This is the authors' version of the paper that has been accepted for publication in the
18th IEEE International Conference on Software Testing, Verification and Validation (ICST 2025)

Towards Cross-Build Differential Testing

1st Jens Dietrich, 2nd Tim White
Victoria University of Wellington
Wellington, New Zealand
{jens.dietrich,tim.white}@vuw.ac.nz

3rd Valerio Terragni
University of Auckland
Auckland, New Zealand
v.terragni@auckland.ac.nz

4th Behnaz Hassanshahi
Oracle Labs Australia
Brisbane, Australia
behnaz.hassanshahi@oracle.com

***Abstract*—Recent concerns about software supply chain security have led to the emergence of different binaries built from the same source code. This will sometimes result in binaries that are not identical and therefore have different cryptographic hashes. The question arises whether those binaries are still equivalent, i.e., whether they have the same behaviour. We explore whether differential testing can be used to provide evidence for non-equivalence.**

We study this for 3,541 pairs of binaries built for the same Maven artifact version, distributed on Maven Central, Google Assured Open Source Software and/or Oracle Build-From-Source. We use EVOSUITE to generate tests for the baseline binary from Maven Central, run these tests against this baseline binary and any available alternately built binaries, and compare the results for consistency. We argue that any differences may indicate variations in program behaviour and could, therefore, be used to detect compromised binaries or failures at runtime.

Although our preliminary experiments did not reveal any compromised builds, our approach successfully identified three build configuration errors that caused changes in runtime behaviour. These findings underscore the potential of our method to uncover subtle build differences, highlighting opportunities for improvement.

***Index Terms*—test generation, differential testing, reproducible builds, software supply chain security, java, evosuite**

I. INTRODUCTION

Software supply chain security has attracted increasing attention recently after a number of high impact attacks such as equifax, log4shell, solarwinds and xz [3], [9], [10], [18]. In general, these fall into two categories – compromising components (such as equifax and log4shell), and compromising processes (such as solarwinds and xz). When build processes are compromised, attackers can inject malicious code into a program, resulting in a vulnerable program built from clean source code. The classic approach to achieve this is to compromise the compiler [24].

Reproducible builds [1], [17] are a common countermeasure. The idea is to perform a second build leading to the same binary. As an adversary is very unlikely to compromise two build environments, this can confirm the integrity of the binary with high certainty. Establishing that the binaries are identical is usually done by means of bitwise comparison, often using cryptographic hashes as proxies. Several organisations have started to provide infrastructure and services to provide such secondary builds at scale, often meeting additional security-related requirements, such as SLSA compliance [2]. Two such products that include support to build Java / Maven artifacts are Google's *Assured Open Source Software (gaoss)*[1], and Oracle's *Build-From-Source (obfs)*[2].

However, there are several challenges with the reproducibility of builds: (i) build environments are difficult to replicate, (ii) locating the source code version (commit, tag or release) associated with a binary release version is not always straightforward and (iii) builds may be non-deterministic [6], [12], [15], [27]. In particular, different versions of compilers may employ different compilation strategies, resulting in different, yet functionally equivalent binaries [6], [21], [27]. In order to address this, bytecode normalisation techniques have recently been proposed [7], [21], [27].

From a security analysis perspective, comparing binaries from alternative builds can help detect potential compromises. If two binaries differ, it may indicate that one has been compromised, potentially by the insertion of a backdoor during the build process. A comparison based on strict binary equality is likely to result in poor precision as differences between binaries can be explained by the variability of compilers and other tools used in the build toolchain. Low precision is known to have a serious impact on the acceptance of program analysis tools by engineers [8], [20]. Using bytecode normalisation techniques such as *jnorm* [21] can help to reduce the false positive rate by identifying *equivalent* binaries. The question arises whether binaries built from the same sources that are not equivalent actually have behavioural differences. A good way to check this is to find or construct test cases to demonstrate such differences.

This is the question we set out to study. We use test case generation with EVOSUITE [11] to synthesise regression tests for a baseline build, and then assess whether these tests behave in the same way on programs that are the result of an alternative build from the same sources.

The paper is organised as follows: in Section II we describe the methodology used in our study. Results are discussed in Section III, followed by a brief overview of related work in Section IV and the conclusion (Section VI). Details on how to access the dataset used and the generated tests are provided in Section V.

[1]https://cloud.google.com/security/products/assured-open-source-software
[2]https://maven.oracle.com/public/

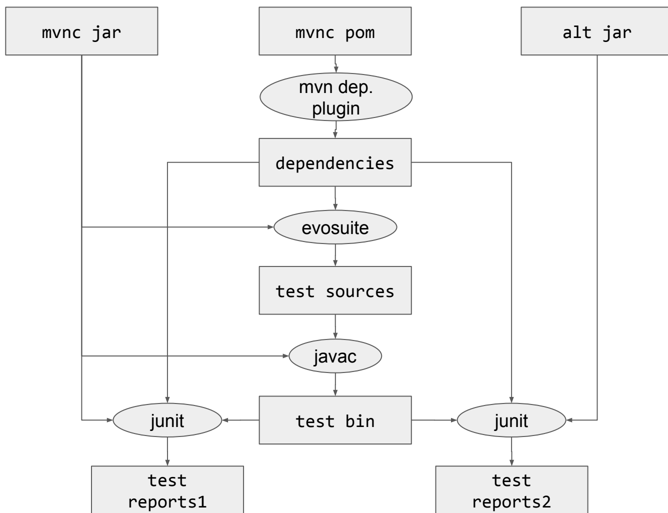

Fig. 1. Methodology Overview

II. Methodology

A. Overview

An overview of our methodology is depicted in Figure 1. As a baseline build we use the artifacts distributed on Maven Central[3] as this is the default artifact repository for most open source Java projects. From there we obtain the binary (jar file containing Java bytecode) *mvnc jar* and the pom *mvnc pom*. From this pom we extract the *dependencies* using the Maven dependency plugin[4]. The *mvnc jar* and its *dependencies* can be used as input to generate *test sources* with EVOSUITE, which are then compiled into test binaries (*test bin*). Finally, these tests can be executed against both the original *mvnc jar* and an alternative jar (*alt jar*) from a different build, with the resulting test reports compared.

B. Project Selection

We started with the dataset from Dietrich et al. [7], which contains jars corresponding to the same artifact identified by the same group id, artifact id and version (i.e., GAV coordinates) built by different parties. This dataset contains the results of builds from Maven Central (*mvnc*), Google (*gaoss*), Oracle (*obfs*) and RedHat. We ignored jars built by RedHat as they have an additional patch id that suggests behavioural differences. We only considered pairs where one of the jars was sourced from Maven Central, i.e., built by the developer(s). There are 3,541 such pairs.

C. Static Pre-analysis

We then ran a static pre-analysis to eliminate pairs that are probably equivalent. We first applied *jnorm* [21][5], and compared jars by comparing the result of the transformation applied to each class; we eliminated pairs with equal *jnorm* output, resulting in 226 pairs of jars. We then applied a filter based on a change in the Java 18 compiler[6], which is not yet supported by *jnorm*. Applying this additional filtering resulted in 166 pairs of jars, 31 of those pairs with an alternatively built jar from *gaoss*, 135 with such a jar from *obfs*. In total there are 1,248 classes that differ between *mvnc* and one of the other two providers, which reduces to 802 after condensing classes to top-level classes.

D. Test Generation

We chose to automatically generate tests for these experiments because the existing tests within the project are typically executed during builds. In other words, alternative builds can ensure that such existing tests pass successfully.

We used EVOSUITE[7] to generate differential tests. Although various alternatives exist for Java test generation, recent benchmarks demonstrate EVOSUITE's superior performance in achieving higher code coverage and mutation scores [13].

EVOSUITE automatically generates test cases for a given object-oriented class under test (CUT). A test case consists of (i) a method call sequence that instantiates and modifies the state of objects of the CUT through method invocations and (ii) one or more assertions that predicate on the values returned by the method invocations. The state of objects often dictates program behaviour, so manipulating these states is crucial for achieving high coverage. It is worth noting that EVOSUITE may need to create and modify objects that do not instantiate the CUT, as the methods of the CUT might require non-primitive parameters. For primitive parameters, EVOSUITE uses randomised values guided by heuristics. The tool implements an evolutionary algorithm that evolves populations of test cases to maximise code coverage, typically targeting branch coverage.

EVOSUITE and similar automated testing tools are most effective when running in regression mode due to the oracle problem. In fact, they are predominantly evaluated in this mode [13], [22]. While the oracle problem is one of the greatest challenges in test automation, it is mitigated in the context of regression testing, where we assume that one version is correct. In regression mode, EVOSUITE adds assertions to the generated test cases that capture the "implemented" behaviour of the given version. These assertions are based on the actual values returned by method call invocations during execution, without regard for the "intended" behaviour of the CUT. This approach also enables effective differential testing by identifying behavioural differences between two implementations (without assuming that one is correct). If a test generated for one version fails when run on another, it reveals a behavioural difference.

EVOSUITE can be set up to generate targeted tests for certain classes. Since the static pre-analysis compares *jar* files by comparing the *.class* files they contain, we can use targeted test generation for those classes only.

EVOSUITE was applied to generate regression tests for the *mvnc* version of each of these 802 differing top-level classes,

[3] https://central.sonatype.com/

[4] https://maven.apache.org/plugins/maven-dependency-plugin/

[5] We used *jnorm* version 1.0.0, downloaded from https://github.com/stschott/jnorm-tool/releases/tag/v1.0.0, with the arguments `-o -n -s -a -p`.

[6] https://github.com/openjdk/jdk/pull/5165

[7] EVOSUITE version 1.2.0, downloaded from https://github.com/EvoSuite/evosuite/releases/tag/v1.2.0

resulting in 476 generated test classes comprising in total 20,437 JUnit4 tests, or 25.5 tests per differing class on average.

We ran EVOSUITE test generation per-class with all settings at their defaults, in particular with `search_budget` (generation time) set to 1 minute.[8] All computation was performed using a JDK 8 toolchain on an 8-core Linux Mint 21.2 VM with 50GB RAM.[9] Resource settings were informed by [13].

III. RESULTS

A. Overview

Table I shows the number of differing test outcomes for each artifact where one or more tests produced a different outcome on an alternative build than on *mvnc*. Note that this excludes 55 test failures that occurred on *mvnc* itself.

We will now discuss the identified inconsistencies in detail.

B. Incorrect Build Configurations

The first kind of test outcome difference, corresponding to rows 4–11 of Table I, occurs 48 times across 7 versions of `io.undertow:undertow-servlet`, and twice "in the other direction" in `io.netty.netty-codec-http:4.1.102.Final`. We first describe the *undertow-servlet* cases.

For each of the 7 versions of the *undertow-servlet* artifact shown in Table I, there are two binaries available: the jar from *mvnc*, and an alternatively built jar from *gaoss*. In each case, when the *gaoss* jar is tested using the `io.undertow.servlet.spec.UpgradeServletInputStream_ESTest`[10] test class generated from the *mvnc* jar, test failures occur, while these tests all succeed against the original *mvnc* jar.

All 48 *gaoss* test failures result from a `java.lang.NoSuchMethodError`, caused by an attempt to invoke `java.nio.ByteBuffer.flip()Ljava/nio/ByteBuffer;`. Failure counts vary across versions only because of random variation in the number of test methods that (indirectly) attempt to call this method. Concretely, for version *2.2.23.Final*, the 5 failing tests are *test00*, *test01*, *test12*, *test15* and *test16*.

The errors occur during linking, and are related to binary compatibility [25]: Two *undertow* methods (`UpgradeServletInputStream.readIntoBufferNonBlocking` and `UpgradeServletInputStream.readIntoBuffer`) have call sites for this method, however, the method is missing. A closer inspection reveals that the *gaoss* version was built with Java 11[11]. In Java 11, `java.nio.ByteBuffer.flip()Ljava/nio/ByteBuffer;` exists[12], and the respective call site is added. The *mvnc* jar was built against Java 8, where this method did not exist, causing the compiler to instead add a call site to invoke `java/nio/ByteBuffer.flip:()Ljava/nio/Buffer;`[13].

Therefore, this issue is caused by the different JDK versions used during the build. This is surprising, as reproducing builds usually try to replicate the build environment in order to maximise the chances of the build to succeed and the resulting binary to be identical to the original binary. In this case the engineers seem to have attempted this somehow by setting the compiler target level to bytecode version 52, which is associated with Java 8 [25, Sect. 4.1] instead of using version 55 associated with Java 11. However, they did not account for the fact that there are changes in the standard library that have an impact on how bytecode is compiled. Since Java 9 (JEP247 [16]), there is a *-release* compiler flag that can be used to avoid issues like this. It is likely that engineers used only *-target* instead. The generated tests reveal those changes.

The two *gaoss* test failures on `io.netty.netty-codec-http:4.1.102.Final` are similar, but "in reverse": Here, it is the *mvnc* jar that was built with a newer JDK version (19), while *gaoss* was built with JDK 8[14]. This time, the generated tests *expect* `NoSuchMethodError` to be thrown, resulting in test failure on *gaoss* when the call to `java/nio/ByteBuffer.flip:()Ljava/nio/Buffer;` completes normally.

Table II summarises these findings. One might expect different behaviour not revealed by tests when tests are executed with Java 11, as the call sites in these two binaries refer to different methods in the standard libraries. However, this is not the case as the Java compiler creates a synthetic *bridge method* when compiling overriding methods with covariant return types. In this case, it creates the bridge method `java.nio.ByteBuffer.flip()Ljava/nio/Buffer;` that invokes `java.nio.ByteBuffer.flip()Ljava/nio/ByteBuffer;`. This bridge method is used during the runtime resolution of the virtual call for the *mvnc* binary.

C. EvoSuite Mock Generation

We have identified three generated test classes with inconsistent behaviour between the *obfs* and *mvnc* builds for `commons-io:commons-io:2.15.0`:

1) `org...io.monitor.FileAlterationObserver_ESTest`
2) `org...io.filefilter.NotFileFilter_ESTest`
3) `org...io.output.UncheckedAppendableImpl_ESTest`

All of these test inconsistencies are caused by the same pattern. The *obfs* binary was built with Java 8, while the *mvnc* binary was built with Java 21. The target level for both was set to 52 (compatible with Java 8). Table III summarises the situation.

In Java 18, a subtle change occurred regarding how methods implemented in the root class (`java.lang.Object`) such as `toString`, `equals` and `hashCode` are invoked for interfaces[15].

[8]Extending generation time to 10 minutes on a subset of classes did not generate any additional tests.

[9]JDK 8 was used in order to improve the reliability of EVOSUITE test generation. Although EVOSUITE could sometimes generate tests under JDK 11—including for 32 CUTs that refused to compile at JDK 8—doing so produced around one third as many tests overall due to widespread failures.

[10]The class under test by an EVOSUITE-generated test can easily be identified by removing the *_ESTest* suffix.

[11]The jar manifest contains the entry *Build-Jdk-Spec: 11*

[12]Though its existence is not mentioned in the documentation at https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/ByteBuffer.html.

[13]Unlike Java source code, Java byte code supports return type overloading, and changing the return types alters the descriptor, and therefore is a binary incompatible change causing linkage errors. [5], [25]

[14]The jar manifests contain the entries *Build-Jdk-Spec: 19* and *Build-Jdk-Spec: 1.8*, respectively.

[15]https://github.com/openjdk/jdk/pull/5165

TABLE I
NUMBER OF TESTS WITH DIFFERENT RESULTS BETWEEN *mvnc* AND ALTERNATIVE BUILDS

build	artifact	test class	tests
gaoss	commons-io:commons-io:2.15.0	org.apache.commons.io.filefilter.NotFileFilter_ESTest	2
gaoss	commons-io:commons-io:2.15.0	org.apache.commons.io.monitor.FileAlterationObserver_ESTest	2
gaoss	commons-io:commons-io:2.15.0	org.apache.commons.io.output.UncheckedAppendableImpl_ESTest	1
gaoss	io.netty:netty-codec-http:4.1.102.Final	io.netty.handler.codec.http.multipart.AbstractDiskHttpData_ESTest	2
gaoss	io.undertow:undertow-servlet:2.2.23.Final	io.undertow.servlet.spec.UpgradeServletInputStream_ESTest	5
gaoss	io.undertow:undertow-servlet:2.2.24.Final	io.undertow.servlet.spec.UpgradeServletInputStream_ESTest	17
gaoss	io.undertow:undertow-servlet:2.2.25.Final	io.undertow.servlet.spec.UpgradeServletInputStream_ESTest	5
gaoss	io.undertow:undertow-servlet:2.2.26.Final	io.undertow.servlet.spec.UpgradeServletInputStream_ESTest	6
gaoss	io.undertow:undertow-servlet:2.2.28.Final	io.undertow.servlet.spec.UpgradeServletInputStream_ESTest	5
gaoss	io.undertow:undertow-servlet:2.2.31.Final	io.undertow.servlet.spec.UpgradeServletInputStream_ESTest	5
gaoss	io.undertow:undertow-servlet:2.2.32.Final	io.undertow.servlet.spec.UpgradeServletInputStream_ESTest	5
obfs	commons-io:commons-io:2.15.0	org.apache.commons.io.filefilter.NotFileFilter_ESTest	2
obfs	commons-io:commons-io:2.15.0	org.apache.commons.io.monitor.FileAlterationObserver_ESTest	2
obfs	commons-io:commons-io:2.15.0	org.apache.commons.io.output.UncheckedAppendableImpl_ESTest	1

TABLE II
TEST RESULTS FOR *UpgradeServletInputStream_ESTest* TESTS ON *io.undertow:undertow-servlet:2.2.23* AND FOR *io.netty.handler.codec.http.multipart.AbstractDiskHttpData_ESTest* TESTS ON *io.netty.netty-codec-http:4.1.102.Final*, RUNNING TESTS ON THE ORIGINAL MAVEN CENTRAL ARTIFACT AND THE ALTERNATIVE BUILD FROM *gaoss*

artifact	build	bytecode version	JDK version used to build	test results (Java 8)	test results (Java 11)
undertow	mvnc	52	8	pass	pass
undertow	gaoss	52	11	fail	pass
netty	mvnc	52	19	pass	fail
netty	gaoss	50	8	fail	fail

TABLE III
TEST RESULTS FOR SELECTED TESTS GENERATED FOR *commons-io:commons-io:2.15.0*, RUNNING TESTS ON THE ORIGINAL MAVEN CENTRAL ARTIFACT AND THE ALTERNATIVE BUILD FROM *obfs*

build	bytecode version	JDK version used to build	test results (Java 8)
mvnc	52	21	pass
obfs	52	8	fail

For instance, consider the code in the following listing:

```
Comparable comp =  ... ;
comp.toString();
```

Up to Java 17, this call was compiled to `invokevirtual Object::toString`[16]. Starting with Java 18, this call is now compiled to `invokeinterface Comparable::toString`.

This results in changes to runtime behaviour when using the EVOSUITE runtime, as EVOSUITE replaces these calls with calls to its own mocks, such as `org.evosuite.runtime.System::toString`. But this only replaces the older call pattern, not the new one.

An example of such inconsistent behaviour we have encountered is when a `toString` implementation is recursive. Then EVOSUITE generates a test with a self-referential data structure that expects a `StackOverFlowError` to occur when `toString` is invoked on this data structure. However, when the mock substitution is performed by EVOSUITE, this error does not occur as the mock `toString` method is not recursive, and the test fails. I.e., the test outcome now depends on whether EVOSUITE performs the substitution or not, and this depends on the compiler that had been used.

In this case we think the misconfiguration is on the Maven Central side, where a very recent Java version has been used to create a binary compatible with Java 8, but this hasn't been configured correctly.

[16]Package names and descriptors are omitted for brevity

IV. RELATED WORK

Differential Testing [19] is an automated approach that identifies various errors by comparing the behaviour of two or more comparable systems [14], [23], [26]. In the context of object-oriented (OO) programming, notable techniques include DIFFUT [26], BERT [14], and DIFFGEN [23]. Similar to our approach, these techniques leverage the differences between two software versions to generate test cases that reveal behavioural discrepancies. However, our approach focuses on comparing binaries, which required us to design and implement specialised ad-hoc analyses.

Although the implementation details may differ, all differential testing techniques share the core idea of generating and executing the same test cases on the versions being compared to reveal behavioural differences. While differential testing has been effective in detecting semantic errors in various domains (e.g., C compilers [28] and JVM implementations [4]), to the best of our knowledge, it has not yet been applied to identify differences between alternative builds of the same program version.

V. DATA ACCESS

The input jar pairs, generated tests and raw test run reports can be found here: https://doi.org/10.5281/zenodo.14753368.

VI. CONCLUSION

In this paper we have explored a novel use case for differential testing: to assess the behavioural equivalence of binaries built independently from the same source code. Our

research was motivated by software supply chain security. While we did not find any evidence that components in our dataset had been compromised during the build, we did find two interesting cases of build misconfiguration.

A possible avenue for future research is to analyse other signals from test executions, namely coverage. This could still reveal differences in program behaviour not captured by the assertions in generated tests.

Acknowledgements

The work of the first two authors was supported by a gift by Oracle Labs Australia.

References

[1] Reproducible builds. https://reproducible-builds.org/.

[2] Supply chain levels for software artifacts (slsa) 1.0. https://slsa.dev/spec/v1.0/#core-specification/.

[3] Executive order 14028, improving the nation's cybersecurity, 2022. https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity.

[4] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed Differential Testing of JVM Implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 85–99, 2016.

[5] Jens Dietrich, Kamil Jezek, and Premek Brada. Broken promises: An empirical study into evolution problems in java programs caused by library upgrades. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 64–73. IEEE, 2014.

[6] Jens Dietrich, Tim White, Mohammad Abdollahpour, Elliott Wen, and Hassanshahi Behnaz. Bineq– a benchmark of compiled java programs to assess alternative builds. In *Proc. SCORED'24*. ACM, 2024.

[7] Jens Dietrich, Tim White, Behnaz Hassanshahi, and Paddy Krishnan. Levels of binary equivalence for the comparison of binaries from alternative builds. *arXiv preprint arXiv:2410.08427*, 2024.

[8] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. Scaling static analyses at facebook. *Communications of the ACM*, 62(8):62–70, 2019.

[9] Robert J Ellison, John B Goodenough, Charles B Weinstock, and Carol Woody. Evaluating and mitigating software supply chain security risks. *Software Engineering Institute, Tech. Rep. CMU/SEI-2010-TN-016*, 2010.

[10] William Enck and Laurie Williams. Top five challenges in software supply chain security: Observations from 30 industry and government organizations. *IEEE Security & Privacy*, 20(2):96–100, 2022.

[11] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[12] Behnaz Hassanshahi, Trong Nhan Mai, Alistair Michael, Benjamin Selwyn-Smith, Sophie Bates, and Padmanabhan Krishnan. Macaron: A logic-based framework for software supply chain security assurance. In *Proc. SCORED'23*, 2023.

[13] Gunel Jahangirova and Valerio Terragni. Sbft tool competition 2023-java test case generation track. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, pages 61–64. IEEE, 2023.

[14] Wei Jin, Alex Orso, and Tao Xie. Automated Behavioral Regression Testing. In *Proceedings of the 3rd IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 137–146, 2010.

[15] Mehdi Keshani, Tudor-Gabriel Velican, Gideon Bot, and Sebastian Proksch. Aroma: Automatic reproduction of maven artifacts. *Proc. FSE'24*, 2024.

[16] Jan Lahoda. JEP 247: Compile for Older Platform Versions, 2014. https://openjdk.org/jeps/247.

[17] Chris Lamb and Stefano Zacchiroli. Reproducible builds: Increasing the integrity of software supply chains. *IEEE Software*, 39(2):62–70, 2021.

[18] Jeferson Martínez and Javier M Durán. Software supply chain attacks, a threat to global cybersecurity: Solarwinds' case study. *International Journal of Safety and Security Engineering*, 11(5):537–545, 2021.

[19] William M McKeeman. Differential Testing for Software. *Digital Technical Journal*, 10(1):100–107, 1998.

[20] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.

[21] Stefan Schott, Serena Elisa Ponta, Wolfram Fischer, Jonas Klauke, and Eric Bodden. Java bytecode normalization for code similarity analysis. 2024.

[22] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 201–211. IEEE, 2015.

[23] Kunal Taneja and Tao Xie. DiffGen: Automated Regression Unit-Test Generation. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 407–410, 2008.

[24] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[25] Gilad Bracha Alex Buckley Daniel Smith Tim Lindholm, Frank Yellin. The Java®Virtual Machine Specification, Java SE 17 Edition, 2021. https://docs.oracle.com/javase/specs/jvms/se17/html/index.html.

[26] Tao Xie, Kunal Taneja, Shreyas Kale, and Darko Marinov. Towards a Framework for Differential Unit Testing of Object-oriented Programs. In *Proceedings of the 2nd International Workshop on Automation of Software Test (AST)*, page 5, 2007.

[27] Jiawen Xiong, Yong Shi, Boyuan Chen, Filipe R Cogo, and Zhen Ming Jiang. Towards build verifiability for java-based systems. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 297–306, 2022.

[28] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, 2011.