BinEq – A Benchmark of Compiled Java Programs to Assess Alternative Builds

Jens Dietrich
jens.dietrich@vuw.ac.nz
Victoria University of Wellington
Wellington, New Zealand

Tim White
tim.white@vuw.ac.nz
Victoria University of Wellington
Wellington, New Zealand

Mohammad Mahdi Abdollahpour
mohammadmahdi.abdollahpour@uwaterloo.ca
University of Waterloo
Waterloo, Canada

Elliott Wen
elliott.wen@auckland.ac.nz
University of Auckland
Auckland, New Zealand

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

Abstract

Incidents like *xz* and *SolarWinds* have led to an increased focus on software supply chain security. A particular concern is the detection and prevention of compromised builds. A common approach is to independently re-build projects, and compare the results. This leads to the availability of different binaries built from the same sources, and raises the question of how to compare the respective binaries (to confirm the integrity of builds, to detect compromised builds, etc). It is however not clear how to do this: naive bitwise comparison is often too strict, and establishing the behavioural equivalence of two binaries is undecidable.

A pragmatic step towards a solution is to provision a benchmark that can be used to test and train equivalence relations. We present such a benchmark for Java bytecode, consisting of 622,029 pairs of binaries (compiled Java classes) labelled as to whether these classes are equivalent or not. We refer to these pairs as equivalence and non-equivalence oracles, respectively.

We derive equivalence oracles from building 56 projects and project versions using 32 dockerised build environments (with different compilers, compiler versions and configurations). Non-equivalence oracles are derived from three different sources: (1) proven breaking API changes, (2) semantic code changes synthesised by means of bytecode mutations, and (3) code changes extracted from vulnerability patches. To illustrate how to use the benchmark, we describe an experiment using two equivalence relations based on locality-sensitive hashing.

CCS Concepts

• **Security and privacy → Software security engineering**; *Vulnerability management*; • **Software and its engineering →** *Software post-development issues.*

SCORED '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1240-1/24/10
https://doi.org/10.1145/3689944.3696162

Keywords

Software Supply Chain Security, Build Security, Reproducible Builds, Java, Maven

ACM Reference Format:
Jens Dietrich, Tim White, Mohammad Mahdi Abdollahpour, Elliott Wen, and Behnaz Hassanshahi. 2024. BinEq – A Benchmark of Compiled Java Programs to Assess Alternative Builds. In *Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3689944.3696162

1 Introduction

Modern software engineering heavily depends on software supply chains. This includes the use of commoditised libraries through managed dependencies and component repositories, and sophisticated automated processes to build, integrate and deploy software. The security of these supply chains has become a major concern [7], highlighted by incidents and vulnerabilities such as *SolarWinds*, *codecov* and *xz* [22, 23, 38].

This has created the need to rethink software supply chains, paying attention to the security of both components and processes. While technologies like software composition analysis [28, 42, 43] and the construction of software bills of materials [10, 14] focus on component security, there is also a need to look at build processes for weaknesses that can be exploited [26]. The possibility of such attacks has long been hypothesised, in particular, in Ken Thompson's famous Turing Award address [50]. In recent years, similar attacks have been observed in the wild, including *XcodeGhost*, *ccleaner*, *shadowpad*, *ShadowHammer*, *SolarWinds* and *xz* [8, 9, 38], and evidence is starting to emerge that compiler-induced vulnerabilities are more common than previously thought [21, 27].

One way to tackle the problem of compromised builds is to aim for reproducible builds [2, 33] or to build redundancy into software supply chains and use consensus protocols to compare independent builds performed by separate parties [1]. The SLSA initiative, which aims at improving build security, mandates builds in a secure hosted environment [4]. Several companies and organisations are now offering such build services and publish independently built components for popular open source packages in dedicated repositories. This has led to a situation where several binaries are available that have been built from the same source code. This raises the question of how such binaries can be compared. In particular, it is of interest to establish equivalence (e.g., to demonstrate that an independent build B confirms build A), and non-equivalence (e.g., to demonstrate that one of two builds may have been compromised).

In general, the equivalence of binaries is a difficult problem. Bitwise comparison is too simplistic as it is sensitive to even the smallest changes caused in the build environment, such as compiler optimisations. What is more, builds might be non-deterministic, and therefore, even building twice on the same platform may result in different binaries. On the other hand, establishing semantic equivalence is generally not feasible as it is undecidable [32].

We believe that there is space for pragmatic solutions between these two options, captured by a formal notion of *binary equivalence* on pairs of programs, together with *classifiers* that evaluate them on actual program pairs given as input. This however requires benchmark datasets to assess the performance of such equivalences in terms of correct classifications. Such benchmarks can then also be used to train classifiers deciding whether two binaries are equivalent or not. In this paper, we present such a benchmark for Java programs, i.e. for Java source code compiled into Java bytecode.

The paper is organised as follows. We discuss the emergence of multiple alternative builds in the background Section 2. We then introduce the core concepts of binary equivalence, equivalence oracle and non-equivalence oracle in Section 3. Section 4 discusses the equivalence oracles, while Section 5 discusses the non-equivalence oracles in detail. We then conduct some experiments with the benchmark to demonstrate its utility by assessing two simple equivalence relations, and report the results in Section 6. This is followed by a discussion of related work in Section 7. We wrap up our contribution in Section 9.

2 Background: Alternative Builds

Open source packages are typically built by developers and released into package repositories. Using the Java ecosystem as an example, developers use build tools like *Maven* or *Gradle* to build projects, and deploy the resulting binaries (*"jars"*) into a Maven repository, usually Maven Central [1]. For other software ecosystems, similar solutions exist. While Maven and Gradle allow deployment builds to be performed by developers on their workstations, it is considered best practice to build on a hosted build platform (a requirement for SLSA level-2 [4]), which in practice is often achieved by using GitHub Actions. However, this remains optional from an open source developers point of view [26].

There are several initiatives by third parties to reproduce builds with improved security and provenance-gathering capabilities, using hosted build environments. The resulting binaries are then stored and managed in curated repositories controlled by the third party. This improves security, and therefore adds value to open source components. This in turn allows vendors to productise those services. Such offerings are now available from Google [2], Oracle [3] and RedHat [4]. In general, these builds try to replicate the platform used in the original build in order to maximise the chances of build success. In the case of Java projects, information that can be used for this purpose includes: (1) The build script itself, such as properties that set language and target bytecode versions. (2) Additional settings used in GitHub Actions or similar scripts used by projects, such as the compiler version or OS being used. (3) Metadata in existing binaries (from Maven Central), such as the `Build-Jdk-Spec` manifest key inserted by the Maven archiver plugin [5].

The use case driving this is to replicate and confirm existing builds. Preferably, a reproduced build will result in the same binary, i.e., a binary that is byte-for-byte identical to the original one (e.g., using binaries from Maven Central as reference), and therefore also has the same cryptographic hash. A different approach is to purposely alter the build environment (e.g., use a different compiler and operating system). A closely reproduced build environment will also make the build exposed to the same vulnerabilities that may have affected the original build. Mutating the build environment on the other hand may reveal a *compromised build*. For instance, consider a build that creates a binary with a backdoor, injected by a compromised compiler. The backdoor consists of an additional call site to a network or system API in a function reachable from the `main` entry point. A second build using a different platform with an alternative compiler that has not been compromised would produce a binary that does not have such a call site, facilitating the detection of the backdoor (with a simple static analysis), and therefore the compromised compiler. In general, comparing binaries built from the same source code using a diverse array of independent software supply chains forces a would-be attacker to compromise *all* such builds to remain undetected – dramatically escalating the cost of such an attack.

The problem with this approach is that different build environments (including compilers) usually produce different binaries, even when the environment has not been compromised. Compilers, and even compiler versions, differ by employing often intricate optimisations, resulting in different binaries. In the case of Java, there have been significant changes in recent years to how some of the most common code is compiled, such as string concatenation (JEP280) [47] and member access by inner classes (JEP181) [46]. Interestingly, even when the same platform is used, builds may not be deterministic. Non-determinism of the Java compiler on the same platform has been reported by Xiong et al. [51]. Compiler non-determinism is seen as not desirable and if discovered is treated as a bug. Recent issues in OpenJDK's *javac* include: *JDK-8264306*, *JDK-8072753*, *JDK-8076031* and *JDK-8295024* [6]. What is more, there are other sources of non-determinism in builds, such as build time code generation. This is widely used to generate parsers from grammars or schemas, using build plugins for *antlr*, *javacc*, *jaxb*, etc. , and for annotation processing, used in popular frameworks like *lombok*, *spring* and *hibernate*.

While there are recent approaches to control sources of non-determinism [51] by normalising bytecode in a post-compilation step (using the proposed *JavaBEPFix* tool), it is unclear how this performs for significant variations (e.g., related to JEP280 and JEP181) and in settings where the full build specification of the build to be reproduced is not known, as it is the case for many binaries on Maven Central. According to [51], the focus of this work is on

[1]https://central.sonatype.com/
[2]https://cloud.google.com/security/products/assured-open-source-software
[3]https://blogs.oracle.com/java/post/announcing-graal-development-kit-437
[4]https://maven.repository.redhat.com/ga/
[5]https://maven.apache.org/shared/maven-archiver/
[6]Issue URLs: *https://bugs.openjdk.org/browse/<issue-id>*

sources of non-determinism that occur when very similar build environments are used, like constant-pool orderings. We also note that the use of invasive tools like *JavaBEPFix* potentially introduces new risks to the build process, as this tool itself could become the target of an attack [7].

Another compelling motivation for such a benchmark is in scenarios where source code needs to be compared to bytecode, such as in patch presence testing [16, 41]. In such cases one does not necessarily know what compilation environment has been used to obtain the bytecode originally, thus needing to deal with compilation-induced differences.

3 Binary Equivalence

3.1 Definition

Establishing whether two binaries b_1 and b_2 are equivalent is a matter of defining an equivalence relation $\simeq\ \subseteq B \times B$ between binaries, and then determining whether $(b_1, b_2) \in\ \simeq$ or not. As is customary we use the notion $b_1 \simeq b_2$ to state that $(b_1, b_2) \in\ \simeq$. Such an equivalence relation is usually expected to satisfy the following three conditions:

(1) $b \simeq b$ for all b - *reflexivity*
(2) $b_1 \simeq b_2 \Rightarrow b_2 \simeq b_1$ for all b_1, b_2 - *symmetry*
(3) $b_1 \simeq b_2 \wedge b_2 \simeq b_3 \Rightarrow b_1 \simeq b_3$ for all b_1, b_2, b_3 - *transitivity*

The baseline is bitwise (for bytecode: byte-by-byte) comparison. In practice, this is not very useful in the context of comparing binaries built from the same sources but different compilers or compiler versions since the output can vary widely [46, 47], and compilers cannot always be assumed to be deterministic as discussed above.

The perfect equivalence is behavioural (semantic) equivalence. However, this can be reduced to the halting problem, meaning it is undecidable [32].

Java binaries are jar files, which are archives containing metadata, resources, and finer-grained, atomic binaries representing compiled classes (*.class* files). Establishing the equivalence of jars can be reduced to establishing the equivalence of their content, using existing mechanisms to compare character-based metadata and resources, and some standard mechanisms for other embedded binary resources such as media files. We therefore focus on the equivalence of *.class* files from here on.

3.2 Equivalence Oracles

While it is difficult or even impossible to capture the full semantics of such an equivalence relation, it is fairly easy to find pairs of binaries (i.e. Java class files in this case) that should be considered equivalent. Such examples include:

(1) Two classes compiled using the same compiler (same *javac* version) from the same sources on the same platform.
(2) Two classes compiled using the same compiler (same *javac* version) from the same sources, one with debug info (`-g`), one without (`-g:none`) [8].
(3) Two classes compiled using different compilers or compiler versions from the same sources.

[7]We contacted the authors of [51] in order to obtain *JavaBEPFix* for evaluation and were informed that the tool is not publicly available.
[8]https://docs.oracle.com/en/java/javase/17/docs/specs/man/javac.html

We refer to such a set of pairs of equivalent classes as an *equivalence oracle*. Given a set of equivalence oracles $eq \subseteq B \times B$ and an equivalence relation $\simeq\ \subseteq B \times B$, we can assess $\simeq$ by checking how many of the records in eq are correctly classified *as being equivalent*.

$$correctness_{eq}(\simeq) := \frac{|eq \cap \simeq|}{|eq|} \quad (1)$$

3.3 Non-Equivalence Oracles

Similarly, there are clear cases where two classes must not be considered equivalent. Examples include:

(1) Two classes with different APIs, such as differences in non-private method names or descriptors, supertypes, declared exceptions, etc.
(2) Two classes with semantic changes applied to one, such as changed arithmetic operators, additional call sites invoking non-synthetic methods, additional precondition checks, etc.
(3) Two classes with changes that have a causal effect on program behaviour, evidenced by the changed outcomes of tests.

We refer to such pairs of classes as *non-equivalence oracles*. Given a set of non-equivalence oracles $neq \subseteq B \times B$ and an equivalence relation $\simeq\ \subseteq B \times B$, we can assess $\simeq$ by checking how many of the records in neq are correctly classified *as not being equivalent*.

$$correctness_{neq}(\simeq) := \frac{|neq \cap \not\simeq|}{|neq|} \quad (2)$$

3.4 The Structure of the Benchmark

The benchmark is organised as a set of *tsv* (tab-separated values) files, each containing records belonging to an oracle. This facilitates the use of the data by many tools, including relational databases and standard statistical software like R. The columns in those files are defined by a simple relational database schema. Each oracle has its own schema, but all share a core schema, i.e., a set of common columns. Oracle-specific additional columns provide additional information. The core schema is depicted in Table 1. Container paths are relative with respect to a root folder containing the distribution of the oracles. Figure 1 shows the folder structure.

Table 1: Core Schema for all Oracles

name	type	description
container_1	path	a folder or jar file containing bytecode
container_2	path	a folder or jar file containing bytecode
class_1	path	the path of a .class file within container_1
class_2	path	the path of a .class file within container_2

```
<root>
 |__ README.md // readme with additional information
 |__ EQ.tsv // data file for EQ oracle
 |__ EQ-schema.tsv // schema for EQ oracle
 |__ ..more data and schema files
 |__ jars/ // compiled code (jar files)
 |__ supplementary/ additional information such as error logs from
     failed builds
```

Figure 1: Folder structure

3.5 The Benchmark API

There is a simple Java API to access the benchmark. The key element is a type `DataSet` with an API to acquire a stream of records. `Record` is a simple Java record type representing a single row in the benchmark as defined by the schema, particular oracles may use a custom subclass to accommodate additional columns.

The stream-based API particularly facilitates two use cases: (1) the definition of views by defining *filters* (i.e., Java stream predicates), similar to SQL views [9]. This will be discussed further in Section 4.6. (2) the expansion of the stream with *flatmap* to infer additional records. This is used to infer records for anonymous inner classes, this is discussed in Section 4.5.

3.6 Downloading and Using the Benchmark

dataset:	https://zenodo.org/records/13381845
api:	https://github.com/binaryeq/bineq-api

4 Equivalence Oracles

We describe the equivalence oracles included in BinEq in this section. There is a single oracle that is materialised (i.e. distributed as a file), but additional equivalence oracles can be extracted from this by means of filters (similar to SQL views) provided as part of the benchmark API, this is described in Section 4.6.

4.1 Compiler Selection

An equality oracle EQ can be constructed by building the same project with different compilers or compiler versions. For this purpose, we set up an environment using *eclipse-temurin* docker images [10] with the respective compilers. We included compilers from four categories defined by vendor, distribution and debug configuration: OpenJDK, OpenJDK-nodebug, Oracle, and ejc. In total, we used 32 different compilers, of which 8 are versions of the Eclipse compiler (*ejc*), 4 are versions on the Oracle JDK compiler, and all other compilers are versions of OpenJDK's *javac*. Among the OpenJDK compilers, 4 use a custom Maven property to modify the debug information that is being created (`-Dmaven.compiler.debug=false`, the default setting being *true* [11]).

We selected compiler versions from major version 8 to 20, with preference for the popular long-term support major versions (8, 11, 17) [12]. While the mapping to versions is explicit for the compilers from the OpenJDK and the Oracle JDKs, it is less obvious for the Eclipse compiler. To establish the correspondence of *ejc* versions to OpenJDK releases, we ran *ejc* with the *-help* option, then inspected the output of *-target* and used the latest version listed there.

The selection of compilers and compiler versions was informed by recent industry surveys on the state of the Java ecosystem [13].

The dataset consists of records of pairs of classes and information about the compilers (name, major/minor/patch version) used to compile each class. We selected compiler combinations according to the following relationship between compilers to create records: (1) Pairs of binaries built by adjacent compiler versions within the same compiler category. Adjacency is defined according to the rules of semantic versioning [3]. (2) Pairs built by corresponding or (if unavailable) similar versions of two compilers from different categories. Figure 2 gives an overview of the compilers used.

4.2 Build Setup

Maven is used to customise building Java projects. This includes the use of a custom Maven dependency cache on the host, and setting custom build properties to remove non-essential build steps, such as testing (*maven.test.skip*), license checks (*rat.skip*), generation of javadocs (*maven.javadoc.skip*), SBOMs (*cyclonedx.skip*) and class version checks (*animal.sniffer.skip*). This also avoids build failures caused by test flakiness [37].

Projects often define a target bytecode version. I.e., even if a later Java version is used to build, projects still compile into some earlier version of bytecode for maximum compatibility. This is usually defined by setting the `maven.compiler.target` property in *pom.xml*. We overrode this property to set the target version to match the major JDK version of the compiler being used in order to maximise the amount of diversity in the benchmark. E.g., when using a Java 17 compiler we attempt to create Java 17 bytecode. While switching to a later compiler is usually unproblematic, using older compilers (e.g., building a project that declares Java 17 as source and/or target Java level with Java 8) will generally fail due to unsupported language features or APIs. We checked a sample of the generated class files with `javap`, verifying that their major version properties were set accordingly.

4.3 Project Selection

Projects were selected from popular open source libraries, mainly Apache libraries. Different versions of these libraries were used to facilitate the construction of the evolution-related oracle NEQ1 (see Section 5.1). These projects are (the number in brackets indicates the number of versions): *commons-io* (6), *commons-lang* (6), *commons-codec* (6), *commons-net* (4), *commons-configuration* (4), *commons-compress* (6), *commons-csv* (5), *commons-bcel* (6), *JSON-java* (5), *jackson-core* (5) and *checkstyle* (3).

We aimed to select projects based on the following criteria:

(1) Popular projects, as evidenced by metrics for the binary distribution (rank in category) and social metrics of the respective source code projects (GitHub fork and star counts). Popularity suggests that this code is representative and similar to code found in other projects.
(2) Projects with permissive open source licenses to facilitate the use as training data for machine learning (such as bytecode classifiers).
(3) Projects from different domains, evidenced by the categorisation used in the Maven repository.
(4) Projects using Maven as build system, in order to facilitate automated builds. This eliminated the Spring Framework, which is built using Gradle.
(5) Projects that are easy to build, in particular mid-sized standalone projects, in order to facilitate automated builds.

[9]Since the benchmark is distributed in a tabular *tsv* format, it is actually possible to access it with SQL by using a database driver for tsv, or importing the files into a relational database
[10]https://hub.docker.com/_/eclipse-temurin
[11]https://maven.apache.org/plugins/maven-compiler-plugin/compile-mojo.html
[12]https://www.oracle.com/java/technologies/java-se-support-roadmap.html
[13]https://www.jetbrains.com/lp/devecosystem-2023/java/, https://newrelic.com/resources/report/2024-state-of-the-java-ecosystem, https://snyk.io/reports/jvm-ecosystem-report-2021/

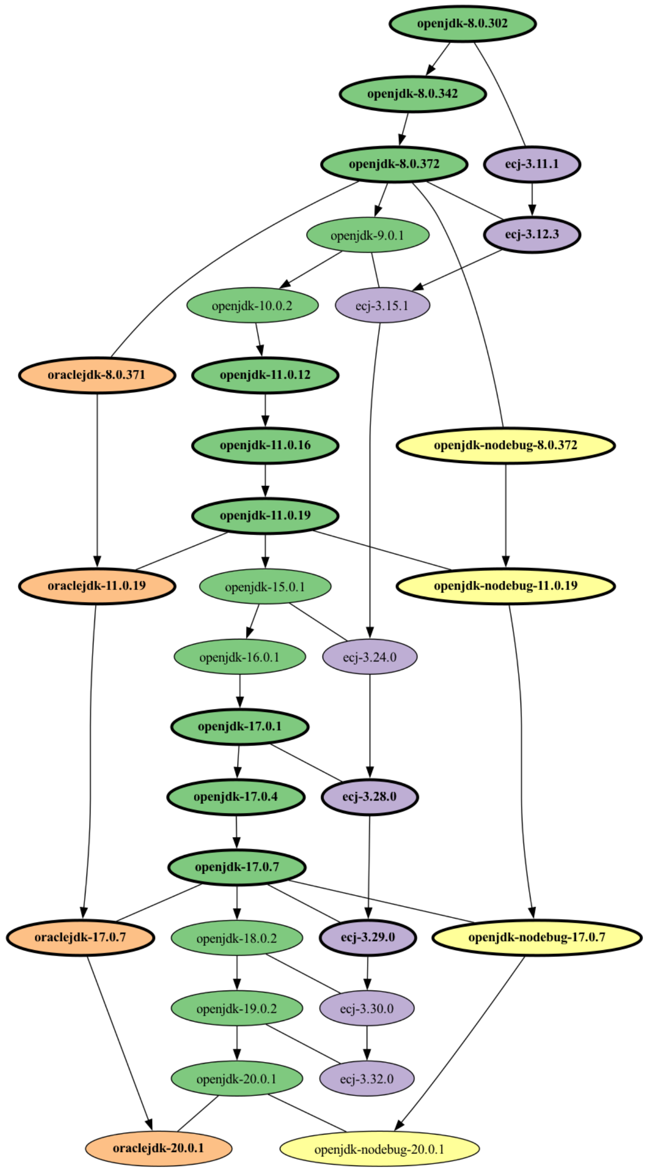

Figure 2: Compilers used to create EQ. Edges indicate the creation of records for the respective pairs. Directed edges are between adjacent versions of the same compiler or compiler configuration. Undirected edges are between different compilers or compiler configurations with identical or similar versions. Compiler versions corresponding to long-term OpenJDK long-term support (LTS) releases are highlighted bold. Similar compilers or compiler configurations are rendered using the same background colour.

(6) Project versions that have small incremental changes to facilitate the creation of dataset records that test whether equivalence relations are sensitive to those changes. This is interpreted as using several patch and minor versions of the same project [3].

Altogether, this resulted in 56 projects, and therefore 1,792 (56×32) attempted builds. Many builds produce a jar file for the main classes as well as a jar for tests. Not all builds succeeded; if they failed, we retained the Maven output. A typical failure reason is that a project uses language features not supported by the compiler used. From successful builds we obtained 1,347 jar files containing classes in the main scope, and 1,077 additional jar files containing test classes.

Table 2 provides an overview of the projects selected, including some of the project meta-data used to justify inclusion.

The docker-based setup to synthesise EQ can be easily extended: compilers and projects used are defined in simple JSON files, and the process to generate the oracle is scripted.

4.4 Reduction

As stated in Section 4.1, we do not consider all pairs of compilers because additional equivalence records can be easily inferred from the transitivity property an equivalence relation has by definition. Given two records that involve three binaries, each built in a different environment, and with EQ records stating $b_1 \simeq b_2$ and $b_2 \simeq b_3$, we can easily infer an additional record stating that $b_1 \simeq b_3$. Given the size of the dataset, the objective of this step becomes clear: there are 32x31 = 992 pairs of different compilers. Assuming that there are 56 binaries from the built projects (this is conservative, since some produce multiple jars) and 100 classes per project (again conservative), we estimate that pre-computing the full transitive closure would produce an EQ benchmark of over 5 million records ($992 \times 56 \times 100$), larger than the actual dataset of 465,858 (see Table 4) by an order of a magnitude.

To further reduce the number of low-value records produced, we also exclude the following type of records:

(1) records containing two identical classes (as these records can be inferred from the reflexivity of the equivalence)
(2) records swapping the two classes being compared in an existing record (as these records can be inferred from the symmetry of the equivalence)

Libraries and algorithms to compute the equivalence closures are readily available and can be used if those additional records are required.

4.5 Anonymous Inner Classes

Anonymous inner classes (AICs) are inner classes without a name and for which only a single object is created. The compiler creates names for those classes by appending a "$" character followed by a counter (1,2,..). AICs are represented by a value representing the number of AICs. E.g., given a class `pck.Foo` with two anonymous inner classes `pck.Foo$1` and `pck.Foo$2`, each record for `pck.Foo` contains an `n_anon_inner_classes_` column with the value set to 2 (the number of AICs in `pck.Foo`). The dataset is then expanded using this counter into three records – one for the outer class, and two for the AOCs. The API has the respective expansion logic (see Section 3.5) applying a functional *flatmap* operation to the record stream. Inferring those records expands the oracle from 465,858 to 518,138 records.

AICs have to be used with caution, and the benchmark contains oracles to ignore them as described in the next Section. The

Table 2: Project selection details. Mvn rank in category data are from https://mvnrepository.com/. Star and fork counts were acquired from the projects' GitHub pages. Ranks, stars and fork counts collected on 25 June 2024. *"c.-"* is short for *"commons-"*

project	versions			mvn rank in category	GitHub metrics		GitHub Project URL
	count	first	last		stars	forks	
c.-io	6	2.7	2.12.0	#1 in I/O Utilities	996	662	https://github.com/apache/commons-io
c.-lang	6	3.8	3.12.0	#2 in Core Utilities	2.7k	1.5k	https://github.com/apache/commons-lang
c.-codec	6	1.11	1.16.0	#1 in Base64 Libraries	449	231	https://github.com/apache/commons-codec
c.-net	4	3.7.1	3.9.0	#1 in FTP Clients and Servers	249	184	https://github.com/apache/commons-net
c.-configuration2	4	2.6	2.9.0	#9 in Configuration Libraries	194	135	https://github.com/apache/commons-configuration
c.-compress	6	1.16	1.23	#1 in Compression Libraries	321	256	https://github.com/apache/commons-compress
c.-csv	5	1.6	1.10.0	#1 in CSV Libraries	372	256	https://github.com/apache/commons-csv
c.-bcel	6	6.4.0	6.7.0	#11 in Bytecode Libraries	239	125	https://github.com/apache/commons-bcel
json.org	5	20211205	20230618	#5 in JSON Libraries	4.5k	2.6k	https://github.com/stleary/JSON-java
jackson-core	5	2.14.2	2.15.2	#3 in JSON Libraries	2.2k	765	https://github.com/FasterXML/jackson-core
checkstyle	3	10.12.2	10.12.4	#2 in Code Analyzers	8.2k	3.6k	https://github.com/checkstyle/checkstyle

problem is that the order of the numbers used to generate names can also depend on the compiler. To study this we used a simple over-approximating method to identify equivalent classes based on computing a feature summary consisting of the superclass, the interfaces and the non-synthetic fields and methods of an AIC. This has revealed some cases where the class numbering clearly differs across compilers. For instance, consider the classes `ZipFile$1` and `ZipFile$2` in *commons-compress-1.16.1.jar* (package names omitted for brevity). Table 3 lists some basic properties of these classes generated by two different compilers.

Table 3: Properties of anonymous inner classes created by different compilers for `org.apache.commons.compress.-archivers.zip.ZipFile` in *commons-compress-1.16.1*

		ecj 3.11.1.v20150902-1521	openjdk 8.0.302
ZipFile$1	superclass	java.lang.Object	java.util.zip.InflaterInputStream
	interface(s)	java.util.Comparator	
	method(s)	compare(ZipArchiveEntry, ZipArchiveEntry)	close()
ZipFile$2	superclass	java.util.zip.InflaterInputStream	java.lang.Object
	interface(s)		java.util.Comparator
	method(s)	close()	compare(ZipArchiveEntry, ZipArchiveEntry)

This suggests that the two compilers use a different numbering scheme, but the classes are otherwise equivalent. We studied the prevalence of this problem using feature summaries as described above. In all cases the compilers were from different groups (e.g., OpenJDK *javac* vs *ecj*), and not just different versions of the same compiler. The derived oracles described in the next section facilitate the exclusion of such records.

4.6 Derived EQ Oracles

Several EQ subsets are of interest to study particular features of equivalence relations. The selection of subsets directly supported by the benchmark is based on the following criteria: the compiler selection (Selection 4.1, visualised in Figure 2) follows two principles (4.2)– to compare (classes compiled from the same source code using) non-OpenJDK compilers with an equivalent OpenJDK version, and to compare adjacent versions of the same compiler. We also include additional views for OpenJDK compilers only (due to its popularity), and for records where both classes have been compiled with the same compiler using the same major version. This is of interest as jar files built with Maven often don't contain details of the compiler that has been used, and only report the major version (via the `Build-Jdk-Spec` manifest entry). Given the issues around AICs it is a reasonable choice to exclude them from the benchmark altogether, at least for some scenarios. For this purpose, we provide *-no-aic* versions for each oracle.

This yields the following 10 equivalence oracles (we consider the OpenJDK *javac nodebug* configuration as a different compiler here):

- EQ – the entire dataset described above
- EQ-no-aic – removes anonymous inner classes (with compiler-generated names) from EQ
- EQ-OpenJDK – only records with both classes compiled with (some version) of the OpenJDK *javac* compiler are considered
- EQ-OpenJDK-no-aic – like EQ-OpenJD, but with anonymous inner classes removed
- EQ-SameComp – both classes have been compiled using the same compiler (but different versions)
- EQ-SameComp-no-aic – like EQ-SameComp, but with anonymous inner classes removed
- EQ-DiffComp – both classes have been compiled using a different compiler
- EQ-DiffComp-no-aic – like EQ-DiffComp, but with anonymous inner classes removed
- EQ-SameMjCompVer – both classes have been compiled using the same compiler, the major versions used are identical
- EQ-SameMjCompVer-no-aic – like EQ-SameMjCompVer, but with anonymous inner classes removed

Those oracles are defined in the API via filters applied to the record stream (similar to SQL SELECT queries) for EQ.

4.7 Build Time and Environment

The dataset is created on an i7-1355U 1.70 GHz laptop with 10 cores and 64 GB, running Linux 5.15.0 inside a VM. Building the EQ jar files took 358 minutes using 3-way parallelization via `make -j 3`; building the EQ.tsv oracle took a further 17 minutes.

4.8 Observations

4.8.1 Summary. Table 4 contains an overview of the EQ oracles with some data about the prevalence of certain features. Column 2 is the size of the oracle. The remaining columns are discussed in the following sections, where we examine a few interesting cases we have encountered.

4.8.2 Different Bytecode Versions in Same Jar. An unexpected insight from our experiments is that jar files may contain *.class* files with different bytecode versions [14]. There is a legitimate use case for this. Consider the project *bcel* [15] that is part of our dataset. The *bcel-6.9.0-tests* official artifact from Maven Central [16] contains classes with different bytecode versions (versions are in format *(major.minor)*), including:

(1) `org.apache.bcel.AbstractTestCase` – version *52.0*
(2) `issue369.Test` – version *46.0*
(3) `issue362.Bcel362` – version *55.0*

The last two classes are used as *test data*. It does make sense to include bytecode compiled with different compilers in order to test compatibility of *bcel*. This means that the respective classes have to be compiled outside the (main) build. A look at the repository reveals the strategy that has been used here: the classes used as test data (`issue*/*`, both sources and compiled) are test resources in `src/test/resources`. I.e., during the build they will be copied into the `target/classes` folder and included in the jar file alongside the classes compiled during the build.

In general, the presence of classes with different bytecode versions is suspicious and could indicate that the build has been compromised and code has been included in the binary to be deployed by copying them into the resource folder. Multiple builds in different environments are more likely to reveal such issues.

4.8.3 Bytecode Size and Complexity. Bcel tests (*bcel-6.9.0-tests*) also contains a class `org.apache.bcel.data.LargeMethod` to be used as test data, compiled during the main build. While the source code is relatively small (ca 50 lines of code), it contains a pattern of deeply nested `try-finally` blocks that need to be expanded by the compiler, resulting in a 6.6 MB class file.

It is reasonable for the *bcel* test library to contain such test data, but in general, similar features might be used to craft denial-of-service attacks on program analyses used in software supply chain security. There are a number of related known patterns in Java, often leveraging data structures similar to billion laughs [6, 19].

4.8.4 Use of Build-time Source Generators. Some binaries contain classes compiled from sources generated during the build. In Maven, the source code of such classes is created in subdirectories of `target/generated-classes`, and these classes can be cross-referenced with binaries. We added columns `generated_by_1` and `generated_by_2` to EQ records for this purpose: for classes built from source files generated at build time, these columns store the name of the subdirectory within `target/generated-classes` they were found in (corresponding to the name of the tool that generated them); for classes present in the original source code, the values for those columns are empty (`-` is used to represent this).

The EQ benchmark contains numerous records for classes compiled from generated code, details can be found in columns 5-7 of Table 4. The code generators we encountered in benchmark programs are the *antlr* and *javacc* parser generators and annotation processors for (test) classes.

We think that it is desirable to have such projects in the benchmark for two reasons. Firstly, code generators are a potential source of non-determinism. While compiler builders try to control this, third party tools could still cause this, even though we have not observed this in the wild. Secondly, those tools could be part of an attack surface and could be used to inject malware into binaries.

4.8.5 Evidence of Non-Determinism. Xiong et al. [51] report non-determinism in bytecode generation. If this non-determinism is interpreted as variation across compiler versions that differ only in minor or patch version levels, then this concurs with our findings, however we were unable to reproduce run-to-run non-determinism in our attempts to build either `docker-maven-plugin:0.36.0` [17] or `kubernetes-client-project:5.4.1` [18].

We ran the build scripts used to compile the projects three times and compared the produced jar files. We encountered differences in the file order within jars, in the `Bnd-LastModified` timestamp in `MANIFEST.MF` and in the timestamp in the `pom.properties` header comment, but no other differences: In particular, we did not encounter any bytecode differences. This suggests that while non-deterministic run-to-run variation in bytecode generation is possible, it is rare in the wild.

4.8.6 JEP181 and JEP280. Throughout the compiler version range we consider there have been some major changes to how bytecode is created, in particular related to member access by inner classes (JEP181) and string concatenation (JEP280). We run a lightweight bytecode analysis on the produced binaries. The results are attached to EQ records using four special columns with boolean values: *bytecode_jep181_1*, *bytecode_jep181_2*, *bytecode_jep280_1* and *bytecode_jep280_2*, respectively.

The EQ benchmark contains numerous records where one class uses one of those bytecode features, while the other does not, details can be found in columns 3 and 4 of Table 4. These records are particularly valuable in order to test the performance of equivalences that can identify the common semantics of the pre- and post-JEP bytecode patterns.

5 Non-Equivalence Oracles

5.1 NEQ1 – Breaking API Changes

Different versions of software may introduce API changes that break clients either at compile time (if the API change is source incompatible), linkage time (if the API change is binary incompatible), or even runtime (when the compiler generates runtime checks). These issues are surprisingly common [18, 29, 45], and static analysis tools exist to locate and categorise these changes [30].

To construct this oracle, we used the *revapi* [19] static analysis tool version 0.28.1 to compare adjacent versions of the same project in the dataset discussed in Section 4.3. Adjacent versions are identified according to the conventions of semantic versioning [3].

[14]Java byte code is generated by a Java compiler. The structure and version of this byte code is formally defined in the JVM Specification [35].
[15]https://github.com/apache/commons-bcel
[16]https://repo1.maven.org/maven2/org/apache/bcel/bcel/6.9.0/bcel-6.9.0-tests.jar

[17]https://github.com/binaryeq/jcompile/issues/32#issuecomment-1806946498
[18]https://github.com/binaryeq/jcompile/issues/34#issuecomment-1809827183
[19]https://revapi.org/

Table 4: Properties of datasets derived from EQ

name	all	JEP181	JEP280	antlr	javacc	annot. proc.
EQ	518,138	10,830	36,827	8,721	745	2,796
EQ-no-aic	465,858	5,474	29,422	8,721	735	2,796
EQ-OpenJDK	129,929	3,377	4,947	971	204	880
EQ-OpenJDK-no-aic	115,549	1,688	3,717	971	198	880
EQ-SameComp	285,705	10,797	14,989	2,913	456	1,772
EQ-SameComp-no-aic	255,665	5,441	11,469	2,913	450	1,772
EQ-SameMjCompVer	42,631	666	0	0	108	276
EQ-SameMjCompVer-no-aic	37,957	377	0	0	108	276
EQ-DiffComp	232,433	33	21,838	5,808	289	1,024
EQ-DiffComp-no-aic	210,193	33	17,953	5,808	285	1,024

Revapi detects API changes such as changes to method signatures, type hierarchies, class names, etc., that may result in compilation (source compatibility) or linking (binary compatibility) issues for client code. *Revapi* detects some basic semantic issues as well, such as issues related to constant inlining. We consider the *revapi* analysis to be precise, i.e., it will not produce false positives [30].

This analysis produces an oracle consisting of 14,384 records related to changes of the respective classes between versions. Of those 308 relate to binary compatibility-breaking changes only, 6,957 to source compatibility-breaking changes only, 7,711 to changes that break both binary and source compatibility, the rest are semantic compatibility-breaking changes.

The NEQ1 schema has extra columns referring to the kind of incompatibility (*source*, *binary* or *semantic*) *revapi* has detected.

5.2 NEQ2 – Mutations

Mutation testing [31] is a technique designed to inject changes into code that change the program semantics, for the purpose of assessing the sensitivity of existing unit, integration or regression tests written against that code. This makes it a suitable technique to generate a non-equivalence oracle.

In order to generate NEQ2, we used an existing mutation testing framework, *pitest-1.15.0* [15]. We customised *pitest*, removing its dependency on testing frameworks, and work in bare-bones mode to transform bytecode into mutated bytecode (i.e., treat *pitest* as a function `byte[]` $\rightarrow$ `byte[]`). We also added an additional verification check based on *asm* [12] to double-check that the modified bytecode is valid. We used all mutators available in *pitest* except those flagged as experimental – 19 mutators in total. We did not attempt to confirm semantic differences by running tests.

The NEQ2 schema extends the core schema by adding columns providing details about the mutations being made, including the type of mutation and the location of the mutated code.

As input for mutations we have used 20 jars from EQ. For each of the 11 projects, the latest project version compiled with *javac-11.0.19* is used as a baseline version; this compiler version was chosen since it is the latest that successfully compiles all 11 such jars. The remaining 9 jars used are the corresponding test jars.

NEQ2 contains 141,585 records. All pass our additional conservative *asm*-based verification. Interestingly, we encountered a rather large number of mutations (222,445) for which verification failed. The respective records were ignored.

5.3 NEQ3 – Vulnerability Patches

NEQ3 consists of security-relevant changes. This is based on another dataset, SAP's *project-kb* [20], described in [44]. Records in this Apache 2.0-licensed dataset consist of a vulnerability id (CVE), a repository URL, a commit id, and a class name. In general, building arbitrary projects (before and after the commit) is challenging to automate [25]; we therefore opted for finding existing binaries of the last version before and the first version after the commit. For this purpose, we identified the GHSA [21] entry corresponding to the CVE; GHSA has references to binaries in the Maven repository. This required the CVE to have a corresponding GHSA record. We only consider reviewed GHSA records that referred to a single *fix commit* and single *fixed artifact* identified by group, artifact, and version (GAV) to obtain a unique reference to a binary in the Maven repository. We obtained 270 CVE records up to this stage.

To obtain the *latest vulnerable version* of each binary, we scanned the `maven-metadata.xml` of the corresponding package and extracted the latest version before the *fixed* version. We used *Maven Central* and *Jenkins CI* as our Maven repository sources. In the next step, we cloned the repository of each package and analysed the modified Java files in the corresponding patch commits. We only kept the files with at least one method modification to remove potential false positives. Since *project-kb* lacks source changes for many records, we extracted patches directly from the repositories.

Finally, we looked up and extracted the modified class files from the corresponding jars. This step is necessary to ensure a match between the identified classes from the previous step and the downloaded jar files, as they stem from different data sources. The commit information is obtained from *project-kb* dataset, and the artifact coordinates (GAVs) are extracted from the *GHSA* dataset. We found that 186 of the modified classes are test classes which are not included in the compiled binaries. In 28 cases, source changes do not change the compiled binary. In several cases the modified class file is in a different jar which is used as a dependency of the vulnerable artifact (e.g., CVE-2019-10093). Several cases are attributed to imprecisions in the GHSA dataset (e.g., CVE-2013-7285).

Overall, this process yielded 202 pairs for classes from 77 different repositories. NEQ3 is based on patches for the vulnerabilities listed in Figure 5. Sometimes there are multiple patches for the same vulnerability.

The NEQ3 schema extends the core schema by adding columns providing details about the CVE, the GHSA id, the latest vulnerable

[20]https://github.com/SAP/project-kb
[21]https://github.com/github/advisory-database/

Table 5: CVEs for which patches are included in NEQ3

CVE-2012-0881	CVE-2012-4386	CVE-2012-6612	CVE-2013-1879
CVE-2013-2035	CVE-2013-4310	CVE-2013-4316	CVE-2013-4366
CVE-2013-5679	CVE-2013-5960	CVE-2013-6397	CVE-2013-6407
CVE-2013-6408	CVE-2013-6430	CVE-2014-0168	CVE-2014-1972
CVE-2014-3600	CVE-2014-8152	CVE-2015-3253	CVE-2015-3271
CVE-2015-4165	CVE-2015-5258	CVE-2015-5531	CVE-2016-0785
CVE-2016-10006	CVE-2016-10750	CVE-2016-3720	CVE-2016-4437
CVE-2016-6814	CVE-2016-8738	CVE-2017-1000486	CVE-2017-1000487
CVE-2017-1000498	CVE-2017-12197	CVE-2017-14063	CVE-2017-3523
CVE-2017-3586	CVE-2017-7672	CVE-2017-9803	CVE-2018-1000129
CVE-2018-1000134	CVE-2018-1000820	CVE-2018-1000850	CVE-2018-1002200
CVE-2018-1002201	CVE-2018-10899	CVE-2018-11761	CVE-2018-11771
CVE-2018-11775	CVE-2018-11799	CVE-2018-12418	CVE-2018-12541
CVE-2018-12542	CVE-2018-1309	CVE-2018-1324	CVE-2018-15531
CVE-2018-17187	CVE-2018-17194	CVE-2018-17197	CVE-2018-20433
CVE-2018-21234	CVE-2018-8016	CVE-2018-8718	CVE-2019-0193
CVE-2019-0226	CVE-2019-1003005	CVE-2019-10071	CVE-2019-10088
CVE-2019-10091	CVE-2019-10093	CVE-2019-10094	CVE-2019-10173
CVE-2019-10462	CVE-2019-10463	CVE-2019-10770	CVE-2019-11343
CVE-2019-11777	CVE-2019-12422	CVE-2019-13990	CVE-2019-16771
CVE-2019-17513	CVE-2019-17555	CVE-2019-17556	CVE-2019-17572
CVE-2019-17638	CVE-2019-5427	CVE-2019-9658	CVE-2020-11050
CVE-2020-11987	CVE-2020-11988	CVE-2020-13692	CVE-2020-13973
CVE-2020-15250	CVE-2020-1729	CVE-2020-1925	CVE-2020-1929
CVE-2020-1953	CVE-2020-1963	CVE-2020-25020	CVE-2020-26258
CVE-2020-26259	CVE-2020-28191	CVE-2020-35460	CVE-2020-5289
CVE-2020-8929	CVE-2021-21234	CVE-2021-29425	

and the fixed version, the Git repository, the fix commit hash and the method modified in the fix.

5.4 Build Time and Environment

The computing environment used to create the NEQ oracles is identical with the environment used for EQ, reported in Section 4.7. Building NEQ1.tsv took 239 minutes (including all `revapi` runs, 2-way parallelized with `make -j 2`). Building all NEQ2 jar files and NEQ2.tsv took 15 minutes in total.

Building NEQ3 was done mainly manually by inspecting records using the process described above, we can therefore not report computing times.

6 Experiments

In order to illustrate how to use the benchmark, we describe an experiment using *tlsh* [40] to establish binary equivalence. *Tlsh* is a locality-sensitive hash that has been used in malware detection. We define two equivalence relations based on the *tlsh* implementation by *trendmicro* [22], with thresholds of 10 (*tlsh10*) and 100 (*tlsh100*), respectively, as follows:

$$b_1 \simeq b_2 \text{ iff } |tlsh(b_1) - tlsh(t_2)| < threshold$$

These thresholds were chosen based on the *tlsh* documentation [23]. Note that thresholding a distance measure in this way only approximates an equivalence relation, since transitivity is not guaranteed to hold. We computed the correctness of the classification we achieve for the various oracles in the benchmark using those equivalent relations. The results are reported in Table 6. This clearly shows that *tlsh10* is too sensitive to correctly identify most equivalent classes, but it can correctly distinguish between almost all non-equivalent classes. If the threshold is increased to 100, this is reversed, illustrating the trade-offs that often have to be made between false positives and false negatives. A false positive in this context means identifying pairs as equivalent when they are not, a false negative means not classifying a pair of equivalent classes as being equivalent. The BinEq benchmark can be used to assess this. It also provides a starting point for analyses to detect unexpected behaviour.

[22]https://github.com/trendmicro/tlsh, version 4.5.0
[23]https://github.com/trendmicro/tlsh/blob/master/java/src/main/java/com/trendmicro/tlsh/Tlsh.java

Table 6: Classification correctness of equivalence relations based on *tlsh10* and *tlsh100*.

oracle	EQ	EQ-no-AIC AIC	EQ-Same-Comp-no-aic	EQ-Diff-Comp-no-aic	NEQ1	NEQ2	NEQ3
tlsh10	0.235	0.236	0.428	0.002	0.97	1	0.985
tlsh100	0.778	0.774	0.858	0.67	0.428	0.406	0.173

7 Related Work

We focus the discussion on benchmarks and datasets for Java.

DaCapo [11] is an older benchmark widely used in program and performance analysis. Originally released in 2006 containing 11 programs, subsequent releases were made in 2009 and 2023. DaCapo comes with drivers to execute the programs. It draws from and improves upon older benchmarks such as *SPEC JVM98* [5].

The *Qualitas Corpus* [48] is a collection of 100 Java programs (20100719), 23 of which have multiple versions, for 495 versions in total. The corpus was updated in 2013. The Qualitas corpus has been mainly used for studies on source code such as code smell detection. A shortcoming of the corpus is that programs cannot be easily built. The evolution edition of the Qualitas Corpus that has several versions of the same program is similar to our use of different versions of the same project in EQ and NEQ1. Many projects in the Qualitas Corpus did not use Maven or similar build systems based on the convention over configuration philosophy, making it much harder to build projects mechanically. The reason is the age of the corpus – those build systems were only emerging when the projects that are part of the corpus were created and released. The *Qualitas.class* corpus [49] contains compiled versions of the respective projects. The compilation process used may differ from the intended build of the projects, and sometimes contains ad hoc refactoring to address compilation errors. The *XCorpus* [20] consists of 70 programs from the Qualitas Corpus and adds another 6 programs. It adds *ant*-based builds to compile the programs, and integrates existing and synthesized tests. The focus is on test coverage. It has been used to assess program analyses where dynamic and static techniques must be compared.

Jezek et al. [30] present a micro-benchmark consisting of synthetic code to assess API change analysis tools. This is related to NEQ1, which leverages such a tool.

The *50K-C* corpus [39] contains 50,000 compilable Java projects gathered from GitHub. The dataset includes the projects in both source and compiled forms, as well as dependencies and build scripts, and a VirtualBox VM for convenient usage. While the dataset is large, it is not curated – GitHub projects were downloaded (479,113 initially), and the dataset consists of projects randomly selected from the projects the authors were able to build.

There are several security-related datasets for Java, associating vulnerabilities and patches with programs. *Project-kb* [44] consists of 1,297 publicly disclosed vulnerabilities (624 at time of analysis) affecting 205 distinct open-source Java projects, and includes patch commits. We use this as a base for NEQ3. A slightly newer but smaller dataset is *Vul4j* [13] which consists of 79 vulnerabilities with patches from 51 open-source projects. *Vulinoss* [24] is a dataset extracted from the National Vulnerability Database (NVD), mapping vulnerabilities to project versions. It does not contain information about commit-level patches. *TaintBench* [36] is a specialised benchmark for taint analysis of Android programs.

Table 7 summarises related work and compares it with BinEq.

8 Threats to Validity

In Section 3.2, we assume that class files produced by compiling the same source class with different compilers, or with different versions of the same compiler, are semantically equivalent. However, due to reflection, and the fact that compilers may produce different but (absent reflection) functionally equivalent bytecode, this cannot be guaranteed. For instance, JEP 181 [46] removes the generation of synthetic methods for accessing outer-class private fields. It is, therefore, possible to craft code that uses reflection to count the number of methods in a class, and behave differently for different compiler versions. That said, arguably even bitwise equality is insufficient to guarantee an ideal notion of behavioural equivalence, since a program may be crafted to behave differently based on, e.g., the location of its class files in the filesystem. In practice, few programs explore these corners, so imperfect notions of equivalence remain useful.

In order to simplify the build process, the benchmark uses only Maven-based projects. This notably excludes Gradle-based projects such as the Spring Framework, potentially limiting diversity.

In Section 5.2, we do not run tests to verify that the mutations introduced in NEQ2 actually alter program semantics. Such a mutation would arguably be a bug in the *pitest* mutation testing framework, so the chances of this occurring are low. Given this fact, and the reality of incomplete test coverage, such stringent testing would likely shrink the dataset unnecessarily, and at considerable computational cost. Nevertheless this introduces a small possibility of false positives in NEQ2.

9 Conclusion

We have presented BinEq, a benchmark of compiled Java code labelled as either equivalent or non-equivalent. The benchmark consists of four parts, the equivalence oracle EQ with 465,858 records (518,138 when expanded to include anonymous inner classes), the non-equivalence oracles NEQ1 with 14,384 , NEQ2 with 141,585 and NEQ3 with 202 records, 622,029 records in total.

We hope that this will facilitate research into binary equivalence relations that approximate semantic equivalence of Java byte code, with applications to software supply chain security. The benchmark can not only be used for assessing relations, but also to train relations via supervised machine learning techniques. To this end it would be interesting to assess more sophisticated tools for determining equivalence, like SootDiff [17].

While the benchmark is large, there is potential to extend it as new compilers, compiler versions, bytecode features, code generators and build systems become available. Adding non-Maven build systems such as Gradle, and non-Java programs (Kotlin, Scala etc.) compiling into Java byte code are other desirable features for future versions of the benchmark. Similar problems exist in different ecosystems, in particular where diverse and quickly evolving compilers are being used, as in Rust/WASM [34]; similar benchmarks could be developed to improve build security for those platforms.

10 Acknowledgements

The work of the first author was supported by a gift by Oracle Labs Australia, the first and the second author were supported by the New Zealand National Science Challenge for Technological Innovation (Sfti) -funded Veracity project.

References

[1] [n. d.]. Pyrsia - decentralised package network. https://pyrsia.io/.
[2] [n. d.]. Reproducible Builds. https://reproducible-builds.org/.
[3] [n. d.]. Semantic Versioning 2.0.0. https://semver.org/.
[4] [n. d.]. Supply Chain Levels for Software Artifacts (SLSA) 1.0. https://slsa.dev/spec/v1.0/#core-specification.
[5] 1998. SPEC JVM98 Benchmarks. https://www.spec.org/jvm98/.
[6] 2003. CVE-2003-1564 (billion laughs). https://nvd.nist.gov/vuln/detail/CVE-2003-1564.
[7] 2022. Executive Order 14028, Improvin the Nation's Cybersecurity. https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity.
[8] 2024. CVE-2024-3094 (xz). https://nvd.nist.gov/vuln/detail/CVE-2024-3094.
[9] Anthony Andreoli, Anis Lounis, Mourad Debbabi, and Aiman Hanna. 2023. On the prevalence of software supply chain attacks: Empirical study and investigative framework. *Forensic Science International: Digital Investigation* 44 (2023), 301508.
[10] Tingting Bi, Boming Xia, Zhenchang Xing, Qinghua Lu, and Liming Zhu. 2023. On the way to sboms: Investigating design issues and solutions in practice. *ACM Transactions on Software Engineering and Methodology* (2023).
[11] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 169–190.
[12] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems* 30, 19 (2002).
[13] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E Díaz Ferreyra. 2022. Vul4J: a dataset of reproducible Java vulnerabilities geared towards the study of program repair techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 464–468.
[14] L Jean Camp and Vafa Andalibi. 2021. SBOM vulnerability assessment & corresponding requirements. *NTIA Response to Notice and Request for Comments on Software Bill of Materials Elements and Considerations* (2021).
[15] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. Pit: a practical mutation testing tool for java. In *Proceedings of the 25th international symposium on software testing and analysis*. 449–452.
[16] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. 2020. BScout: Direct Whole Patch Presence Test for Java Executables. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1147–1164. https://www.usenix.org/conference/usenixsecurity20/presentation/dai
[17] Andreas Dann, Ben Hermann, and Eric Bodden. 2019. SootDiff: bytecode comparison across different Java compilers. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (Phoenix, AZ, USA) *(SOAP 2019)*. Association for Computing Machinery, New York, NY, USA, 14–19. https://doi.org/10.1145/3315568.3329966
[18] Jens Dietrich, Kamil Jezek, and Premek Brada. 2014. 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)*. IEEE, 64–73.
[19] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. 2017. Evil pickles: DoS attacks based on object-graph engineering. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[20] Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. 2017. XCorpus – An executable Corpus of Java Programs. *Journal of Object Technology* 16, 4 (Aug. 2017), 1:1–24. https://doi.org/10.5381/jot.2017.16.4.a1

Table 7: Comparison of datasets from related research with BinEq. "Buildable" refers to easy buildability with scripts included in the dataset, "executable" to the availability of drivers (harnesses, test suites) as part of the dataset. Size categories: *small* (≤ 100), *medium* (100-1,000), *large* (≥ 1,000). BinEq dataset sizes use the number of jars, not records, for a fair comparison.

dataset	year(s)	size	real-world	buildable	executable	vulnerabilities	patches	evolution	build variability
Dacapo	2006,2009,2023	small	yes	yes	yes	no	n/a	no	no
Qualitas Corpus	2010,2013	medium	yes	no	no	no	n/a	yes	n/a
Qualitas.class	2013	medium	yes	yes	yes	no	n/a	no	no
XCorpus	2017	medium	yes	yes	yes	no	n/a	no	no
Jezek et al	2017	small	no	yes	yes	no	n/a	yes	no
50K-C	2018	large	yes	yes	yes	no	no	no	no
Vulinoss	2018	large	yes	no	no	yes	no	yes	n/a
Project-KB	2019	large	yes	no	no	yes	yes	yes	n/a
TaintBench	2022	small	yes	yes	no	yes	no	no	no
Vul4J	2022	medium	yes	no	no	yes	yes	yes	n/a
BinEq-EQ	2024	medium	yes	yes	no	no	n/a	yes	yes
BinEq-NEQ1	2024	small	yes	yes	no	no	n/a	yes	no
BinEq-NEQ2	2024	medium	partially	yes	no	no	n/a	no	no
BinEq-NEQ3	2024	small	yes	yes	no	yes	yes	yes	no

[21] Yufei Du, Omar Alrawi, Kevin Snow, Manos Antonakakis, and Fabian Monrose. 2023. Improving Security Tasks Using Compiler Provenance Information Recovered At the Binary-Level. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2695–2709.

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

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

[24] Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. 2018. VulinOSS: a dataset of security vulnerabilities in open-source systems. In *Proceedings of the 15th International conference on mining software repositories*. 18–21.

[25] Foyzul Hassan, Shaikh Mostafa, Edmund SL Lam, and Xiaoyin Wang. 2017. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 38–47.

[26] Behnaz Hassanshahi, Trong Nhan Mai, Alistair Michael, Benjamin Selwyn-Smith, Sophie Bates, and Padmanabhan Krishnan. 2023. Macaron: A Logic-based Framework for Software Supply Chain Security Assurance. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '23)*. 29–37. https://doi.org/10.1145/3605770.3625213

[27] Michael J Hohnka, Jodi A Miller, Kenrick M Dacumos, Timothy J Fritton, Julia D Erdley, and Lyle N Long. 2019. Evaluation of compiler-induced vulnerabilities. *Journal of Aerospace Information Systems* 16, 10 (2019), 409–426.

[28] Nasif Imtiaz, Seaver Thorn, and Laurie Williams. 2021. A comparative study of vulnerability reporting by software composition analysis tools. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–11.

[29] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, Samuel Ou, and Kelly Blincoe. 2023. Understanding Breaking Changes in the Wild. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1433—-1444. https://doi.org/10.1145/3597926.3598147

[30] Kamil Jezek and Jens Dietrich. 2017. API Evolution and Compatibility: A Data Corpus and Tool Evaluation. *Journal of Object Technology* 16, 4 (Aug. 2017), 2:1–23. https://doi.org/10.5381/jot.2017.16.4.a2

[31] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[32] Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. 2013. Fast location of similar code fragments using semantic'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. 1–6.

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

[34] Chenghao Li, Yifei Wu, Wenbo Shen, Zichen Zhao, Rui Chang, Chengwei Liu, Yang Liu, and Kui Ren. 2024. Demystifying Compiler Unstable Feature Usage and Impacts in the Rust Ecosystem. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

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

[36] Linghui Luo, Felix Pauck, Goran Piskachev, Manuel Benz, Ivan Pashchenko, Martin Mory, Eric Bodden, Ben Hermann, and Fabio Massacci. 2022. TaintBench: Automatic real-world malware benchmarking of Android taint analyses. *Empirical Software Engineering* 27 (2022), 1–41.

[37] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 643–653.

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

[39] Pedro Martins, Rohan Achar, and Cristina V Lopes. 2018. 50k-c: A dataset of compilable, and compiled, java projects. In *Proceedings of the 15th international conference on mining software repositories*. 1–5.

[40] Jonathan Oliver, Chun Cheng, and Yanggui Chen. 2013. TLSH–a locality sensitive hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*. IEEE, 7–13.

[41] Zhiyuan Pan, Xing Hu, Xin Xia, Xian Zhan, David Lo, and Xiaohu Yang. 2024. PPT4J: Patch Presence Test for Java Binaries. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 225, 12 pages. https://doi.org/10.1145/3597503.3639231

[42] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. 2020. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 1513–1531.

[43] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2020. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering* 25, 5 (2020), 3175–3215.

[44] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 383–387.

[45] Steven Raemaekers, Arie Van Deursen, and Joost Visser. 2014. Semantic versioning versus breaking changes: A study of the maven repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 215–224.

[46] John Rose. 2013. JEP 181: Nest-Based Access Control. https://openjdk.org/jeps/181.

[47] Aleksey Shipilev. 2015. JEP 280: Indify String Concatenation. https://openjdk.org/jeps/280.

[48] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. The qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia pacific software engineering conference*. IEEE, 336–345.

[49] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S Bigonha. 2013. Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *ACM SIGSOFT Software Engineering Notes* 38, 5 (2013), 1–4.

[50] Ken Thompson. 1984. Reflections on trusting trust. *Commun. ACM* 27, 8 (1984), 761–763.

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