate(obs) do
31:             remove_rec(obs, i, it, initialized, seen, depth + 1)
32:     if !is_instance(obs, dict) then return
33:     hashKey ← compute_hash_key(obs)
34:     if hashKey ∈ seen then
35:         p[key] = annotate_recursion(p[key])
36:         return
37:     seenKeys ← seen|hashKey
38:     for k, val ∈ obs.items do
39:         remove_rec(obs, k, val, initialized, seen, depth + 1)
40:
41: function GET_FULL_SERIALIZATION_MAP(observations)
42:     pointerKeys ← set()
43:     for observation in observations do
44:         pointerKeys.update(pointer_serializations(observation))
45:     fullSerializations ← {}
46:     for observation in observations do
47:         fullSerializations.update(full_serialization (observation))
48:     for s in fullSerializations do
49:         init_serialization(∅, ∅, s, fullSerializations, set(), 0)
50:     fullSerializations ← deep_copy(fullSerializations)
51:     for s in fullSerializations do
52:         remove_rec(∅, ∅, s, fullSerializations, set(), 0)
53:     return fullSerializations
```

To write the observations to an external storage for later processing, we use the Android app DB, in which we store them as soon as the 'ObservationLogger' serializes this. To tackle the scalability challenge of this highly concurrent writing, we:

(1) Execute all the serialization logic and the saving of the observations to the App DB on different background threads, that are not prioritized by the OS.

(2) Optimise the Android app DB for multiple concurrent writes. For this, we enabled write ahead logging[5] and set synchronous to off[6]

---
[5]https://www.sqlite.org/wal.html
[6]https://www.sqlite.org/pragma.html#pragma_synchronous

Once the observations are written to the app DB, the next steps of TestGen read them from the DB and store them as resources files, in json format, together with the generated test case source code.The generated test case thus consists of a Kotlin unit test that deserializes the serialized versions of objects stored in the resources files, and uses these as the parameters to be passed to the function under test. It makes assertions about the returned values from those calls.

2.3 Test Generator

The test generator reads previously stored observations, and generates tests for methods decorated with `@GenerateTestCases`. The test generator is also implemented as a compiler plugin.

The TestGenerator workflow consists of three top level steps:

Traverse code under test: Traverse the compiler's Intermediate Representation (IR) of the method under test, using compiler specific APIs (the specific API depends on whether the TestGenerator is traversing Java and Kotlin). In this traversal, a 'store' is constructed. The store contains static information required to produce the test cases, such as containing class, containing package, method return type, and the imports that the class under test requires (since many of them might be used in the unit test)

Compute information required to generate tests: A TestData-Generator component to TestGen computes test data for each of the store objects collected in the traversal. The TestDataGenerator has multiple implementations, thereby allowing for different test data generation strategies. Currently, we have implemented the observation-based test generation, as described in this paper. However, we purposely left this component generic, so that other strategies could be incorporated (see Section 5).

Write out test cases: The TestWriter component produces fully executable unit tests, written in Kotlin[7]. An example of a single test case generated by TestGen can be found in Figure 3. As can be seen, the format of the test is extremely simple, and human-readable, and the code that expresses the test case uses standard unit testing formats. All generated tests include a 'kill switch', which allows us to globally switch off all generated test cases in case of problems

Placeholders (starting with $) are instantiated using static analysis. The tests assert that, when the test is executed with respect to a previous observation of the same method, in same object state (i.e., current object instance; lines 15–19), and with the same inputs (i.e., parameters; lines 20–27), then the method under test produces the same result (i.e., the same returned value) as the one previously observed (lines 28–36).

When the method under test has no return, the template changes slightly. It will only call the method under test, without adding an assertion. This makes the tests less valuable than the tests that assert something about a returned value, but they still bring value by asserting that the method under test does not crash; essentially falling back on the implicit oracle [9].

2.3.1 Assertion Equality. An important component of the generated tests is the equality assertion between the value returned by the method under test and the value stored as a previous observation. Generally, TestGen cannot assume that it is possible to assert equality between two arbitrary Android objects: this will

---

[7]It would be technically trivial to adapt TestGen to write the test cases it generates in other languages; the choice of Kotlin was a matter of convenience only.

```
1  class <elided test class name> {
2    private val serializer: DASAD =
3      DASAD.getDASAD()
4    private val TESTGEN_RUN_COMMAND =
5      "cd ~<elided path name> && ./<elided script name>.sh" +
6      "$ARGS_USED_BY_TESTGEN_TO_GENERATE_THE_CURRENT_FILE"
7    private val BUCK_INVOCATION_COMMAND =
8      "buck test $BUCK_TARGET_GENERATED_FOR_THIS_TEST"
9
10   @Test
11   fun `test for $METHOD_UNDER_TEST_NAME $TEST_UUID`() {
12     if (DASAD.TESTGEN_KILLSWITCHED) { return; }
13     val subject: $CLASS_UNDER_TEST =
14       DASAD.initializeObservation(
15         "$PATH_TO_CURRENT_OBJECT_INSTANCE_OBSERVATION",
16         javaClass,
17         $CLASS_UNDER_TEST::class.java)
18     val ret =
19       subject.$METHOD_UNDER_TEST(
20         DASAD.initializeObservation(
21           "$PATH_TO_PARAMETER_1_OBSERVATION_IN_RESOURCES",
22           javaClass,
23           $CLASS_OF_PARAMETER_1::class.java),
24         // ... the same pattern for all the parameters of the MUT
25   )
26     assertTrue(
27       DASAD.testGenAssertEqual(
28         "$PATH_TO_RETURN_OBSERVATION",
29         javaClass,
30         ret,
31         serializer,
32         TESTGEN_RUN_COMMAND,
33         BUCK_INVOCATION_COMMAND))
34   }
35 }
36 // End of auto-generated text.
```

Figure 3: An Example generated test class. Some commercially sensitive details have been elided. The test method: 'test for $METHOD_UNDER_TEST_NAME $TEST_UUID' asserts that when called with the previously observed parameters on the previously observed object state, the method returns the previously observed value (lines 26–34)

only work when their class happens to override the equality operator (otherwise TestGen would be asserting pointer equality not value equality). To ensure the TestGen tests perform meaningful equality assertions, we implemented a custom equality assertion: `testGenAssertEqual`.

`testGenAssertEqual` takes a serialized observation and the runtime return of the method under test that should be equal in a passing test. It serializes to json the runtime return and compares the two json strings for equality. It considers only common elements for equality (i.e., commonly existing fields in the recursive traversal of the data type), since fields that are initialized in a unit test may differ from those in the E2E run. That is, in the E2E run, many methods other than the one under test will have been called. These methods can, for example, call the setter on a field that is not called in the unit test run. As long as the common fields are equal, and test execution does not crash, then TestGen determines that the equality test has passed.

2.4 Test Publisher

The test publisher is the final high level TestGen component executed by an overall run of the TestGen workflow. It takes, as input, the test source file as produced by the Test Generator, and produces

a diff that contains runnable reliable (i.e., not flaky) tests in CI (i.e., including the needed buck build files). To do this, it employs the following steps:

Buck file: A buck file is a form of 'make' file, that describes how to build a particular component, called a "buck target". The buck file generator automatically detects when (and where in the file system) tests have been generated, and automatically generates corresponding buck files at the right directory locations. The generated buck files implement the necessary dependencies, visibility rules and other buck configuration details required to ensure that the tests can be fully executed.

Tests can be executed from the command line using a `buck test` command. This is the normal way in which engineers execute unit tests on demand; TestGen tests are, after all, just normal tests, albeit machine-generated. TestGen tests will also automatically be selected when a submitted diff contains code covered by the tests.

Buck Dependencies: The challenge in automatically generating a buck target is to be sure that it contains all the required dependencies. The first category of dependencies includes those required by any and all unit tests. Since the tests will directly call the methods on the class under test and use various types imported there (e.g., parameters and return types), we also automatically add all the dependencies of the class under test to the dependencies of the unit test's buck target.

The most challenging dependencies to determine automatically are those required for observation deserialization. These can be arbitrary dependencies used in the recursive traversal of any data type used by the current object instance, returns or parameters. To compute them, we first analyze all the object types used across all observations. For each unique object, we construct a set of unique buck dependencies containing them. We then add all these dependencies to the buck dependencies of the test.

In this way, we ensure that all the required dependencies are included. The dependence computation process is 'liberal'. Therefore, it favors the generation of a correctly executable test, rather than one that might fail due to insufficient dependence information in the buck files. The buck file so-generated may potentially include some extra dependencies (e.g., not all dependencies of the class under test are used in the unit test), but this is relatively harmless. The existing build system often catches and removes any unneeded dependencies using an independent workflow designed to cater for arbitrary buck files.

Test Runner: The Test Publisher will run all the tests in the newly-added buck target and will collect runtime information about them: whether the test passes consistently, coverage data, etc. The test runner deletes tests that are either broken or flaky so that the Test Publisher does not include any tests that do not consistently pass or that are broken. This insures that the signal to engineers from automatically-generated test is of the highest quality.

Test Selection: Test generation can produce many test cases; one per observation. This could lead to a large number of generated tests, each of which is testing the same method with a different input-output pair. The Test Selector component produces a test suite from the available test cases. Currently, we implement a simple greedy test selection algorithm, guided by a single test objective of coverage. For the single-objective (test coverage) test-selection

optimization problem, a greedy algorithm is known to be efficient, and usually not too far from a global optima [43].

Linting: The publisher runs standard Meta CI linters to fix formatting issues and other lint-level issues, such as unused imports. We found that it was easier to fix these using the existing lint tooling, as a post-processor, rather than trying to statically fix them at test generation time. Using the standard linters, which interpose in continuous integration also insures that the diff that is ultimately published, does indeed meet all current linting standards and, therefore, avoids unnecessary spurious lint error messages on the generated diffs.

Diff Submission: Finally, the publisher submits the generated diff into the standard Meta CI system. This is the mechanism by which TestGen's generated tests enter production; they go through the diff review process just like any other unit test.

3 RESULTS

This section describes our results from deploying TestGen at Meta for Instagram. We report 3 different results: TestGen test results when running in CI at a small scale for 6 months; TestGen coverage at large scale from E2E tests; and back testing TestGen against past large regressions captured in launch blocking tasks (LBs); in the rest of these section.

3.1 Running TestGen in Meta CI

In our deployment process we decided to land TestGen cautiously and incrementally, and with full human code review, rather than automatically landing every test that TestGen could generate (which could potentially run to hundreds of thousands of test cases). In this way, we iron out deployment and scaling issues incrementally and avoid bombarding our engineers with a large amount of unfamiliar test signal all at once. While so-doing we also look out for any problems that the deployment of tests at such scale could create for CI and other parts of Meta infra. This section answers the following research question:

Research Question 1: What is the impact of running TestGen tests in Meta CI on the relevant code changes that Meta engineers submitted for the Instagram Android app?

In particular, the RQ establishes baseline statistics for how many tests have landed into Meta production, and how this initial deployment has behaved in production. For example, how many times the tests have been executed, and how many bugs they have found so far in production. Table 1 presents these results. It gives top level statistics on the TestGen tests that we landed so far and their runs in CI since mid 2023.

The number of blocked diffs is a measure of bug-revealing potential. However, it should be noted that these diffs could also have received signal from other testing and build infrastructure. Also, the results include experimental diffs. That is, engineers often put up initial versions of their changes purely in order to stimulate the testing system to report bugs, fully expecting to see such signal.

Therefore, although these results indicate the bug-revealing potential of the technology, they are not necessarily an accurate reflection of the number of bugs that occur in development, but more reflection of the speed at which developers operate, safe in the knowledge they have a reliable infrastructure that reports bugs. As

Landed Tests	Total Runs	Diff Time Failures	Blocked Diffs
518	9,617,349	75,722	5,702

Table 1: Top level TestGen statistics: 518 landed TestGen tests executed a total of 9,617,349 times, failing on 75,722 test executions and detecting 5,702 faults.

the results show, the TestGen tests do run reliably, and do find bugs. This answers RQ1:

Answer to Research Question 1: TestGen has so-far landed 518 distinct unit tests. In the 6 months between July and December 2023, these tests run in Meta CI a total of 9, 617, 349 times. Test-Gen tests failed on 75, 722 of these 9, 617, 349 executions, thereby revealing bugs in 5, 702 code changes that, if landed to production, would have caused regressions.

3.2 TestGen Coverage Achieved by Observations Generated from E2E Test Executions

TestGen relies on observing End-to-End (E2E) executions of the app to collect the observations that it needs in order to be able to auto-generate tests. There are many ways in which we can generate such observations, such as executing the app manually, running Sapienz [4], and running other automated tools that stimulate app execution.

In the Meta CI, one readily available source of end-to-end executions is the execution of Jest E2E tests[8]. This is an interesting source of observations, because it comes from E2E tests. Generating observations in this way implements a form of test 'carving' (see section 4). The unit test observations are carved from End-to-End tests, while the unit test assertions are generated from the return values.

This allows us to investigate a form of 'coverage completeness' for the carving process; the degree to which the coverage achieved by the E2E tests is replicated at the unit test level, by carving the unit test observations from the E2E test executions. Therefore, in RQ2, we investigate the TestGen completeness when generating unit tests for all the code that the Jest E2E runs execute:

Research Question 2: For how many classes that all Jest E2E tests cover, can TestGen produce covering unit tests?

To answer this question, we run all Jest E2E tests that are marked as 'good' (that is, they execute reliably) in CI on the instrumented Instagram Android app. We collect observations from these app executions and generate tests from the observations. We report the number of Kotlin classes for which TestGen generated tests, as a proportion of the total number of Kotlin classes that Jest E2E tests cover.

Overall, we have a total of 5,284 Jest E2E tests for the Android Instagram app. These tests cover a total of 14,621 Kotlin files (this is approximately 40% of the total number of Kotlin files for the

---

[8]https://jestjs.io/

Android app). Out of these tests, 4,361 are marked as 'good' by CI, while 689 marked as broken, and marked as 234 are flaky[9].

We found that, of all 4,361 Jest E2E tests, TestGen automatically generates tests that covered 86% of the 14,621 Kotlin files that all the Jest E2E tests cover (5,284 tests). As a source of TestGen observations, we used the 4,361 tests marked as 'good' by CI on the day we evaluated RQ2. The 86% coverage result is, therefore, a lower bound, since some of these files would be covered by currently broken or flaky tests that had previously executed and were thus included in coverage statistics for Jest E2E test execution.

In particular, we believe that the remaining 14% can be (or already are) covered because:

- TestGen only runs 'good' Jest E2E tests, while the Jest E2E coverage is reported for all. Only 82% of Jest E2E tests are 'good', so TestGen coverage is inherently underestimated.
- TestGen cannot instrument files that execute only in the main Android Dalvik executable (dex file). Loading all TestGen dependencies in the main dex is not currently scalable.
- There are a few remaining (relatively obscure, and infrequently used) Kotlin constructs and compiler corner cases that TestGen does not yet fully support.

At this scale we cannot know the precise distributions of these three root causes, the first of which leads to an underestimate, while the second two of which are root causes for lack of coverage. We believe the biggest cause of the three is highly likely to be the first of the three, suggesting that the lower bound of 86% retained coverage is a highly conservative lower bound. The answer to the RQ2 is thus:

Answer to Research Question 2: When running all 4,361 reliable Jest E2E tests, TestGen produced unit tests for 12,827 files. Thus, a conservative lower bound is that TestGen automatically covers 86% of the files that the end-to-end tests cover.

3.3 Back testing TestGen Against Past Large Regressions

TestGen's main purpose is regression testing. In order to validate our hypothesis that TestGen is effective at preventing regressions that would otherwise impact developer velocity, we back tested TestGen on previous large regressions that landed in master in October and November 2023.

At Meta, once a diff lands in master, it will eventually be pushed to an internal 'alpha' release. The alpha release uses a monitoring system that creates launch blocking tasks (LBs), thereby protecting the next stage ('beta' release), and ultimately the deployment of the new version into the App Store. Thus, until launch-blocking tasks are fixed, the release is blocked. This is why we regard such launch-blocking tasks to be high–impact regressions, worthy of study. We ask the following research question to assess TestGen's performance at protecting the 'alpha' release stage from regressions, and thereby reducing engineering effort, since it is known that shifting left tends to reduce bug fixing effort [5].

---

[9]These numbers concerning good, broken and flaky, are computed by the Meta CI dynamically and thus change daily. They do not necessarily reflect typical numbers for an arbitrary day of execution. However, we report them here for completeness, so that the reader can fully understand the results we report, as of the census date when we executed TestGen to collect results for RQ2.

Tasks	Result	Language	Notes
13	confirmed	Kotlin	Regression caught successfully
2	unsupported	Kotlin	Anonymous objects unsupported
1	unsupported	Kotlin	Private methods unsupported

Table 2: Back testing results: TestGen was able to prevent 13 out of 16 app-launch-blocking issues. Of the three missed, two where not unit-testable, while the other was a bug in a private method.

Research Question 3: What proportion of past high-impact regressions (LB tasks) can TestGen automatically generate a test for that would prevent them?

3.3.1 Back testing Method. For back testing, we identified a pool of LB tasks. These tasks are typically auto-generated to document detected crashes or bugs that will block Instagram's future app launches. Further, because we launch apps at a defined cadence, launch blockers present a real threat to submitting Instagram's latest build to the app store. These LB tasks are usually prioritized based on severity and manually triaged and fixed by engineers. From the pool of launch blocking tasks, a total of 16 tasks, previously manually resolved by engineers with clear code fixes, were randomly selected without consultation with those actively working the TestGen project (Table 3).

We only consider LB tasks caused by Kotlin code because Test-Gen for Java currently remains under development. With the tasks identified, we then generated unit tests covering the impacted files. With these unit tests we can determine whether the test would fail were the fixes not to be implemented, by reverting the fix. Where such generated tests fail, this indicates that the unit tests could have prevented the launch blocking bug from landing into master.

3.3.2 Back testing Results. Table 2 summarizes our results when back testing TestGen on the 16 randomly-selected launch-blocking tasks involving Kotlin files. TestGen was able to prevent 13 (81%) of them. Out of the 3 (19%) for which TestGen was not able to generate a unit test that would have caught the regression:

- 2(13%) were not prevented because they are not unit-testable (they were part of anonymous methods or objects for which TestGen cannot create a unit test because it cannot call them).
- 1 (6%) was not prevented because the method that caused the crash was declared to be a private method.

For the 2 non-unit-testable LB tasks, we could use Testability Transformation [14, 23, 27] to make them unit-testable.

The other remaining uncovered LB task arose due to a bug in a private method. We believe that tests for public methods that call private ones would to catch them, but we had not originally planned to use TestGen to directly test private methods. We decided to not generate tests for private methods, since this would couple the generated tests to implementation details (which is generally a bad practice). However, if we find regressions are prevalent in private methods, and that these are not trapped by tests generated for the public methods that call the private methods, we may revise this position.

The answer to our research question is:

Instagram Product	Error Type	#TestGen Tests	TestGen Caught?
Profile	Illegal State Exception	2	Yes
Stories	Uninitialized Property Access	-	No
Stories	Illegal Argument	1	Yes
Deep Links Infra	Class Cast Exception	1	Yes
Comments	Illegal Argument	1	Yes
Ads	Null Pointer Exception	-	No
Direct	Class Cast Exception	1	Yes
OnDevice Tech	Illegal State Exception	3	Yes
Feed	Null Pointer Exception	1	Yes
Camera	Runtime Exception	1	Yes
Camera	Android native crash	1	Yes
Direct	Illegal State Exception	1	Yes
Design Systems	Resources Not Found	1	Yes
Well Being	Null Pointer Exception	-	No
Camera	No Such Method	1	Yes
Notifications	Null Pointer Exception	1	Yes

Table 3: Launch blocking tasks on which we back tested Test-Gen. All these tasks were caused by crashes.

> **Answer to Research Question 3:** Out of 16 previous high-impact regressions in Kotlin code, captured by launch blocking tasks (LBs), TestGen was able to successfully generate unit tests that would have prevented 81% of them from becoming launch blocking (Table 2).

4 RELATED WORK

TestGen is related to Test Carving [20]. Test carving takes an existing end-to-end test, and carves the unit test from it. As formulated by Elbaum et al. in 2009 [20], carving seeks to carve out a 'pre-state' from the entire state of the system under test, and a 'post state'. A regression test is then formed by re-establishing the pre-state and asserting the post state. Our approach is similar to this, in the sense that the pre-state, for our observation-based tests, consists of the values of the parameters observations of the method under test, together with the current object instance, while the post state is the observation of return values.

TestGen generates unit level tests. As shown by Gross, Fraser and Zeller in 2012 [24], unit test generators may generate false positives because the unit level parameters and return values are not realizable in any system-level execution. They reported that all 181 errors found using a popular random test generation system on five open source systems were false positives. TestGen circumvents this potential false positive issue using an observation-based approach with tests generated from observations of *real* end-to-end executions.

Kampmann and Zeller [30], overcome the false positive issue by carving *parameterized* unit tests, such that the resulting unit test can be 'lifted' back to a system level test. This allows possible false positive test failures to be detected, by checking whether the lifted computation raises a system failure or not (unit test failures that do not lift to system test failures are discarded as false positives). Kampmann and Zeller implemented this for C, and showed how it allowed them to extend carving to make it applicable for fuzzing [31].Other authors extended the notion of carving, most recently focusing on carving UI tests to generate API tests [42], where the goal is to

dynamically infer suitable API endpoints, so that UI-based testing can be carved to stimulate the corresponding API endpoints.

TestGen's use of serialized observations is related to previous work on object capture, such as the OCAT system, which is used to capture objects for automated testing [29]. Jaygarl et al. applied OCAT to three open source systems, after integrating it with Randoop [34]. They also mutate the captured observations, although there is no mechanism for avoiding the consequent potential for false positives that this may introduce.

Finally, TestGen requires sophisticated serialization and deserialization. It is, therefore, also related to previous work on serialization and deserialization, such as XStream[10] and JSX[11]. One of the primary contributions of the present paper is the introduction of a scalable, depth-aware, fully-featured serialization and deserialization framework, suitable for capturing the complex object serializations needed by industrial observation-based testing.

One of the other potential applications of observation-based testing, is the use of deserialized objects in human authoring of test cases. As reported recently by Fazzini et al. [22] developers typically go to some considerable length to mock, partly spy, fake, and otherwise build complex objects in order to execute unit tests. The observations from observation-based test generation can be used to circumvent the need for the developer to spend (what is typically considerable) time and effort building such 'test double' objects.

The idea of testing based on observations has been in the literature for over two decades, dating back to the JRapture system for Java, introduced in 2000 [38]. Both TestGen and previous observation-based capture/replay tools share the same orientation towards regression testing, based on captured observations of runtime values. However, while TestGen's observations are serialized high level object instances captured from the instrumentation of source code, JRapture captures low level interactions at bytecode level. Other more recent observation-based testing approaches have also used higher-level observations captured, for example, for automotive systems [41]. However, in this automotive work, the observations are of system-level scenarios, mined from traces of simulations and real behaviors for replay in simulation. That makes these automotive system approaches more like Meta's simulation-based testing approach [2], than its TestGen observation-based approach, for which observations are serialized object values.

TestGen is also related to existing work on generation of tests more generally. There have been a great many techniques proposed for test generation over the past five decades, starting with the early work on random test selection in the mid-1970s [13]. An account of the many and varied techniques for test data generation would require a dedicated paper in its own right. For brevity, we refer the reader to existing comprehensive surveys [8, 15, 33].

TestGen's primary novelty and technical advance, lies in the way it tackles the challenges inherent when collecting observations at scale. Without this, TestGen could not achieve an observation-based test generation technique that operates on apps like Instagram, that consist of many tens of millions of lines of code.

5 FUTURE WORK

TestGen is designed to allow easy incorporation of other test generation strategies. It will be particularly interesting to explore hybrid approaches that extend existing approaches to test generation, such as concolic [15], fuzzing-based [31], random [34] and search-based [28], hybridizing each to additionally incorporate observations.

Future work may also consider observation-based approaches to Metamorphic Testing [17, 36]. While TestGen currently targets regression testing, there is an established link between regression testing and Metamorphic Testing [1], which has also recently been deployed at scale in industry at Meta [1, 12] and at Google [18]. In future work, it may be possible to adapt existing metamorphic relation inference algorithms [39, 44] to apply them to observations. This would allow us to extend TestGen to tackle metamorphic testing, especially where such algorithms already have available implementations that target Java/Kotlin documentation [11], thereby residing within the same programming language domain as TestGen.

We would be excited to partner with the academic community on collaborative projects to extend these techniques to consider observation-based formulations, which we can then collaboratively evaluate at scale, using the TestGen framework in place at Meta.

TestGen could hybridize with other observation-based approaches, such as Observation-based Slicing [10]. Observation-based Slicing may also be helpful to TestGen, since it may help identify suitable boundaries for the introduction of mocks. TestGen does not currently generate mocks, but fortunately the core problem has previously been studied [7, 35] and extensions that do create mocks could be informed by work on decisions about when to mock [37]. Observation-based Slicing, combined with these existing mocking techniques, may additionally provide powerful observation-based dependence analysis, flexible mocking and debugging.

Future work may also consider multi-objective regression test formulations [25] to balance different competing regression test criteria, and may therefore extend test selection to use other selection algorithms. Future work might also draw on invariant inference, such as techniques like Daikon [21] to infer likely invariants, over observations, and use these to detect abnormal behaviors, thereby extending TestGen from regression testing to other forms of testing.

6 CONCLUSION

This paper introduced TestGen, a scalable observation-based unit test carver, developed by Meta's Instagram Product Foundation team, since October 2022, and first deployed in July 2023. So far, TestGen has reported 5,702 bugs, while back testing demonstrated it can trap at least 81% of high-impact regressions that would otherwise become app launch blocking for Instagram.

---

[10]https://x-stream.github.io/
[11]https://facebook.github.io/jsx/

REFERENCES

[1] John Ahlgren, Maria Eugenia Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Maria Lomeli, Erik Meijer, Silvia Sapora, and Justin Spahr-Summers. 2021. Testing Web Enabled Simulation at Scale Using Metamorphic Testing. In *International Conference on Software Engineering (ICSE) Software Engineering in Practice (SEIP) track*. Virtual.

[2] John Ahlgren, Kinga Bojarczuk, Sophia Drossopoulou, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Maria Lomeli, Simon Lucas, Erik Meijer, Steve Omohundro, Rubmary Rojas, Silvia Sapora, Jie M. Zhang, and Norm Zhou. 2021. Facebook's Cyber–Cyber and Cyber–Physical Digital Twins (keynote paper). In *25th International Conference on Evaluation and Assessment in Software Engineering (EASE 2021)*. Virtual. Keynote talk given jointly by Inna Dvortsova and Mark Harman.

[3] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. In *Foundations of Software Engineering (FSE 2024)*. Submitted.

[4] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. 2018. Deploying Search Based Software Engineering with Sapienz at Facebook (keynote paper). In 10th *International Symposium on Search Based Software Engineering (SSBSE 2018)*. Montpellier, France, 3–45. Springer LNCS 11036.

[5] Nadia Alshahwan, Mark Harman, and Alexandru Marginean. 2023. Software Testing Research Challenges: An Industrial Perspective. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST 2023)*. IEEE, 1–10.

[6] Nadia Alshahwan, Mark Harman, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Assured LLM-Based Software Engineering (keynote paper). In $2^{nd.}$ *ICSE workshop on Interoperability and Robustness of Neural Software Engineering (InteNSE)* (Lisbon, Portugal). To appear.

[7] Nadia Alshahwan, Yue Jia, Kiran Lakhotia, Gordon Fraser, David Shuler, and Paolo Tonella. 2010. AUTOMOCK: automated synthesis of a mock environment for test case generation. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[8] Saswat Anand, Antonia Bertolino, Edmund Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Jenny Li, Phil McMinn, and Hong Zhu. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (August 2013), 1978–2001.

[9] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (May 2015), 507–525.

[10] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: Language-Independent Program Slicing. In 22nd *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*. Hong Kong, China, 109–120.

[11] Arianna Blasi, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Antonio Carzaniga. 2021. MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation. *Journal of Systems and Software* 181 (2021), 111041.

[12] Kinga Bojarczuk and Mark Harman. 2022. Testing of and with cyber-cyber digital twins. In 7th *International workshop on metamorphic testing (MET 2022)*. Pittsburgh, PA, USA. Keynote talk given jointly by Kinga Bojarczuk and Mark Harman.

[13] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT – a Formal System for Testing and Debugging Programs by Symbolic Execution. In *International Conference on Reliable Software* (Los Angeles, California). ACM, New York, NY, USA, 234–245.

[14] Cristian Cadar. 2015. Targeted Program Transformations for Symbolic Execution. In *10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)* (Bergamo, Italy). 906–909.

[15] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90.

[16] Cristiano Calcagno, Dino Distefano, and Peter O'Hearn. [n. d.]. Open-sourcing Facebook Infer: Identify bugs before you ship. ([n. d.]). code.facebook.com blog post, 11 June 2015.

[17] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: a review of challenges and opportunities. *ACM Computing Surveys (CSUR)* 51, 1 (January 2018), 4:1–4:27.

[18] Alastair F. Donaldson. 2019. Metamorphic testing of Android graphics drivers. In *Proceedings of the 4th International Workshop on Metamorphic Testing, MET@ICSE 2019, Montreal, QC, Canada, May 26, 2019*, Xiaoyuan Xie, Pak-Lok Poon, and Laura L. Pullum (Eds.). IEEE / ACM, 1.

[19] Inna Dvortsova and Mark Harman. 2022. Automated Testing as Production Simulation: Research Opportunities and Challenges. In 37th *IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*. Michigan, USA. Keynote talk given jointly by Inna Dvortsova and Mark Harman.

[20] Sebastian G. Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Matthew Jorde. 2009. Carving and Replaying Differential Unit Test Cases from System Test Cases. *IEEE Transactions on Software Engineering* 35, 1 (2009), 29–45.

[21] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering* 27, 2 (Feb. 2001), 1–25.

[22] Mattia Fazzini, Chase Choi, Juan Manuel Copia, Gabriel Lee, Yoshiki Kakehi, Alessandra Gorla, and Alessandro Orso. 2022. Use of test doubles in Android testing: An in-depth investigation. In *Proceedings of the 44th International Conference on Software Engineering*. 2266–2278.

[23] Dunwei Gong and Xiangjuan Yao. 2012. Testability transformation based on equivalence of target statements. *Neural Computing and Applications* 21, 8 (2012), 1871–1882.

[24] Florian Gross, Gordon Fraser, and Andreas Zeller. 2012. Search-based system testing: high coverage, no false alarms. In *International Symposium on Software Testing and Analysis (ISSTA 2012)*. 67–77.

[25] Mark Harman. 2011. Making the Case for MORTO: Multi Objective Regression Test Optimization (invited position paper). In 1st *International Workshop on Regression Testing (Regression 2011)*. Berlin, Germany.

[26] Mark Harman. 2022. Scaling Genetic Improvement and Automated Program Repair (keynote paper). In *3rd IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2022, Pittsburgh, PA, USA, May 19, 2022*. IEEE, 1–7. https://doi.org/10.1145/3524459.3527353

[27] Mark Harman, Lin Hu, Robert Mark Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. 2004. Testability Transformation. *IEEE Transactions on Software Engineering* 30, 1 (Jan. 2004), 3–16.

[28] Mark Harman, Yue Jia, and Yuanyuan Zhang. 2015. Achievements, open problems and challenges for search based software testing (keynote Paper). In 8th *IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*. Graz, Austria.

[29] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K Chang. 2010. OCAT: object capture-based automated testing. In *Proceedings of the 19th international symposium on Software testing and analysis*. 159–170.

[30] Alexander Kampmann and Andreas Zeller. 2019. Carving parameterized unit tests. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 248–249.

[31] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2018. The Art, Science, and Engineering of Fuzzing: A Survey. *CoRR* abs/1812.00140 (2018). arXiv:1812.00140 http://arxiv.org/abs/1812.00140

[32] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *International Symposium on Software Testing and Analysis (ISSTA 2016)*. 94–105.

[33] Phil McMinn. 2004. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability* 14, 2 (June 2004), 105–156.

[34] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[35] David Saff and Michael D Ernst. 2004. Mock object creation for test factoring. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 49–51.

[36] Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering* 42, 9 (2016), 805–824.

[37] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. 2017. 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, 402–412.

[38] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. 2000. jRapture: A capture/replay tool for observation-based testing. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software Testing and Analysis (ISSTA 2000)*. 158–167.

[39] Fang-Hsiang Su, Jonathan Bell, Christian Murphy, and Gail Kaiser. 2015. Dynamic inference of likely metamorphic properties to support differential testing. In *2015 IEEE/ACM 10th International Workshop on Automation of Software Test*. IEEE, 55–59.

[40] Shreshth Tuli, Kinga Bojarczuk, Natalija Gucevska, Mark Harman, Xiao-Yu Wang, and Graham Wright. 2023. Simulation-Driven Automated End-to-End Test and Oracle Inference. In *45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 122–133.

[41] Christian Wolschke, Thomas Kuhn, Dieter Rombach, and Peter Liggesmeyer. 2017. Observation based creation of minimal test suites for autonomous vehicles. In *2017 IEEE International symposium on software reliability engineering workshops (ISSREW)*. IEEE, 294–301.

[42] Rahulkrishna Yandrapally, Saurabh Sinha, Rachel Tzoref-Brill, and Ali Mesbah. 2023. Carving UI Tests to Generate API Tests and API Specification. *arXiv preprint arXiv:2305.14692* (2023).

[43] Shin Yoo and Mark Harman. 2012. Regression Testing Minimisation, Selection and Prioritisation: A Survey. *Journal of Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.

[44] Jie Zhang, Junjie Chen, Dan Hao, Yingfei Xiong, Bing Xie, Lu Zhang, and Hong Mei. 2014. Search-based inference of polynomial metamorphic relations. In *ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*, Ivica Crnkovic, Marsha Chechik, and Paul Gruenbacher (Eds.). Vasteras, Sweden, 701–712. https://doi.org/doi:10.1145/2642937.2642994