Generating TCP/UDP Network Data for Automated Unit Test Generation

Andrea Arcuri
Scienta, Norway, and
University of Luxembourg,
Luxembourg

Gordon Fraser
University of Sheffield
Dep. of Computer Science
Sheffield, UK

Juan Pablo Galeotti
Saarland University –
Computer Science
Saarbrücken, Germany

ABSTRACT

Although automated unit test generation techniques can in principle generate test suites that achieve high code coverage, in practice this is often inhibited by the dependence of the code under test on external resources. In particular, a common problem in modern programming languages is posed by code that involves networking (e.g., opening a TCP listening port). In order to generate tests for such code, we describe an approach where we mock (simulate) the networking interfaces of the Java standard library, such that a search-based test generator can treat the network as part of the test input space. This not only has the benefit that it overcomes many limitations of testing networking code (e.g., different tests binding to the same local ports, and deterministic resolution of hostnames and ephemeral ports), it also substantially increases code coverage. An evaluation on 23,886 classes from 110 open source projects, totalling more than 6.6 million lines of Java code, reveals that network access happens in 2,642 classes (11%). Our implementation of the proposed technique as part of the EvoSuite testing tool addresses the networking code contained in 1,672 (63%) of these classes, and leads to an increase of the average line coverage from 29.1% to 50.8%. On a manual selection of 42 Java classes heavily depending on networking, line coverage with EvoSuite more than doubled with the use of network mocking, increasing from 31.8% to 76.6%.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

Keywords

Unit testing, automated test generation, Java, JUnit

1. INTRODUCTION

Unit testing is a common practice in industry with the aim to improve software quality. However, writing effective unit tests is a challenging and tedious task. Automated test generation techniques such as random testing [1,2], dynamic symbolic execution (DSE) [3], search-based software testing (SBST) [4], or hybrid approaches [5], have been developed to address these problems. These approaches generate sequences of function calls with the right data to achieve high code coverage on the classes under test (CUTs).

To evaluate just how well this works in practice, we conducted a large empirical study [6] on the SF110 corpus of classes (100 projects randomly chosen from SourceForge, also referred to as SF100, and the top 10 most downloaded ones), artificial software previously used in the literature, and seven industrial systems, using the EvoSuite unit test generation tool [7]. Although reasonably high code coverage was achieved, these experiments pointed out a severe limitation of current test data generation techniques: environment dependencies. It is not uncommon that code manipulates files, takes inputs from a GUI, opens network connections, etc. In those cases, sequences of function calls on the CUTs are not enough: the environment needs to be taken into account, and environment events (e.g., incoming TCP connections) need to be part of the test data.

To overcome some of these limitations, in previous work [8] we introduced a technique to control environmental dependencies using bytecode instrumentation. For several of the Java API classes involving, for example, the file system (e.g., `java.io`), the CPU clock, console inputs/outputs, and some non-deterministic functions in the JVM, we wrote "mock" classes that are semantically equivalent, operating on a virtual environment (e.g., a virtual clock, and a virtual file system in memory). When a CUT is loaded in a test, our instrumentation automatically replaces all those Java classes operating on the environment (e.g., `java.io.File`) with the corresponding mocks (e.g., `MockFile`). As these mocks are semantically equivalent, the instrumentation is transparent to the CUT. This technique not only makes the tests more stable (e.g., we can control the CPU clock, thus making assertions relating to time deterministic), but also leads to higher code coverage. We measured increases of up to +90% code coverage on some classes when applying this technique. When run on all the 11,219 classes of the SF100 corpus, the code coverage increased from 76.5% to 77.9%, indicating that there are further problems that need to be addressed.

In this paper, we extend our initial work on mocking [8] by implementing a virtual network, used to mock classes in the `java.net` package, like TCP sockets. Each generated test case uses its own virtual network, which is independent from the ones used in the other tests. This has several advantages, as (1) it is possible to run tests in parallel that bind to the same local ports; and (2) it is possible to programmatically control remote resources. Furthermore, we also present search operators to better control how network inputs are generated as test data.

To study the effects of a virtual network on code coverage, we carried out an empirical study on the SF110 corpus [6]. SF110 consists of 6.6 millions of Java code lines involving 23,886 classes coming from 110 different systems. Using a custom security manager to

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786828

```
1 public class Example_UDP_TCP {
2  public String getMessage(int port) throws IOException {
3
4   //defines message to send in UDP broadcast
5   InetAddress addr =
6    InetAddress.getByName("192.168.0.1");
7   String handShake = "HAND_SHAKE";
8   String outMsg = addr+":"+port+":"+handShake;
9   byte[] data = outMsg.getBytes();
10
11  //send the message
12  DatagramPacket packet = new DatagramPacket(
13   data, data.length,
14   InetAddress.getByName("255.255.255.255"),12345);
15  DatagramSocket udpSocket = new DatagramSocket();
16  udpSocket.send(packet);
17
18  //open listening TCP server based on sent out message
19  ServerSocket tcpServer =
20   new ServerSocket(port,1,addr);
21  Socket tcpSocket = tcpServer.accept();
22
23  //read string from incoming TCP connection
24  Scanner in = new Scanner(tcpSocket.getInputStream());
25  String msg = in.nextLine();
26
27  //close all resources
28  in.close();
29  tcpSocket.close();
30  tcpServer.close();
31
32  //check if first line contains the handshake token
33  if( msg.startsWith(handShake)){
34   return msg;
35  } else{
36   throw new IOException("Invalid header: "+msg);
37  }
38 }
39}
```

Figure 1: Example class with UDP and TCP communication.

track network accesses, it turned out that networking is a problem for 2,642 classes (11%). Of those, our current implementation of the presented technique can fully handle the network accesses in 1,672 cases (63%), increasing average line coverage from 29.1% to 50.8%. Furthermore, we carried out more detailed experiments on a manual selection of 42 Java classes, showing that the achieved line coverage more than doubled, increasing from 31.8% to 76.6%.

The main contributions of this paper are:

- Empirical evidence that networking is a common problem, happening in 11% of the classes out of 6.6 millions of Java code lines used in the experiments.
- An extension of the instrumentation-based environment mocking approach [8] to (a) allow mocking of `final` classes, and (b) allow the mocking behavior to be activated and deactivated at runtime.
- A virtual network implementation that can be controlled by a test generator through method calls in the tests.
- An adaptation of the search-operators in the search-based EvoSuite test generation tool that uses runtime observations to improve the selection of operations on the virtual network.
- Empirical evidence that these techniques increase line coverage by more than 20% on average on network-related classes.

2. MOTIVATING EXAMPLES

The Java class listed in Figure 1 presents a non-trivial usage example covering a wide range of network functionalities. The method `getMessage` takes as input an integer value representing a local port. The method first sends a UDP broadcast on the network, with a handshake code and the local port number. Then, it opens a TCP listening server on the same port. When an incoming connection from a remote entity is established, the CUT reads the incoming

```
1 @Test public void test0() throws Throwable {
2  Example_UDP_TCP example_UDP_TCP0 =
3   new Example_UDP_TCP();
4  // Undeclared exception!
5  try {
6   example_UDP_TCP0.getMessage((-115));
7   fail("Expecting exception: IllegalArgumentException");
8  } catch(IllegalArgumentException e) {
9   // Port value out of range: -115
10  }
11}
```

Figure 2: Without virtual network, tests generated by EvoSuite on the class in Figure 1 only result in exceptions for `getMessage`.

```
1  try {
2   Socket socket = serverSocket.accept();
3   InetSocketAddress addr =
4    (InetSocketAddress) socket.getRemoteSocketAddress();
5   if( addr.isUnresolved() || ! isAllowed(addr) ) {
6    System.out.println(
7     "TelnetUI: rejecting connection from: " +
8     addr + " as address is not allowed");
9    socket.close();
10  } else {
```

Figure 3: Code excerpt from `SocketServer` in Vuze.

message, and check if it is valid by looking at the handshake code. If so, the received message is returned to the caller of `getMessage`. Otherwise, an exception is thrown.

To the best of our knowledge, no current unit test generation tool can achieve full coverage on such a CUT, as it requires network communications as test data. For example, previously EvoSuite would not be able to execute the method `getMessage` without leading to a security exception first caused by Line 15, which leads to an attempt to bind to an ephemeral port. Deactivating EvoSuite's security manager would still lead to an exception at that line, due to "maximum number of DatagramSockets reached" when generating several test cases, as the UDP sockets are never closed in this code example. Furthermore, a correct execution also depends on whether the IP address at Line 6 is a valid local address, otherwise opening a TCP server on the address at Line 20 would fail. Figure 2 shows the only test that EvoSuite could previously generate.

2.1 Network Transmissions

Code relying on networking is not only a problem with respect to achieving code coverage. For example, during test generation with an SBST or DSE tool, the method `getMessage` could be called several times to find the right test data, even in the order of hundreds of thousands of times. However, broadcasting hundreds of thousands of messages (Line 16) on a network might not be the smartest idea, especially when an engineer is using EvoSuite on her development machine, which most likely would be connected to a corporate network. This is not a problem in a virtual network.

2.2 Blocking Calls

Consider the call at Line 21 in Figure 1: The code attempts to listen to an incoming TCP connection, which is a blocking call: If no external entity actually tries to connect, the test calling `getMessage` will never return. Although a JUnit test case can be run with a timeout (e.g., five seconds in EvoSuite), each test execution would always use the full amount of time specified in the timeout, making the search for test cases extremely inefficient.

As another example, see the snippet in Figure 3 from the class `SocketServer` in the Vuze program (one of the most used peer-to-peer programs in the world). The `if` statement will never be executed if there is no incoming connection (the `accept` method is blocking). Therefore, for testing purposes, there is the need to create incoming connections as test data for the CUT.

```
1 public class Example_URL {
2  public boolean checkURL() {
3   try {
4    URL url =
5     new URL("http://www.evosuite.org/index.html");
6    URLConnection connection = url.openConnection();
7    Scanner in =
8     new Scanner(connection.getInputStream());
9    String line = in.nextLine();
10   in.close();
11   if(line.contains("<html>")){
12    return true;
13   } else {
14    return false;
15   }
16  } catch (IOException e) {
17   return false;
18 }}}
```

Figure 4: Example of an HTTP read with a URL object.

```
1 @Test public void test0() throws Throwable {
2     Example_URL example_URL0 = new Example_URL();
3     boolean boolean0 = example_URL0.checkURL();
4     assertFalse(boolean0);
5 }
```

Figure 5: Without networking support, tests generated by EvoSuite on the class in Figure 4 will only return false.

```
1 public static final String WHOIS_SERVER =
2  "whois.geektools.com";
3 public static final int WHOIS_SERVER_PORT = 43;
4 //...
5 try {
6     Socket socket = new Socket(WHOIS_SERVER,
          WHOIS_SERVER_PORT);
7     UnsyncBufferedReader unsyncBufferedReader =
8      new UnsyncBufferedReader(
9       new InputStreamReader(socket.getInputStream()));
10    PrintStream out = new PrintStream(
11     socket.getOutputStream());
12    //...
13 catch (Exception e) {
14  throw new WebCacheException(_domain + " " +
15   e.toString());
16 }
```

Figure 6: Code excerpt `WhoisWebCacheItem` in Liferay.

On the other hand, a case of interest for testing might be the behavior of a CUT if no incoming connection is registered; again this is difficult to achieve in a test in an efficient way.

2.3 Networking Data

Directly opening TCP connections or sending UDP packets are not the only way to do networking in Java. For example, one could use HTTP to download the content of a remote resource (e.g., a webpage) by using a URL object, as shown in Figure 4. The branch at Line 11 is not trivial: only if the resource exists and contains the needed data, the predicate will be true. But having two different test cases in the same test suite, one for each branch, would require a change in the remote resource. If the tester has write-permission access to the resource, it could be done, although it might be a bit cumbersome. If not, then there would not be many options. Even without a sandbox activated, previous EvoSuite versions could only generate tests covering the false branch, as shown in Figure 5.

Another interesting case can be seen in Figure 6, which shows a code snippet from the class `WhoisWebCacheItem` in the Liferay project. The CUT connects to a remote host, whose IP address is hardcoded in a final variable. At the time of writing of this paper, it points to a valid public server (the interested reader might want to use a web browser to look at the address *whois.geektools.com*). It would be cumbersome, if possible at all, to deterministically control the behavior of that remote server from within a test (e.g., if one wants to cover the code within the catch block). Furthermore, a generated test that captures the current behavior of the CUT could fail tomorrow if the remote server behaves differently, or more simply if the internet connection of the developer running the test momentarily goes down.

3. BACKGROUND

It is not in the scope of this paper to provide a full description of computer networking. In this section, we provide a brief description of different key concepts used in the paper, as well as background information on test generation. For more details about networking we refer to standard literature on the topic (e.g., [9]).

3.1 Networking Concepts

3.1.1 Network Addresses

A machine on a network is identified with an Internet Protocol (IP) address. The most used version of the protocol is IPv4, in which an address is specified with 32 bits, usually visualized by showing its 4 bytes separated by dots, e.g., 127.0.0.1 for the loopback address.

To communicate with a remote host, one not only needs to know its IP address, but also the *port* on which the remote host is listening for incoming connections. Typically, there are up to $2^{16} = 65{,}536$ different ports, where the first 1024 are reserved for special purposes. For example, a webserver hosting webpages will have a server application listening on port 80, and an ssh connection is usually done on port 22. Binding on a port is unique: no other process/thread can bind on the same port with the same protocol (e.g., UDP/TCP). Even if a port is closed, it is not necessarily going to be available again in the immediate future. The closing/reallocation of ports is handled by the operating system, and, in special cases, it can even take several minutes. Instead of specifying a local port to open, one can also use an *ephemeral* one: the operating system will just use any currently available port that is not already bound.

Instead of specifying an IP address numerically, one can also use the name of the remote server, which is then mapped to an IP address using the Domain Name System (DNS). For example, to connect to *www.evosuite.org* one would first connect to a DNS server, which would return the IP address 143.167.8.56.

To connect to a remote host, a network interface card is needed. Typical connections are through ethernet and WiFi. A computer can have more than one interface, specifying different IP subdomains. When an outbound message is sent, based on its remote IP address, it will be forwarded to the right interface serving that subdomain, which will then send the message to the first hop (e.g., a router) toward the final destination.

3.1.2 UDP/TCP Internet Protocols

The user datagram protocol (UDP) is one of the core elements of the internet. It is a connectionless, simple protocol. It does not guarantee that messages will be delivered to the final destination, and, even if they arrive, their order is not guaranteed either. A message can get lost if, for example, it is forwarded through a router that is overloaded and has no space left in its memory buffer.

In UDP, one sends packets, with five main components: the IP address and port of the remote host (i.e., final destination), the IP address and port of the local interface from which the message is originated from (i.e., the source), and the payload (up to 65k bytes).

The transmission control protocol (TCP) is more sophisticated than UDP: it does guarantee delivery, and the order of messages. This is achieved by, for example, resending packets that are lost, and re-ordering the packets on the destination if they are in the wrong order. Similar to UDP, one needs to specify a remote destination with an IP address and port, and a local interface address and port.

From the point of view of the user, the "connection" with the remote host can be seen as a *stream*, to/from where data can be sent/read. Accurate delivery of TCP compared to UDP comes at the cost of possible time delays, e.g., when lost packets need to be resent. Which protocol to use depends on the application requirements.

3.1.3 HTTP

The hypertext transfer protocol (HTTP) is an application protocol, usually built on top of TCP. It defines a request/response protocol between hosts, usually used to retrieve textual data. The most typical case is the world wide web. For example, if one uses a web browser to view the page *http://www.evosuite.org/evosuite/*, then the browser will open a TCP connection on port 80 on the server *www.evosuite.org*. Once the TCP connection is established, the browser will send a message looking like "GET evosuite". On the same connection, the remote server will send the textual information contained in that web page.

The HTTP protocol is very common. Therefore, many programming languages provide libraries to handle it without the need of dealing with the raw details of the TCP messages. For example, in Java, one can simply use a `URL` object to "get" the content of a remote resource, as was done in Figure 4.

3.2 Test Data Generation

Different techniques have been proposed in order to generate unit tests automatically. Random test generators like RANDOOP [2], AUTOTEST [10] and JCRASHER [11] are able to automatically synthesise method sequences for a given CUT. Dynamic symbolic execution based tools (e.g., [3, 12, 13]) use constraint solvers to generate primitive input data, but rely on heuristic approaches to generate sequences of calls (e.g., [14]). Search-based approaches use meta-heuristic algorithms to optimize sequences of calls [4]. Various tools such as TESTFUL [15] or EVOSUITE [7] implement different flavours of search algorithm.

The techniques presented in this paper are implemented and evaluated in the context of the EVOSUITE unit test generation tool, which combines SBST and DSE, using a Genetic Algorithms (GA). A GA works by mimicking the natural process of evolution: Given a population of individuals, the best individuals are selected (survival of the fittest), and combined with operations such as crossover and mutation to produce offspring. This iterative process continues until all search goals are covered, or the computational resources at hand are exhausted. Then, one of the best individuals is reported as the result of the search.

EVOSUITE applies a GA guiding the evolution of whole test suites towards some selected criterion (e.g., branch coverage). In EVOSUITE, individuals are entire test suites. Starting from an initial population of test suites (this initial population can be seeded or randomly created), EVOSUITE will evolve these whole test suites towards the selected criterion. Some of the criteria implemented currently in EVOSUITE include optimizing branch coverage [7], dataflow coverage [16] and killing the highest number of mutants [17].

Besides combining whole test suites using crossover, single test cases can be mutated with a certain probability. This modification may include (non-deterministically) adding a new statement, removing an existing statement or modifying the argument on an existing statement.

3.3 Environment Mocking

Mocking is an approach to isolate a class from its dependencies by using replacements of classes instead of original ones. In common terminology, a *stub* is a replacement with a fixed (usually default) behavior, while a *mock* not only replaces the original class, it also has some partial behavior (mimicking the intended behavior of the class) that needs to be configured, usually during the preparation of the test execution. In this paper, we use "mocks" as synonym for some common terms like *fakes*, *dummies* or *test doubles*.

There are many industrial frameworks that allow a developer to manually write her own mock classes by specifying which behavior they should have. Some examples for Java include Mockito[1], EasyMock[2] and JMock[3]. Although it is a good practice to isolate a class from its dependencies for unit-testing, doing so also adds an additional problem as the behavior of the mock classes should be kept in sync with the real implementation. None of the aforementioned frameworks can automatically create mock objects, it is the user who is expected to declare how the mock behaves.

Perhaps not surprisingly, most of the research towards automatically creating mock objects has been done to enhance automatic test case generation. For example, DSC+MOCK [18] automatically creates mock objects for interfaces with no implementing concrete class while using the symbolic execution tool DSC. If formal pre and post-conditions are given, it is possible to synthesize more realistic mocks satisfying the method's specification [19]. In turn, these mocks are used as input data for parameterized unit tests. Some specific techniques have been presented to improve code coverage of automatic test generators by mocking interactions with databases [20] and file systems [21]. The Moles framework [22] works together with the PEX [23] automatic test generator, Moles not only allows mocking of classes, it also enables redirecting calls by using code instrumentation. Finally, (although not directly related to automatic test generation) Saff and Ernst [24] created mock objects to replace components with long execution times.

In previous work [8] we extended the EVOSUITE test generator to handle classes with environment dependencies. When a user-class is loaded into memory, it is automatically instrumented to isolate the class from its environment. This is done by redirecting invocations accessing environment-dependant data (such as `System.currentTimeMillis`) and by mocking classes that interact with the environment (e.g., `java.io.File` for handling the file system or `System.in` for the console). In turn, search operators are enabled to allow the GA to update the environment dependencies that were used during the test execution. However, that extension did not handle test-data generation for classes with network accesses, used a simpler instrumentation that was neither applicable to `final` classes nor allowed to control when mocked or regular behavior was needed, nor was any systematic study on the impact of networking for automatically generating tests presented.

4. A VIRTUAL NETWORK FOR TESTING

In order to make it possible for automated test generators to target classes with networking dependencies, we propose a technique that uses mocking to replace the networking libraries of the Java standard library with custom, mocked versions. These replacement classes do not access the real network interfaces, but instead access a simulated, virtual network, the state of which is determined by the test generation tool. The virtual network simulates remote hosts, all the communications between the remote and the local host, hostname resolution, choice of ephemeral ports by the operating system, etc. In this section, we describe the two main components of this solution: First, we look at which parts of the standard library need to be replaced. Second, we describe how a test generation tool can be extended in order to manipulate the state of the virtual network.

[1] http://code.google.com/p/mockito, accessed March 2015
[2] http://www.easymock.org, accessed March 2015
[3] http://jmock.org, accessed March 2015

```
public class MockSocket extends Socket
                implements OverrideMock {
  // ...
  @Override
  public InetAddress getLocalAddress() {
    if(! MockFramework.isEnabled())
      return super.getLocalAddress();
    if (!isBound())
      return MockInetAddress.anyLocalAddress();
    // ...
  }
  // ...
}
```

Figure 7: An override mock for `java.net.Socket`; all methods are overriden and ensure that the virtual network is used.

4.1 Network Mocking

4.1.1 Mocked Classes

In order to make the CUT use the virtual network instead of the real one, we extended the mocking technique presented in [8], which was originally used to mock the file system and the CPU clock. The extension introduces different types of mocks to allow mocking of `final` classes, as well as instrumentation to conditionally activate the mocked behavior, or restore the un-mocked behavior. Using instrumentation does *not* require any changes in the CUT, but by using an instrumenting classloader we can automatically modify the CUT code when it is loaded into memory. Every time a network related class is accessed, we rather load a mocked version that is semantically equivalent but operates on the virtual network. Each program behavior, exhibited when a mock class is used, should be possible also with the original program, provided that the environment is properly set up for the execution. In other words, mocked classes should not introduce infeasible behavior. The state of the virtual network is reset after each test execution to avoid dependencies among tests.

To handle Java programs using network features, we needed to mock the following classes from the `java.net` package:

- InetAddress
- NetworkInterface
- InetSocketAddress
- DatagramSocket
- ServerSocket
- Socket
- SocketImpl
- URL
- URLStreamHandler
- URI

Mocking a class means providing implementations for each of its methods. For example, the method `getByName` in the class `InetAddress` returns the IP address for the given hostname, by making a DNS lookup. In our case, we implemented a method that queries a virtual DNS in memory that, for each request, decides whether to return an IP address in valid form or state that the hostname is invalid.

Each mock class needs to be implemented only once, and has to be part of the test data generation framework. The final user should not be needed to write any mock classes. However, it is essential that the implemented mocks are semantically equivalent to the classes they are mocking. The CUT should not behave differently depending on whether a mock or its original class is used. At the end of day, EvoSuite is used to generate test cases that can be used for debugging, and it would not be helpful for this purpose if the CUT would behave different from expectations.

We distinguish between two different kinds of environment mocks to achieve this, which we call *Override Mocks* and *Static Replacement Mocks*.

4.1.2 Override Mocks

Given a Java API class X to mock (e.g., `java.net.Socket`), an override mock is a class that *extends* X (see Figure 7) and overrides every single of its methods and constructors. All those overriden methods will operate on the virtual network. Every single static method will be replicated in the mock with the same signature (note: in Java the static methods are not overridden). When a CUT is loaded in the JVM, we instrument it (using a Java Agent) in a way that each constructor for X will be replaced by a constructor of its mock with same signature (e.g., `new Socket(...)` is replaced by `new MockSocket(...)`). Every static call will be replaced by a static call of the mock, e.g., each call to `Socket.setSocketImplFactory(...)` is replaced by `MockSocket.setSocketImplFactory(...)`. If the CUT extends X, then it will rather extend the mock, e.g., `class Foo extends Socket` will be replaced by `class Foo extends MockSocket`. When instances of X are needed in the test cases as input for the CUT, we rather use the mock. For example, if the CUT has a method to test `public void foo(Socket input)`, then EvoSuite will only instantiate a `MockSocket` to give as input to the method `foo`.

```
public class MockNetworkInterface
    implements StaticReplacementMock {
  // ...
  public static boolean isLoopback(NetworkInterface ni)
    throws SocketException {
    return VirtualNetwork.getInstance()
      .getNetworkInterfaceState(ni.getName())
      .isLoopback();
  }
  // ...
}
```

Figure 8: A static replacement mock for `java.net.MockNetworkInterface`: For each method and constructor of the class URL there is one static method with the same name and signature, except for the additional first parameter that represents the instance of the mocked object.

4.1.3 Static Replacement Mocks

Unfortunately, it is not always possible to create an override mock for the Java APIs. For example, a *final* class cannot be overridden. Likewise, classes with no public constructors cannot be overridden either. In those cases, we use a static replacement mock. A static replacement mock for a Java API class X will not override it, but rather provides a static method for each method/constructor in X. For each non-static method, the mock will need to provide a static one with same name and, a reference to X as first parameter, followed by all the remaining parameters of the mocked method.

For example, given a method `void foo(int v)`, the mock will need to implement `static void foo(X x, int v)`. Then, when the CUT is loaded, each `x.foo(...)` will be replaced by `MockX.foo(x,...)`. There is the need to pass the reference of x to be able to change its state (e.g., by reflection if there are no appropriate setters) accordingly to the semantics of the mocked method `foo`. For each constructor, the mock will need to implement a static method with same input signature and returning an instance of X. Then, the instrumentation will replace each construction call `new X(...)` with `MockX.X(...)`, which will have signature `static X X(...)`. Finally, static methods in X will be treated like they are treated in an override mock. Note, in the case of a static replacement mock, by definition we will not have cases of CUTs extending it, e.g., `class Foo extends X`. Figure 8 shows the example static mock for class `java.net.NetworkInterface`, which contains a mocked method for each method and constructor. For example, method `isLoopback()` is mocked using `isLoopback(NetworkInterface)`, and accesses the virtual network to determine the status of the simulated device.

```
@Test public void test1() throws Throwable {
    EvoSuiteLocalAddress evoSuiteLocalAddress0 = new
        EvoSuiteLocalAddress("192.168.0.1", 596);
    Example_UDP_TCP example_UDP_TCP0 = new
        Example_UDP_TCP();
    boolean boolean0 = NetworkHandling.sendMessageOnTcp(
        evoSuiteLocalAddress0, "HAND_SHAKE");
    String string0 = example_UDP_TCP0.getMessage(596);
    assertEquals("HAND_SHAKE", string0);
}
```

Figure 9: Test generated by EVOSUITE with networking support on the class in Figure 1:

Technically, each override mock could be rewritten as a static replacement mock. So, why two different kinds of mocks instead of making things simple by just always using the static replacement one? The main reason is that at runtime we cannot instrument Java API classes that have already been loaded in memory by the boot-classloader. Assume the CUT creates an instance x of a Java API class X for which we have defined the mock $MockX$. This instance is given as input to another Java API class Y which is not mocked, like for example by calling `Y.foo(x)`. If $MockX$ is an override mock, then all the calls in $Y.foo$ on x will be done on the virtual network. On the other hand, if $MockX$ is rather a static replacement mock, then all the calls in $Y.foo$ will be on the original version of X, and not the mocked one, as Y cannot be instrumented. In this case, an override mock is a better option.

4.1.4 *Switching between mocked and original behavior*

Another essential feature of our framework is the ability to *rollback* the behavior of a mock to its original unmocked class. This became a major requirement when running EVOSUITE in industry, as automatically generated tests could be run together with the existing manually written ones. This was a typical case in continuous integration servers when build programs like Maven are used, and all tests are run in sequence with a "mvn test" call at each new build. Once the CUT has been instrumented when running an EVOSUITE test, all tests afterwards will use that instrumented version. Depending on how the manual tests are written (e.g., depending on an actual remote host to connect to, like for example a web-service), some manual tests could fail because they would now be running on a virtual network that has not been properly initialized for such tests. Those failing tests would be time consuming false positives, as the user would not know whether these tests fail due to a bug before actually spending time in debugging them. Therefore, in our framework we use a flag: each time we instrument a CUT, we automatically wrap each mock call in an `if` statement. If the flag is true, then the mock is used, otherwise the original class is rather used. The flag is activated before each EVOSUITE test (done a in a `@Before` call), and de-activated afterwards (in an `@After` call).

This approach would handle all instrumentation cases but one: the *extends* of a mock class, e.g., `class Foo extends MockX`. Once the class is loaded, its hierarchy cannot be changed, e.g., rolled back to `class Foo extends X`. Therfore, in an override mock we need to manually implement each single method with its own rollback behavior based on such a flag. This is important when instances of that `Foo` are given as input to Java API classes that cannot be instrumented. See for example Line 6 in Figure 7.

4.2 Network as Test Data

Using a virtual network has many advantages, like avoiding DNS resolutions, and preventing port binding conflicts. However, it also enables the creation of remote servers/resources on the fly to be able to better test the CUTs. For this purpose, we created methods to mod-

```
@Test public void test2() throws Throwable {
    EvoSuiteURL evoSuiteURL0 = new EvoSuiteURL(
      "http://www.evosuite.org/index.html");
    Example_URL example_URL0 = new Example_URL();
    boolean boolean0 = NetworkHandling.
        createRemoteTextFile(evoSuiteURL0, "<html>");
    boolean boolean1 = example_URL0.checkURL();
    assertTrue(boolean1);
}
```

Figure 10: Tests generated by EVOSUITE on the class in Figure 4.

```
@Test public void test3() throws Throwable {
    Example_UDP_TCP example_UDP_TCP0 = new
        Example_UDP_TCP();
    try {
      example_UDP_TCP0.getMessage(0);
      fail("Expecting exception: IOException");
    } catch(IOException e) {
      // Simulated exception on waiting server
    }
}
```

Figure 11: Additional test generated by EVOSUITE with networking support on the class in Figure 1.

ify the virtual network, which is accessed by the mocked networking classes, directly in the test cases. For a search-based tool like EVOSUITE, those methods will be part of the search, like any other method of the CUT. See for example the usage of the EVOSUITE framework class `NetworkHandling` in the automatically generated test case in Figure 9 and Figure 10. There, TCP messages are sent with `sendMessageOnTcp`, whereas remote resources can be created on the virtual network with `createRemoteTextFile`.

However, being able to generate network events as test data is not enough. For example, there is no point in sending a TCP message on a port that the CUT is not listening on, or in creating remote resources that the CUT never tries to access. Even more, there is no point at all in using any method from `NetworkHandling` if the CUT does not do any networking. Generating such test data would be just a waste of time.

To overcome these issues, when tests are generated and evaluated as part of the search (e.g., the generations of the genetic algorithm), we keep track of what the CUT tries to access. If the CUT opens a UDP listening socket on a port X, then we will enable in the search the usage of methods to simulate the sending of UDP packets to port X. Likewise, if the CUT accesses a remote resource via a HTTP URL, then we will enable the search to use the methods to create only those remote resources. This drastically reduces the search space of possible test cases to only those relevant ones.

On the other hand, if no incoming connection is registered in the test case, in our virtual network we can just simulate the throwing of a valid IOException (this is the default behavior if no incoming connection is registered), as shown in Figure 11 at Line 7.

Another key aspect of these helper methods in `NetworkHandling` is that they are asynchronously buffered. For example, consider the case of a CUT listening for an incoming TCP connection (e.g., recall Line 21 in Figure 1). Creating an incoming connection *before* the CUT calls the method `accept` would fail, as there is no listening socket yet. Trying to create an incoming connection *after* `accept` is called would not work either, as `accept` would block the JUnit execution until a connection is accepted. One would have have to create a second, separated thread from which to initiate the TCP connection, and synchronize it properly.

To avoid the complications of handling different threads in a JUnit test, we chose a different approach. Because we have full, complete control over the virtual network, every TCP connection or UDP

```
1@Test public void test2() throws Throwable {
2    EvoSuiteLocalAddress evoSuiteLocalAddress0 = new
         EvoSuiteLocalAddress("192.168.0.1", 26);
3    boolean boolean0 = NetworkHandling.sendMessageOnTcp(
         evoSuiteLocalAddress0, "");
4    Example_UDP_TCP example_UDP_TCP0 = new
         Example_UDP_TCP();
5    // Undeclared exception!
6    try {
7      example_UDP_TCP0.getMessage((int) (byte)26);
8      fail("Expecting NoSuchElementException");
9    } catch(NoSuchElementException e) {
10     // No line found
11   }
12}
```

Figure 12: Additional test generated by EvoSuite with networking support on the class in Figure 1.

```
1@Test public void test1() throws Throwable {
2    EvoSuiteURL evoSuiteURL0 = new EvoSuiteURL(
3      "http://www.evosuite.org/index.html");
4    boolean boolean0 = NetworkHandling.
         createRemoteTextFile(evoSuiteURL0, "");
5    Example_URL example_URL0 = new Example_URL();
6    // Undeclared exception!
7    try {
8      boolean boolean1 = example_URL0.checkURL();
9      fail("Expecting NoSuchElementException");
10   } catch(NoSuchElementException e) {
11     // No line found
12   }
13}
```

Figure 13: Test generated by EvoSuite on the class in Figure 4.

packet sent by the test case is buffered. This allows the creation of network test data before the CUT is executed. When the CUT open a listening port, then we check if in the virtual network there is any buffered incoming connection. If yes, then we immediately established the connection. If not, there is no point in keeping the CUT hanging on a blocking call until the test case timeouts, as no further incoming connection would be possible. Therefore, to save precious time that can be used to evaluate new test cases, we hence simulate an immediate error on the network, by throwing a valid IOException.

4.3 Detecting Faulty Networking Code

Besides generating high coverage test suites, EvoSuite can also identify faults in the CUT, e.g., when undeclared exceptions are thrown [25]. In both examples, Figure 1 and Figure 4, there is the same kind of fault: reading a line from a stream without first checking if such a line exists at all (e.g., with the method `hasNextLine` in the class `Scanner`). EvoSuite can generate input data (in this case, network messages) that manage to crash the CUTs by getting them to throw an undeclared `NoSuchElementException` (see Line 9 in Figure 12 and Line 10 in Figure 13).

5. EMPIRICAL STUDY

Intuitively, using a virtual network for unit testing has many immediate benefits. Whether the virtual network is of benefit in automated test data generation is a more difficult question: Will it be sufficient to achieve full code coverage, or is more research needed to develop new techniques to better choose how to generate the network events? Formally, in this section we want to give an answer to the following research questions:

RQ1: How common is network communication in open-source Java classes?

RQ2: What cases of networking code can our mocking technique handle?

RQ3: How much does coverage improve when using a virtual network?

```
1class ClientHelper {
2  static void send(String command, String server, String
       port) {
3    try {
4      doSend(command, server, port);
5    }
6    catch (IOException e) {
7      e.printStackTrace();
8    }
9  }
10
11 static void doSend(String command,
12                     String server,
13                     String port)
14   throws IOException {
15   Socket socket = new Socket(server,
16     Integer.parseInt(port));
17   OutputStream os = socket.getOutputStream();
18   BufferedOutputStream out =
19     new BufferedOutputStream(os);
20   out.write(command.getBytes());
21   out.write('\r');
22   out.flush();
23   socket.close();
24 }
25}
```

Figure 14: Snippet code from `ClientHelper` in jiprof.

```
1public class Clear {
2  public static void main(String[] args) {
3    ClientHelper.send("clear", args[0], args[1]);
4  }
5}
```

Figure 15: Snippet code from `Clear` in jiprof.

5.1 Case Study

We used the SF110 corpus of classes [6] for evaluation. The SF110 is composed of 100 projects randomly selected from SourceForge, which is one of largest web repositories for open-source software. Furthermore, to also take into account programs that are popular, the SF110 corpus includes the 10 most downloaded software. For example, these include the peer-to-peer bitorrent downloading tool Vuze (formerly called Azureus), and the web-portal Liferay. In total, the SF110 corpus consists of 23,886 Java classes, spanning over 6.6 millions of lines of code.

To carry out more detailed experiments, we also selected 42 Java classes coming from 14 different software projects in the SF110 corpus; the smaller number permits more repetitions, which is helpful for the statistical analysis. Those classes were chosen manually, based on different criteria: We first looked at experiments on the whole SF110 to see which classes ask for network permissions. We then searched for keywords (e.g., URL and Socket) in the source code, and identified interesting classes making use of network features. We tried to have at least one class for each major network component we simulate (e.g., UDP, TCP and HTTP via URL). We avoided classes that, albeit using some network features, were only marginally depending on those. Finally, we only selected a relatively small amount (42 classes) as we had to look at their source code manually, and wanted to present their results in more details. Note, to answer **RQ3**, there would not be much point in using classes without network features.

The selected classes are not independent, but rather parts of larger software projects. Therefore, looking at metrics like lines of code can be misleading when evaluating the complexity of such classes. For example, a CUT might be relatively short and not doing any direct network access. On the other hand, such a class might call other classes that do a lot of networking, and that hence require to generate several network events in the test cases. This was the case for many of the analyzed CUTs; for example, consider the class `ClientHelper` from the project jiprof, listed in Figure 14. To

fully cover such a class, one has to create a remote server listening on the specified TCP port, beside providing the right input value to the `send` method. Now, consider the trivially small class `Clear` from the same SF110 project, listed in Figure 15. This class has practically just one statement that calls `ClientHelper`. Still, beside the need to create a string array with at least two elements, one representing a valid IP address, and the other a valid integer port, one has still to create a remote listening server. Otherwise, when a security manager is used, the method `send` will throw a security exception and not return properly.

5.2 Experiments

To give an answer to our research questions, we ran several experiments with EVOSUITE considering two different configurations: with and without the virtual network (VNET). When EVOSUITE is run without the virtual network (i.e., the "Base" default configuration), we employ a security manager to prevent potentially harmful operations [6].

First, we ran the Base configuration once on all 23,886 classes in SF110, to determine which classes try to access the network. Each search was left running for two minutes. From this run, we determined 2,642 classes using the network by analysing the network permissions they asked for. Then, we ran both configurations on those 2,642 classes. To take the randomness of the algorithms into account, each experiment was repeated five times with different random seeds, for a total of $2{,}642 \times 2 \times 5 = 26{,}420$ runs. Finally, on the manually chosen 42 classes, we ran both configurations 30 times, for a total of $42 \times 2 \times 30 = 2{,}520$ runs. All experiments were run with an HPC cluster [26].

The results of these experiments were analyzed following the guidelines in [27]. In particular, we used a Wilcox-Mann-Whitney U-test to check statistical difference among the two analyzed configurations. Effect sizes were measure with the standardized Vargha-Delaney $\hat{A}_{12}$ statistics. A value $\hat{A}_{12} = 0.5$ means no difference between two compared configurations, whereas $\hat{A}_{12} = 1$ means the first configuration always obtained better results, and the other way round for $\hat{A}_{12} = 0$. In other words, the $\hat{A}_{12}$ statistics is a measure of the probability of a run with the first configuration obtaining higher values (e.g., better coverage) than using the second configuration.

5.3 RQ1: How common is networking?

By using a custom Java security manager, we can track each time a CUT attempts to access the network, because this results in the CUT asking for either a `SocketPermission` or a `NetPermission`. In our previous exploratory experiments [6], in which we monitored *all* the Java permissions, it turned out that network permissions were asked in 30% of the classes in SF110. However, after a more in detail analysis carried out in this paper, we found out that such a high number is misleading.

There are many classes in SF110 that are GUI based (e.g., using Swing components). Widgets extend the abstract class `JComponent`. To draw a widget, its frame window needs to know the size of such a component. To our greatest surprise, a call to `getPreferredSize()` leads to network access in Java! In particular, `getPreferredSize()` leads to a call to `getFontMetrics()`, which ends up in the `sun.font.FcFontConfiguration` class where the method `getFcInfoFile()` uses a `InetAddress` to resolve the local host address. If a security exception is thrown, then `getFcInfoFile()` just defaults to using the loopback address. Note, although we mock `InetAddress`, we cannot instrument Java API classes that are loaded by the boot-classloader.

As resolving the local host does seem relatively harmless, and at any rate we do mock `InetAddress` in the CUTs and all third-party libraries it uses, we hence relaxed the security manager to grant this permission. A new run of experiments with the relaxed security manager identified 2,642 classes accessing the network, i.e., 11% of all classes.

RQ1: *11% of all classes in SF110 access the network.*

5.4 RQ2: What is handled by EvoSuites's mocking?

To determine whether a specific networking interaction was successfully mocked, we can again make use of the security manager: Networking related permissions should only be requested by standard networking code, mocked networking code will not ask permission, and instead uses the virtual network. After running EVOSUITE with the virtual network mocking on the 2,642 classes that originally asked for network related permissions, it turned out that, in 1,672 cases, no network permission was asked. This means that our techniques managed to handle all cases of network access in those classes. In other words, our technique fully handled 63% of all classes in which networking is involved. Note that for the remaining 37% classes there are likely also several networking aspects that are handled, but in each of these classes there was at least one attempt to access the actual network.

Why was 100% not achieved? By manually looking at some of the classes that were not handled, we can identify at least two reasons. First, it turned out that there are further classes in the Java API related to networking, which we did not create mocks for. Beside the `java.net` package, there are networking classes like `ServerSocketChannel` in the `java.nio.channels`, and the whole `java.rmi` package (Remote Method Invocation). In principle, this can be solved by extending the virtual network to also mock all those classes.

Second, mocking has limitations when "final" classes are involved. For example, the class `URL` is final. Mocking that class cannot be simply done by subclassing, and so we need to replace every single of its occurrences with our mock (i.e., we need to use a static replacement mock, recall Section 4.1.3). If the CUT calls a method of one class in the Java API that takes a `URL` as input (an example we saw in the experiment logs was one of the constructors in the Swing class `ImageIcon`), then we cannot make those replacements in classes that have been already loaded by the boot-classloader. Fully solving such issues would require to mock all those Java API classes (e.g., `ImageIcon`) to make them use our virtual network (e.g., by rather using our `MockURL` class). Doing this for the whole Java API would most likely be too complicated and time consuming. However, it would be reasonable to do it for a few selected classes that are widely used in practice.

RQ2: *Mocking `java.net` fully covers 63% of the selected classes, but misses cases involving the RMI and NIO packages.*

5.5 RQ3: How much does code coverage improve?

To measure coverage increase, we focus on line coverage instead of the more commonly used branch coverage, for practical reasons: Most Java bytecode based coverage measurement tools measure branch coverage in terms of the outcomes of conditional statements, without considering coverage on non-branching code; that is, in practice, branch coverage does not subsume statement coverage [28]. As many operations on the network results in either blocking method calls (e.g., waiting for an incoming connection) or calls that throw

Table 1: Results of the experiments on the whole SF110.

CUTs using network:	2,642
CUTs handled by VNET:	1,672
Coverage for Base:	29.1%
Coverage for VNET:	50.8%
# Better:	917
# Equal:	515
# Worse:	240
Paired U-test p-value:	$\leq$ 0.001

exceptions (e.g., trying to connect to a non-existing remote host), we argue that line coverage would be a better measure to evaluate improvements on the handling of network related CUTs.

Table 1 shows the results on the SF110 case study. On the 1,672 classes for which the virtual network was successfully used, average line coverage increased from 29.1% (Base) to 50.8% (VNET): a +21.7% improvement. There are 515 classes in which VNET brought no improvement: likely these are cases in which network access happened when generating input parameters to the CUT, but those inputs were not needed to achieve higher coverage. Improvements are obtained on 917 classes, whereas there are 240 in which VNET led to worse results.

A paired U-test on the average line coverage for each class (i.e., 1,672 pairs) resulted in a very low p-value (< 0.01). In other words, there is strong statistical evidence that VNET leads to a higher number of classes in which there is improvement compared to the cases in which it has a negative effect. There are at least three complementary hypotheses to explain why using a virtual network led to worse results in 240 cases: (1) randomness of the algorithm, which also applies to the 917 cases with improvements; (2) the network is not really needed to achieve higher coverage (e.g., it is accessed by unnecessary calls on input data to the CUT), and expanding the search to create network events increases the search space with no benefit; (3) bugs in the virtual network. However, it can be misleading to look at classes in isolation, as the average values are only based on five runs.

Table 2 shows the results of our empirical study on the 42 selected classes. As we have 30 runs per class, results are shown and analyzed for each class in isolation. In most cases, line coverage vastly increases, with strong statical significance. On average, it increases from 31.8% to 76.6%: a +44.9% improvement, more than double. The average standarized $\hat{A}_{12}$ is extremely high: 0.97. This means that, even when taking the randomness of the algorithm into account, in 97 out 100 times using a virtual network leads to higher coverage.

The largest coverage increases seem to be related to code that without virtual networking is completely inaccessible: For example, for project `lilith` there are several classes with 0% code coverage, that with virtual network achieve up to 100% line coverage. In these cases, the network is already accessed in the constructor of the classes: The superclass `AbstractServerSocketEventSourceProducer` attempts to listen on a `ServerSocket`, which the security manager prohibits. Even when it is possible to instantiate networking-related classes, much of the code may be inaccessible due to a networking dependency at method entry. An example for this are classes `ClientHelper` and `Clear` from `jiprof` (see Figure 14), both of which have substantial increases of coverage.

However, code that can only be executed with a virtual network is likely to have further networking dependencies beyond class instantiation: For example, once a `SerializingMessageBasedServerSocketEvent-SourceProducer` is instantiated, it requires TCP data on to the server socket it is listening on, which EvoSuite successfully does. Class `BlockingUDPClient` in project `quickserver` mainly consists of methods related to sending and receiving networking data, and although it can be instantiated without virtual network, the achieved coverage doubles once network traffic can be simulated. As another example, class `HttpMonitor` in project `quickserver` reads content from a remote HTTP server, and reports whether the remote machine appears to be active or down, depending on whether the content transmitted via HTTP matches expected content. EVOSUITE created 11 tests on average that explore several different scenarios, including simulated data matching and not matching the expected content, leading to a coverage increase of 23%.

RQ3: *On average, the virtual network increases line coverage by more than 20% on classes dealing with networking. For classes heavily depending on it, we observed an increase of over 40%.*

6. THREATS TO VALIDITY

Threats to *internal* validity come from how the experiments were carried out. All techniques discussed in this paper have been implemented as part of the EvoSuite tool. Although the tool has been intensively tested, no system is guaranteed to be error free. Furthermore, because EvoSuite is based on randomized algorithms, each experiment has been repeated several times (either 5 or 30), and the results have been evaluated with rigorous statistical methods.

Threats to *construct* validity come from what measure we chose to evaluate the success of our techniques. To measure improvements on testing effectiveness, we considered the achieved line coverage. When dealing with methods that either are very likely to throw exceptions, or block the caller until an external event happens (e.g., an incoming TCP connection), we argue that this measure is more revealing than looking at more common measures like branch coverage. However, to get a better picture, there would also be the need to look at measures like mutation testing score. Furthermore, code coverage does not tell us how easy it will be for the final user to understand the generated test cases (which is needed for debugging).

Even if we can numerically quantify the benefits in terms of increased line coverage, it is hard to quantify all the other benefits that a virtual network gives. Such benefits are for example avoiding test cases failing due to opening ports that are already bound by other unrelated processes, or due to a WiFi network being temporarily down for few seconds, etc. Whether those benefits address very common or rare problems is hard to quantify in an objective way.

Threats to *external* validity come from how well the results generalize to other case studies. To reduce this threat, we used the SF110 corpus, which is a statistically valid random sample of 100 projects from SourceForge, plus its 10 most downloaded ones. That corpus is large, consisting of more than 6.6 millions of lines of code. There is hence reasonable expectance that our virtual network might successfully work as well on other open-source software.

7. CONCLUSIONS

Generating unit tests for real-world software is not only a matter of instantiating classes with the right input objects, and having sequences of function calls on them. Real-world software often interacts with their *environment*, as for example the file system, graphical user interfaces, the CPU clock, databases, etc. Environment events are part of what is needed to effectively unit test classes.

In this paper, we have proposed a novel approach to handle network communications (e.g., UDP messages, TCP connections, or host name resolutions). Our approach is based on semantically equivalent mock classes interacting on a virtual network, and on

Table 2: Results of EVOSUITE with (VNET) and without (Base) the virtual network. For each of the 42 Java classes, we report the obtained average line coverage, standardised $\hat{A}_{12}$ effect sizes and p-values of the statistical tests.

Project	Class	Base	VNET	Difference	$\hat{A}_{12}$	p-value
vuze	TranscodePipeStreamSource	0.0%	31.0%	+31.0%	1.00	$\leq$ 0.001
	CLCacheDiscovery	55.5%	70.5%	+15.0%	0.97	$\leq$ 0.001
	NatCheckerServer	23.1%	48.0%	+24.9%	1.00	$\leq$ 0.001
	PRHelpers	56.5%	82.2%	+25.7%	0.96	$\leq$ 0.001
	TRBlockingServer	29.2%	52.4%	+23.2%	0.92	$\leq$ 0.001
	SocketServer	6.5%	26.4%	+20.0%	1.00	$\leq$ 0.001
freemind	EditServer	29.1%	66.9%	+37.8%	0.97	$\leq$ 0.001
liferay	IPDetector	47.5%	80.8%	+33.3%	1.00	$\leq$ 0.001
	WhoisWebCacheItem	33.3%	85.7%	+52.4%	1.00	$\leq$ 0.001
dsachat	CServer	7.1%	48.7%	+41.5%	1.00	$\leq$ 0.001
	ServerMain	66.7%	100.0%	+33.3%	1.00	$\leq$ 0.001
gangup	ServerConnectionListener	15.8%	70.6%	+54.8%	1.00	$\leq$ 0.001
lilith	AbstractServerSocketEventSourceProducer	6.8%	82.5%	+75.6%	1.00	$\leq$ 0.001
	AccessEventProtobufServerSocketEventSourceProducer	0.0%	99.2%	+99.2%	1.00	$\leq$ 0.001
	LoggingEventProtobufServerSocketEventSourceProducer	0.0%	97.5%	+97.5%	1.00	$\leq$ 0.001
	SerializingMessageBasedServerSocketEventSourceProducer	0.0%	98.7%	+98.7%	1.00	$\leq$ 0.001
	LilithXmlMessageLoggingServerSocketEventSourceProducer	0.0%	96.2%	+96.2%	1.00	$\leq$ 0.001
	LilithXmlStreamLoggingServerSocketEventSourceProducer	0.0%	60.0%	+60.0%	1.00	$\leq$ 0.001
	AbstractLogbackServerSocketEventSourceProducer	0.0%	100.0%	+100.0%	1.00	$\leq$ 0.001
	LogbackAccessServerSocketEventSourceProducer	0.0%	100.0%	+100.0%	1.00	$\leq$ 0.001
summa	SolrSearchNode	55.0%	64.6%	+9.5%	0.96	$\leq$ 0.001
jiprof	Clear	50.0%	100.0%	+50.0%	1.00	$\leq$ 0.001
	ClientHelper	15.4%	97.6%	+82.2%	1.00	$\leq$ 0.001
	File	50.0%	99.0%	+49.0%	1.00	$\leq$ 0.001
	Finish	50.0%	100.0%	+50.0%	1.00	$\leq$ 0.001
	Start	50.0%	100.0%	+50.0%	1.00	$\leq$ 0.001
	Stop	50.0%	100.0%	+50.0%	1.00	$\leq$ 0.001
	RemoteController	63.2%	83.1%	+19.8%	0.88	$\leq$ 0.001
hft-bomberman	TestDriver	66.7%	100.0%	+33.3%	1.00	$\leq$ 0.001
	ForwardingObserver	35.6%	40.1%	+4.5%	0.53	0.767
	ClientQuitRunningSessionMsg	37.5%	62.5%	+25.0%	1.00	$\leq$ 0.001
	PlayerLeftMsg	66.7%	100.0%	+33.3%	1.00	$\leq$ 0.001
	BomberServer	24.2%	54.2%	+30.0%	1.00	$\leq$ 0.001
	ClientInfo	7.7%	51.1%	+43.4%	1.00	$\leq$ 0.001
	StopServer	33.3%	91.7%	+58.3%	1.00	$\leq$ 0.001
	ServerMsgReceiver	9.1%	61.3%	+52.3%	1.00	$\leq$ 0.001
io-project	Server	41.0%	45.8%	+4.9%	0.67	0.050
at-robots2-j	Client	42.9%	56.5%	+13.6%	0.82	$\leq$ 0.001
jaw-br	Update	47.1%	52.9%	+5.9%	1.00	$\leq$ 0.001
quickserver	BlockingUDPClient	47.6%	89.1%	+41.5%	1.00	$\leq$ 0.001
	HttpMonitor	65.4%	88.4%	+23.0%	1.00	$\leq$ 0.001
falselight	Services	49.0%	83.4%	+34.4%	1.00	$\leq$ 0.001
Average:		31.8%	76.6%	+44.9%	0.97	

runtime bytecode instrumentation. To the best of our knowledge, no other unit test generator can create network events as test data.

Our experiments suggest that networking is very common in open-source Java software. In 110 projects (the SF110 corpus) consisting of 23,886 Java classes, network accesses happened in 11% of the classes. Our current implementation fully handles 63% of those cases, leading to an average incease in line coverage of 20%. On classes heavily depending on networking, average improvements are in the range of 40%.

The increase in code coverage we observed indicates that our technique is effective at handling networking communications. However, +21.7% coverage on 1,672 of the classes in SF110 translates to an overall increase of coverage by 1.5% on SF110. Thus, there are still several areas for further investigation, like for example:

Handling more interactions: We have not mocked all the Java API classes involved in networking. Our virtual network can be extended to cover the most common remaining cases.

Improving input data: Although we can generate input data coming through a network channel, how to best do it is open to improvements. For example, if a class reads a byte stream from a remote host, and then later on it uses such data to deserialize an object instance, it would be hard to generate valid bytes using traditional methods like dynamic symbolic execution or search-based testing. Most likely there would be the need to do static analyses to catch the occurrence of those cases, and improve the data generation algorithms accordingly to exploit such information.

Enhancing readability: If one wants to obtain test cases that are useful for debugging and regression purposes, there is the need generate *readable* tests. Test case readability is a very important topic that is still largely unexplored in the literature. We can expect that adding environment events in the generated tests will only exacerbate this problem further.

Framework reuse: Our mocked framework could also be used as a standalone library for helping the writing of manual tests, and/or be integrated in other test data generation tools.

All techniques discussed in this paper have been implemented as part of the EVOSUITE test data generation tool, which is freely available to download. To learn more about EVOSUITE, please visit our website at: `http://www.evosuite.org`.

Acknowledgments. The research leading to these results has received funding from ERC grant 290914, EU FP7 grant 295261 (MEALS), the EPSRC project "EXOGEN" (EP/K030353/1), and the National Research Fund, Luxembourg (FNR/P10/03).

8. REFERENCES

[1] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 2, pp. 258–277, 2012.

[2] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, 2007, pp. 75–84.

[3] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Conference on Programming language design and implementation (PLDI)*, 2005, pp. 213–223.

[4] P. Tonella, "Evolutionary testing of classes," in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, 2004, pp. 119–128.

[5] J. P. Galeotti, G. Fraser, and A. Arcuri, "Improving search-based test suite generation with dynamic symbolic execution," in *IEEE Int. Symposium on Software Reliability Engineering (ISSRE)*, 2013.

[6] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, p. 8, 2014.

[7] ——, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[8] A. Arcuri, G. Fraser, and J. P. Galeotti, "Automated unit test generation for classes with environment dependencies," in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. ACM, 2014, pp. 79–90.

[9] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX network programming*. Addison-Wesley Professional, 2004, vol. 1.

[10] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Experimental assessment of random testing for object-oriented software," in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, 2007, pp. 84–94.

[11] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Softw., Pract. Exper.*, vol. 34, no. 11, pp. 1025–1050, 2004.

[12] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ACM Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2005, pp. 263–272.

[13] N. Tillmann and J. N. de Halleux, "Pex — white box test generation for .NET," in *Int. Conference on Tests And Proofs (TAP)*, ser. LNCS, vol. 4966. Springer, 2008, pp. 134 – 253.

[14] S. Thummalapenta, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "MSeqGen: Object-oriented unit-test generation via mining source code," in *ACM Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2009, pp. 193–202.

[15] L. Baresi, P. L. Lanzi, and M. Miraz, "TestFul: an evolutionary test approach for Java," in *IEEE Int. Conference on Software Testing, Verification and Validation (ICST)*, 2010, pp. 185–194.

[16] M. Vivanti, A. Mis, A. Gorla, and G. Fraser, "Search-based data-flow test generation," in *IEEE Int. Symposium on Software Reliability Engineering (ISSRE)*, 2013, pp. 370–379.

[17] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites." *Empirical Software Engineering (EMSE)*, 2014.

[18] M. Islam and C. Csallner, "Dsc+mock: A test case + mock class generator in support of coding against interfaces," in *International Workshop on Dynamic Analysis (WODA)*, 2010, pp. 26–31.

[19] S. J. Galler, A. Maller, and F. Wotawa, "Automatically extracting mock object behavior from Design by Contract specification for test data generation," in *Proceedings of the 5th Workshop on Automation of Software Test*, 2010, pp. 43–50.

[20] K. Taneja, Y. Zhang, and T. Xie, "Moda: Automated test generation for database applications via mock objects," in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2010, pp. 289–292.

[21] M. R. Marri, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "An empirical study of testing file-system-dependent software with mock objects," in *Automation of Software Test, 2009. AST'09. ICSE Workshop on*, 2009, pp. 149–153.

[22] N. Tillmann and W. Schulte, "Parameterized unit tests," in *ACM Symposium on the Foundations of Software Engineering (FSE)*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 253–262.

[23] J. de Halleux and N. Tillmann, "Moles: tool-assisted environment isolation with closures," in *Objects, Models, Components, Patterns*. Springer, 2010, pp. 253–270.

[24] 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*, 2004, pp. 49–51.

[25] G. Fraser and A. Arcuri, "1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite," *Empirical Software Engineering (EMSE)*, 2013.

[26] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos, "Management of an academic hpc cluster: The ul experience," in *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*. Bologna, Italy: IEEE, July 2014.

[27] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability (STVR)*, vol. 24, no. 3, pp. 219–250, 2012.

[28] N. Li, X. Meng, J. Offutt, and L. Deng, "Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (experience report)," in *IEEE Int. Symposium on Software Reliability Engineering (ISSRE)*, 2013, pp. 380–389.