

# DALEQ - Explainable Equivalence for Java Bytecode

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

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

**Abstract**—The security of software builds has attracted increased attention in recent years in response to incidents like *solarwinds* and *xz*. Now, several companies including Oracle and Google rebuild open source projects in a secure environment and publish the resulting binaries through dedicated repositories. This practice enables direct comparison between these rebuilt binaries and the original ones produced by developers and published in repositories such as Maven Central. These binaries are often not bitwise identical; however, in most cases, the differences can be attributed to variations in the build environment, and the binaries can still be considered equivalent. Establishing such equivalence, however, is a labor-intensive and error-prone process.

While there are some tools that can be used for this purpose, they all fall short of providing provenance, i.e. readable explanation of why two binaries are equivalent, or not. To address this issue, we present *daleq*, a tool that disassembles Java byte code into a relational database, and can normalise this database by applying datalog rules. Those databases can then be used to infer equivalence between two classes. Notably, equivalence statements are accompanied with datalog proofs recording the normalisation process. We demonstrate the impact of *daleq* in an industrial context through a large-scale evaluation involving 2,714 pairs of jars, comprising 265,690 class pairs. In this evaluation, *daleq* is compared to two existing bytecode transformation tools. Our findings reveal a significant reduction in the manual effort required to assess non-bitwise equivalent artifacts, which would otherwise demand intensive human inspection. Furthermore, the results show that *daleq* outperforms existing tools by identifying more artifacts rebuilt from the same code as equivalent, even when no behavioral differences are present.

## I. INTRODUCTION

Software is the foundation of modern digital infrastructure. Modern software systems are assembled from existing components, using automated processes like continuous integration and deployment. This has created new security problems as both components and processes can inject vulnerabilities into systems. Examples include *solarwinds*, *codecov*, *equifax* and *log4shell* [16], [28], [17] for compromised component, and *xcodeghost*, *ccleaner*, *shadowpad*, *shadowhammer* and *xz* [28], [3], [2], [30] for compromised processes.

Initiatives like reproducible builds [1], [25], [17], [8], [18] aim to enhance software supply chain security by rebuilding packages from source code independently and comparing the resulting binaries. The objective is to produce identical, bitwise equivalent binaries. Achieving this is not straightforward and often requires significant engineering effort [27], [39], [29], [34], [15], [32], [24]. Industry initiatives such as Google

Assured Open Source (*gaoss*) and Oracle Build-From-Source (*obfs*), among others, also focus on rebuilding open-source artifacts from source on secure and hardened build services. While the goal is not necessarily to achieve bit-by-bit equality with the reference binaries built and released by the open-source developers (referred to as *reference binaries*), substitutability remains a key requirement. I.e. a binary should be replaceable by the alternatively built binary without changing the behaviour of downstream programs. Substitutability is trivially achieved if the binaries are identical. If not, this becomes more complex, since behavioural equality is undecidable. However, it is still possible to under-approximate behavioural equality by devising an equivalence relation  $\simeq$  between binaries (i.e. in Java) where  $b_1 \simeq b_2$  implies that  $b_1$  and  $b_2$  have the same behaviour.

This idea was introduced in our previous work [12], where we evaluated how equivalences based on existing tools, such as decompilers, disassemblers, and bytecode normalization tools, performed across various datasets. In this paper, we build upon that work and present *daleq*, a novel approach that not only establishes equivalence between Java binaries but also provides a provenance, i.e., an explanation of why two binaries are either equivalent or not. Conceptually, *daleq* is similar to normalization tools like *jnorm* [37] and *JavaBEPEnv* [43], but with the added benefit of explainability features that help users gain confidence in the results.

The technique presented here has been successfully applied to evaluate open-source artifacts used and built from source in Oracle’s Graal Development Kit for Micronaut (GDK) <sup>1</sup>. This application underscores its practical impact on enhancing software supply chain security, addressing the critical need for product teams to verify that binaries remain uncompromised while maintaining functional equivalence.

This paper presents the following contributions:

- 1) *daleq* provides provenance information supporting both equivalence and non-equivalence statements that can be used by security engineers to assess, validate and trust its outputs.
- 2) *daleq* is based on datalog, following a widely used approach in static program analysis. The rule-based constructions assures its high correctness, i.e., while there is

<sup>1</sup><https://www.oracle.com/developer/gdk-developers/>

no formal proof of soundness, it is highly unlikely that *daleq* will flag pairs of binaries with different behaviour as equivalent.

- 3) the comparative evaluation suggests that *daleq* significantly outperforms the state-of-the-art tool *jnorm*. This directly translates into significant cost savings of engineering time required to assess alternative build outputs. Since *JavaBEPEnv* is not available to us, we have not been able to evaluate *daleq* against this tool.

This paper is organised as follows. We briefly review related work including publications and tools in Section II. This is followed by a detailed discussion of the design and implementation of *daleq* in Sections III and IV, respectively. We then evaluate *daleq* against two similar tools on a dataset consisting of jars from Maven Central, compared with jars built by *gaoss* and *obfs* in Section V. This is followed by a conclusion.

## II. RELATED WORK

### A. Diff Tools

*Diffoscope*<sup>2</sup> is a general-purpose diff tool that can also be applied to (the content) of jar files. It directly uses bytecode, without applying abstractions. This makes it sensitive to even minor changes, and creates noise, e.g. it reports changes caused by platform specific new line separators in metadata files. It also reports file attributes. In contrast, *daleq* focuses on differences that influence program behaviour, and on minimising noise (i.e. differences that do not influence program behaviour).

*JarDiff*<sup>3</sup> is a specialized tool designed to show differences in jar files built from Java programs. It shares some conceptual similarities with *daleq*, as both tools generate differences in bytecode abstractions through transformations, specifically using *scalap* and the *asm textifier*. However, *daleq* goes a step further by offering a more comprehensive approach, including detailed equivalence checks and advanced normalization techniques. While *JarDiff* focuses on establishing some equivalences (e.g., resolving constant pool ordering issues) without applying normalization, *daleq*'s equivalence analysis involves additional steps, including the application of rules after extracting the EDBs. This makes *daleq* a more robust solution for analyzing and comparing Java binaries.

There are several specialised diff tools available to check for changes in jars corresponding to different versions of the same program, including *revapi*<sup>4</sup>, *japicmp*<sup>5</sup>, and *clirr*<sup>6</sup>. Those tools focus on detecting API changes violating Java source and binary compatibility rules that can lead to compile- and link-time errors in downstream clients [11], [31], [21]. Those tools are not suitable to detect subtle semantic changes that may indicate a compromised. In general, when assessing artifacts rebuilt from the same source code, the APIs rarely change.

<sup>2</sup><https://diffoscope.org/>

<sup>3</sup><https://github.com/lightbend-labs/jardiff/>

<sup>4</sup><https://revapi.org/>

<sup>5</sup><https://siom79.github.io/japicmp/>

<sup>6</sup><https://github.com/ebourg/clirr>

### B. Binary Equivalence Levels

In our prior work [12], we addressed the strict definition of reproducible builds, which primarily focuses on bitwise equivalence, by proposing a more practical set of levels for establishing binary equivalence. Building on this conceptual foundation, we introduce a technique for achieving level 3 equivalence in this study. Our previous work also included a large-scale evaluation of existing tools, such as decompilers, disassemblers, and *jnorm*, that can be adapted for equivalence checking. Among these, *jnorm* demonstrated the strongest performance and continues to be a key component in our approach. In this paper, we extend the previous analysis by incorporating a discussion of potential spurious equivalences, which can arise during the equivalence checking process. The equivalence levels defined in [12] form the basis of the methodology applied in this work.

Building further on this line of research, Dietrich et al. [13] explored the synthesis of witnesses for non-equivalence statements through automated test case generation.

### C. Binary Normalization and Transformation Techniques

Schott et al. introduce *jnorm* [37], a tool designed to normalize Java bytecode using the *Jimple* intermediate representation provided by Soot [41]. This approach exemplifies a transformation function that aids in the construction of equivalence relations for binary comparisons. We adopt and evaluate this methodology in the context of binary equivalence for software supply chain security. It is important to note that *jnorm* was originally developed for *code similarity analysis*. As such, it applies abstractions that simplify certain details, potentially influencing the semantics of the program. We will explore examples of these abstractions and their impact on binary equivalence in the following sections.

Xiong et al. investigate barriers to the reproducibility of Java-based systems [43]. They analyse sources of bytecode variability across builds and propose *JavaBEPFix*, a tool designed to eliminate certain build-induced differences. In contrast to *jnorm* and disassembler- or decompiler-based transformations, *JavaBEPFix* produces normalised bytecode rather than an abstract representation. Nonetheless, it aligns with the framework of transformation-based equivalence [12]. Unfortunately, we were unable to include *JavaBEPFix* in our evaluation, as the tool is not publicly available.

A substantial body of work addresses bytecode similarity [6], [22], [10], [23], [9], [44], [36]. However, similarity-based approaches are generally unsuitable for establishing semantic equivalence. First, they offer no guarantees of behavioural equivalence, as even minor bytecode differences may alter program behaviour. Second, similarity-based equivalence is inherently non-transitive, limiting its utility for reasoning about correctness-preserving transformations.

Sharma et al. [38] demonstrate how bytecode canonicalization can improve the success rate of reproducible builds. Building upon our previous work [12], they employ *jnorm* for this purpose. Bytecode canonicalization is closely aligned with

the concept of equivalence, as both approaches involve comparing transformations, specifically, the *Jimple* files generated by *jnorm*, to evaluate the success of a rebuilt artifact. *daleq* takes this further by not only establishing equivalence but also offering detailed explanations and formal proofs, thereby increasing the reliability and confidence in the results.

#### D. Reproducible Builds

Hassanshahi et al. [20] present Macaron, a toolkit aimed at improving software supply chain security. Macaron addresses several challenges related to precise build reproducibility and binary variability by automatically identifying the exact source versions of artifacts and capturing detailed build environment metadata. Although primarily focused on Java/Maven, Macaron also supports additional language ecosystems. Similarly, Keshani et al.[24] examine methods for identifying the source code corresponding to Maven binary packages. This is a non-trivial task when rebuilding. Those issues led us to include source code analysis into *daleq* to check the assumption that the binaries being compared have been built from equal or at least equivalent sources. This will be discussed in Sections IV and V-A.

In the broader context, the Linux community has shown sustained interest in reproducible builds, supported by empirical and tooling-focused research. Ren et al.[33] and Bajaj et al. [5] analyse the prevalence and causes of unreproducible builds in Linux packages. Building on this, Ren et al. [34] introduce RepFix, a tool designed to automatically patch build scripts to improve reproducibility, and demonstrate its effectiveness on a set of Linux packages.

### III. DESIGN

#### A. Design Goals

*Daleq* is designed to provide a level 3 equivalence [12] for Java bytecode relation. This implies the following:

- 1) The equivalence is a proper equivalence relation, i.e. it is reflexive, symmetric and transitive.
- 2) Equivalent classes should have the same behaviour.
- 3) Bytecode sequences representing semantically equivalent instructions are considered equivalent.
- 4) Provenance is generated to support both equivalence and non-equivalence statements.

*Daleq* establishes equivalence by transforming bytecode and comparing the results of these transformations. Specifically, equivalence is defined as  $b_1 \simeq b_2$  if and only if  $transf(b_1) = transf(b_2)$ . The other three requirements are addressed in Sections III-B, III-E, and III-F, respectively.

#### B. Soundness vs Soundness

An equivalence relation should be sound in the sense that it under-approximates (undecidable) behavioural equivalence. In other words, equivalent classes should always exhibit the same behavior. However, it is also desirable to establish equivalence for as many class pairs as possible. Note that every pair of artifacts reported as non-equivalent requires manual inspection, which is both costly and prone to error.

In many practical static analyses, *soundness* is considered sufficient [26]. In the context of establishing equivalence between classes, *soundy* means that classes can be considered equivalent even if their behaviour differs, however, reflection or related features must be used by an application to expose this difference. The advantage of accepting *soundness* is that a *soundy* equivalence can create more equivalence statements. From the point of view of an engineer assessing non-equivalent classes for security issues, this equates to removing false positives. This matters as precision is known to be a crucial factor for the acceptance of static analysis tools [35], [14].

The downside of this approach is that some security issues might be overlooked. With *daleq* we took the following approach to address this: *daleq* uses a modular design where code normalisation patterns are implemented using separate files consisting of one or several rules, and each rule set is flagged as being either *sound* or *soundy*. Users could therefore simply remove *soundy* rules if strict *soundness* was required. This also facilitates scenarios where different versions of *daleq* with different rule sets could be used simultaneously.

For the *soundy* rule sets implemented, we carefully assessed the impact those equivalences might have on security.

#### C. Analysis Pipeline Overview

*Daleq* disassembles Java bytecode and normalises it in three stages: (1) **extraction**: a low-level relational representation of bytecode is created, this is referred to as the *EDB* (extensional database) (2) **inference**: rules are applied to extract a second database, the *IDB* (intensional database), those rules normalise some bytecode patterns. The rules applied are recorded within the *IDB* records. (3) **projection**: A textual representation of the *IDB* is created, and auxiliary information is removed from this representation. This representation can then be used for comparison, i.e. to establish equivalence. Figure 1 depicts this process.

#### D. Extraction

Extraction is based on *ASM* [7]. *ASM* is a well-established, well-maintained, low-level library used to extract and manipulate Java bytecode. The extraction layer creates datalog predicates and facts instantiating those predicates. The predicates fall into two categories: global and instruction predicates. Global predicates represent properties of classes not related to instructions in methods. Examples are superclass, interface, (class) access modifiers, annotations and predicates to list fields and methods declared in a class.

Instruction predicates represent bytecode instructions as defined in the JVM spec. For each such instruction, a predicate is defined representing an instruction as a fact. The actual instruction is performed by an instance of a `InstructionPredicateFactFactory` that uses an *ASM* node object (from *ASM*'s tree API) as input. To avoid the performance overhead and limitation on analysability of reflective code, we opted not to use reflection but instead to use a specific factory class for each instruction. As there is

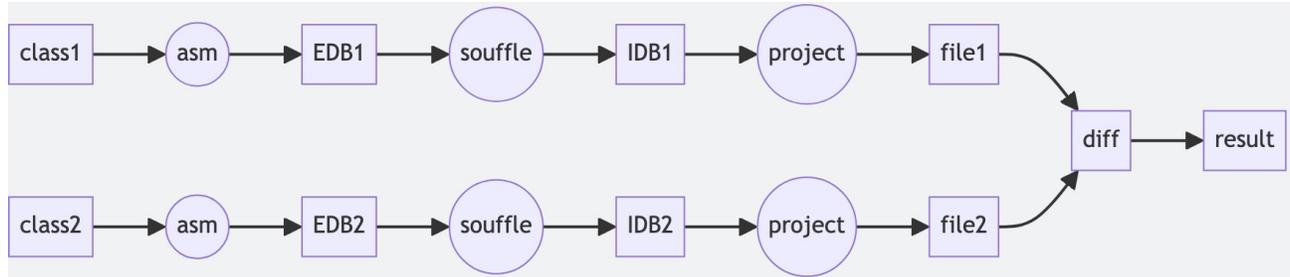


Fig. 1: *daleq* design overview

a large number of Java bytecode instructions<sup>7</sup>, we generated those factories statically from the respective *ASM* AST node types. The respective code generator(s) are part of the *daleq* distribution.

All extracted facts are stored in a tsv text file. There is one such file for each predicate, the name of the file is `<predicate-name>.facts`. The set of all such facts extracted and a file that defines the respective predicates in *souffle* format forms the EDB. This representation of a disassembly is on a level of abstraction similar to the output of the standard Java disassembler *javap*.

Notably, the extraction already applies several normalisations. Firstly, the EDB resolves constant pool references. For instance, in bytecode, at a call site, the invocation target is represented as a constant pool reference. This is being replaced by a fact that represents the target using a combination of defining class, method name and descriptor. The *ASM* API used already provides resolution of constant pool references.

Secondly, labels are normalised. *ASM* label nodes are mapped to simple names (label1, label2, etc) in EDB records. Label nodes that are never used (as jump targets, e.g. in conditional statements) are ignored.

Thirdly, line number tables are ignored, and there are no facts in the EDB to associate instructions with line numbers. We consider line numbers as code-reformatting noise during builds and inserting special legal comments into source code files will change those, but has no effect on the semantics of the program.

In terms of equivalence levels [12], this implies that comparing the EDBs of two .class files is a level 2 equivalence. I.e. aspects of bytecode not related to its semantics are removed (line numbers, unused labels), and only isomorphic transformations (label renaming) are applied.

Each EDB predicate has a *factid* column of type *symbol* as its first column. Unique values for this column are auto-generated by the extraction, using a simple pattern (“F” followed by a counter). Those ids are globally unique across predicates.

### E. Inference

The inference layer uses the *souffle* datalog engine. Datalog in general, and *souffle* in particular, are popular in static

program analysis [42], [19], [40], [4]. This is facilitated by datalog’s simple structure and fix point semantics.

Applying the datalog rules creates a new database, the IDB, (intentional database) from the input EDB. This is then used as the base for comparison. There is a base set of rules that translates each EDB fact into a corresponding IDB fact. This involves a new predicate, and two rules. For instance, consider Listing 1. This defines an IDB predicate `IDB_ILOAD`, and a rule that derives facts from the EDB predicate `ILOAD` (lines 3-4) representing `ILOAD` instructions found in the bytecode of a class. The first terms are the ids of facts. For brevity, those are omitted in Listing 1, we will discuss them later in Section III-F. The structure of the IDB predicate mirrors the respective EDB predicate. The first rule (line 3) translates the EDB fact into an equivalent IDB fact. This rule uses a second prerequisite (line 5) that acts as a guard. Custom rules can use the `REMOVED_INSTRUCTION` predicate to prevent certain IDB facts from being created. The standard rules for bytecode instructions also instantiate a second generic predicate `IDB_INSTRUCTION` (line 6-7). This can be used to describe patterns that apply to all instructions in a certain method.

```

1 .decl IDB_ILOAD(factid: symbol,methodid: symbol,
2               instructioncounter: number,var: number)
3 .output IDB_ILOAD
4 IDB_ILOAD(..,methodid,icounter,var) :-
5   ILOAD(..,method,icounter,var),
6   !REMOVED_INSTRUCTION(_,method,icounter).
7 IDB_INSTRUCTION(..,method,icounter,"ILOAD") :-
8   ILOAD(..,method,icounter,_).

```

Listing 1: Generated rules and predicate to define the IDB for the `ILOAD` instruction.

In addition to the standard rule set that describes EDB and IDB predicates and mappings between those, several custom rules are used. They are all organised in separate .souffle files stored in project resources (i.e. in `src/main/resources/rules`), and discovered and merged at runtime. The first custom rule set is `access.souffle`. This is included for convenience and readability, as it maps integer-encoded access flags to readable facts. An example is shown in Listing 2, defining facts for *synthetic* elements in a class. The id used references a class, field or method. Note that *band* is an operation provided by

<sup>7</sup>The JVM spec version 17 defines 149 <https://docs.oracle.com/javase/17/specs/jvms/se17/html/jvms-6.html#jvms-6.5>

*shuffle*. The premise is an `IDB_ACCESS` fact, itself derived from the `EBD_ACCESS` fact with a standard rule.

```

1 .decl IDB_IS_SYNTHETIC(factid: symbol, Id: symbol)
2 IDB_IS_SYNTHETIC(..,id) :-
3   IDB_ACCESS(factid,id,access), (access band 0x1000)!=0.
4 .output IDB_IS_SYNTHETIC

```

Listing 2: Custom rule to define synthetic classes, fields or methods

Other custom rules are related to normalisations, and aim at creating an equivalence IDB even though EDBs extracted from the respective classes are different. As a first example, consider a rule to normalise the facts representing the bytecode version of a `.class` file. This is shown in Listing 3. Here the bytecode version is always set to 0, enabling bytecodes to be equivalent even if their versions are different. Note how the guard fact for the predicate `REMOVED_VERSION` is instantiated by the second rule (line 4) in order to disable the default rule that would otherwise create a second `IDB_VERSION` fact with the version found in bytecode.

```

1 .decl IDB_VERSION(fid: symbol,classname: symbol,version:
   number)
2 IDB_VERSION(..,classname,0) :- VERSION(fid,classname,_).
3 .output IDB_VERSION
4 REMOVED_VERSION(..,classname) :- VERSION(fid,classname,_)

```

Listing 3: Custom rule to ignore the bytecode version.

In its current version, *daleq* contains rules to implement the following normalisation patterns (in addition to the normalisation performed during EDB construction):

- R1 Null checks in method reference operator, see also [37, Pattern N6]
- R2 Redundant `checkcast` instructions, see also [37, Pattern N9]
- R3 Inline `$values()` method in enumerations, see also [37, Pattern N10]
- R4 Ignore bytecode versions.
- R5 Invocation type of root methods defined in `java.lang.Object` on an object declared using an interface type.<sup>8</sup>
- R6 Anonymous inner classes are always implicitly final, but the flag is inconsistently set across different compilers. This is addressed by setting the access modifier of all anonymous inner classes to `final`.<sup>9</sup>

The selection of these patterns is based on their frequency of occurrence during the evaluation in the previous work [12, Sect. 3]. We did not implement all patterns reported by Schott et al and implemented in the *JNorm* tool. We ignored patterns that only apply when normalising Java bytecode produced by *javac* versions prior to Java 8 (N2,N3,N4,N5,N11) as we only rarely encounter them. N12 and N13 are normalisation patterns related to JEP280 and JEP181, respectively. While

<sup>8</sup>This pattern is not included in [37, Pattern N9], see <https://github.com/openjdk/jdk/pull/5165> and linked issues for a discussion of the change in the compilation strategy

<sup>9</sup><https://bugs.openjdk.org/browse/JDK-8161009>

those changed apply at Java 11, we did not encounter many instances of those changes causing inequality. Support for N13 would require some changes to how *daleq* works at the moment: reasoning is based on processing one (bytecode) class at a time, and N13 would require to make changes to an entire nest of classes, i.e. the set of classes generated from a single compilation unit, containing a class and all of its inner classes.

Two other *JNorm* patterns were rejected due to soundness concerns. Firstly, support for N1 (removal of synthetically generated methods) would introduce unsoundness as synthetic methods are part of the program semantics. Likewise, support for N7 (buffer method invocation) establishes equivalence for classes that would result in different behaviour for downstream clients. This is more subtle, and relates to binary compatibility. A detailed discussion of this particular issue can be found in [13].

To support the null checks in N6, we did not just replace calls to `Object::getClass` by `Objects.requireNonNull`<sup>10</sup>. The reason is that this could be behaviour-changing as different objects will end up on the stack depending on the method being invoked: `Objects.requireNonNull` returns the argument, while `Object::getClass` returns an object representing the type of the argument<sup>11</sup>. The respective rules therefore also look for guards to ensure the consistency of the stack – that the invocation of `Object::getClass` is preceded by a `DUP` and succeeded by a `POP` instruction. We find that in the context of method reference operator, the compiler generates such guards. Representing those constraints in datalog is straight-forward.

A particular interesting example is the normalisation of non-null checks (pattern R1), shown in Listing 4. Here an `invokevirtual Object::getClass` is replaced by an `invokestatic Objects::requireNonNull`. This is unsound in the sense that it changes the program behaviour as after the respective invocation, different objects are on the stack (the object vs an object representing its class). However, those statements are embedded by the compiler between a `DUP` and a `POP` instruction, ensuring the consistency of the stack. The respective rule uses a pattern in the body that reflects this (lines 4 and 5)<sup>12</sup>.

```

1 .decl LEGACY_NON_NULL_CHECK(factid: symbol, method:
   symbol, icounter: number)
2 LEGACY_NON_NULL_CHECK(..,method,icounter) :-
3   INVOKEVIRTUAL(factid1,method,icounter,"java/lang/Object",
   "getClass", "()Ljava/lang/Class;",_),
4   DUP(factid2,method,icounter-100),
5   POP(factid3,method,icounter+100).
6 IDB_INVOKESTATIC(..,method,icounter,"java/util/Objects",
   "requireNonNull", "(Ljava/lang/Object;)Ljava/lang/
   Object;",0) :-
7   LEGACY_NON_NULL_CHECK(factid1,method,icounter).
8 REMOVED_INSTRUCTION(..,method,icounter) :-

```

<sup>10</sup>*JNorm* transforms all occurrences back to the old null-checking mechanism, we chose the newer pattern to normalise

<sup>11</sup>See also <https://bugs.openjdk.org/browse/JDK-8073432>

<sup>12</sup>In this rule, an instruction counter offset is used to identify the previous and the next bytecode instruction. This offset is used during extraction to define instruction counters defining the order of instructions within a method, which is configurable.

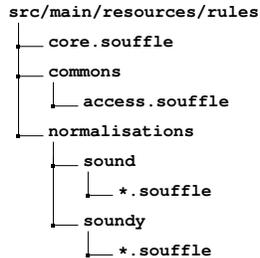


Fig. 2: Rule folder structure

```

9 LEGACY_NON_NULL_CHECK(factid1,method,icounter).

```

Listing 4: Custom rule to normalise non-null checks.

Two rules, R3 and R6 are only soundy as (synthetic) methods (R3) and final flags (R6) can be queried by programs at runtime via reflection. In both cases, it is highly unlikely that those subtle differences in behaviour can be exploited for security violations. Note that R6 only applies to anonymous inner classes.

In terms of equivalence levels [12], the comparison of the IDBs computed from two .class files is a level 3 equivalence. I.e. some abstraction takes place to identify bytecode that does not alter the behaviour of the program.

Rules are organised in a folder structure depicted in Figure 2. They are loaded by the class loader, allowing third parties to provide additional rules.

#### F. Recording Datalog Inference

The inference rules record provenance through the aggregated ids in the first term (column) of each fact. For facts, unique identifiers (“F1”, “F2”, ..) are generated during extraction, and used in the first term in each fact. When a rule is applied and a new fact is inferred, a composite identifier is created. For instance, in Listing 3, the id terms in rule heads were omitted for brevity. For the first rule (line 2), this term is `cat("R_REMOVE_BYTECODE_VERSION", "[" + fid, "]")`. If the id of the fact `fid` in the premise was F42, then this would create an id `R_REMOVE_BYTECODE_VERSION[F42]` for the derived fact.

The constructed ids of derived facts encode the derivations that were used to construct them. I.e., those values can be parsed and presented as a derivation tree. *daleq* includes a simple grammar for the embedded provenance language, along with utilities to parse these expressions and render them as trees in the final HTML report, which serves as the tool’s output. This grammar is shown in Listing 5.

```

1 grammar Proof;
2 proof : node <EOF> ;
3 node : ID children? ;
4 children : '[' node (',' node)* ']' ;
5 ID : [a-zA-Z0-9_]+ ;

```

Listing 5: Grammar for constructed ids encoding derivations.

Using those derivations, we can provide provenance to users supporting equivalence statements. For non-equivalence statements, standard diff tools can be employed. We will discuss this in the following section.

#### G. Projection and Establishing Equivalence

This is the last step of the analysis pipeline. While the IDBs are the base of comparing bytecodes, they cannot be used directly for two reasons. Firstly, the ids of derived facts contain information about the derivation, i.e. *how* the normalisation was achieved. To compare normalised code, this information needs to be removed. Secondly, the IDB still contains instruction counters. Those counters are only used to define the order of instructions. I.e. the particular numbering scheme is irrelevant as long as the order is retained. We therefore remove the respective terms from facts representing bytecode instructions.

The result of those two steps is referred as the *projection*. To establish equivalence, the IDB projections are printed into a textual representation, and then compared. If they are the same, equivalence has been established. Using a textual representation facilitates the use of standard diff tools. If classes are not equivalent, the produced diffs can be used as provenance. If they are equivalent, provenance is provided by a report that includes the derivations of rules applied.

### IV. IMPLEMENTATION

#### A. CLI and Report Generation

*Daleq* consists of several components. The extractor, which generates the EDB, is implemented in Java, while the normalisation rules are written in the *souffle* datalog<sup>13</sup>. *daleq* is primarily designed as a command-line interface (CLI) tool, but it can also be integrated into CI/CD pipelines for artifact validation. The tool accepts two jar files as input and generates an HTML report. Additionally, it can accept jar files containing source code, alongside the bytecode jars, to verify that the compared bytecodes were built from the same source code. This leverages the fact that the source code used for building the binaries is often distributed with them.

An example of an HTML report output is shown in Figure 3. Here, we compare the jar file for the artifact *javax.transaction:jta:1.1* from Maven Central with the corresponding artifact rebuilt by *gaoss*.

*Daleq* employs various *analysers* to compare classes, metadata and resources within the jars. The results are displayed in a table, the rows correspond to files within the jar(s), and the columns correspond to the various analysers. Possible analysis result states are:

- **PASS** the comparison yields true
- **FAIL** the comparison yields false
- **N/A** the comparison cannot be applied (e.g., a source code comparison cannot be used to compare bytecode)
- **ERROR** the comparison has resulted in an error

<sup>13</sup><https://souffle-lang.github.io/docs.html>

# Daleq Jar Comparison Report

## Jars compared

repol.maven.org/maven2/javax/transaction/jta/1.1/jta-1.1.jar

google-aoss/javax/transaction/jta/1.1/jta-1.1.jar

## Comparison results

resource	file present?	same source?	equiv. source?	same file?	javap -v	javap -c -p	daleq
META-INF/LICENSE.txt	PASS	N/A	N/A	FAIL <a href="#">Δ</a>	N/A	N/A	N/A
META-INF/MANIFEST.MF	PASS	N/A	N/A	FAIL <a href="#">Δ</a>	N/A	N/A	N/A
META-INF/maven/javax.transaction.transaction-api/pom.properties	FAIL <a href="#">Δ</a>	N/A	N/A	N/A	N/A	N/A	N/A
META-INF/maven/javax.transaction.transaction-api/pom.xml	FAIL <a href="#">Δ</a>	N/A	N/A	N/A	N/A	N/A	N/A
javax/transaction/HeuristicCommitException.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	PASS	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>
javax/transaction/HeuristicMixedException.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	PASS	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>
javax/transaction/HeuristicRollbackException.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	PASS	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>
javax/transaction/InvalidTransactionException.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	PASS	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>
javax/transaction/NotSupportedException.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	PASS	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>
javax/transaction/RollbackException.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	PASS	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>
javax/transaction/Status.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	PASS	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>
javax/transaction/Synchronization.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	PASS	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>
javax/transaction/SystemException.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	PASS	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>
javax/transaction/Transaction.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	PASS	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>
javax/transaction/TransactionManager.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	PASS	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>
javax/transaction/TransactionRequiredException.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	PASS	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>
javax/transaction/TransactionRolledbackException.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	PASS	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>
javax/transaction/TransactionSynchronizationRegistry.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	PASS	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>
javax/transaction/UserTransaction.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	PASS	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>
javax/transaction/xa/XAException.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	FAIL <a href="#">Δ</a>	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>
javax/transaction/xa/XAResource.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	PASS	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>
javax/transaction/xa/Xid.class	PASS	FAIL <a href="#">Δ</a>	PASS	FAIL	FAIL <a href="#">Δ</a>	PASS	PASS <a href="#">Δ</a> <a href="#">i</a> <a href="#">Δ</a>

Fig. 3: Report generated by *daleq*

The result states in the report are similar to the ones used by automated testing frameworks like *JUnit*<sup>14</sup>. Small markers next to the results indicate that they are explainable; these markers link to additional pages with more details. For failed comparisons, the linked pages typically display diffs (rendered in HTML), while for errors, they contain error logs.

The report lists all files found in either jar. The first check (column 2) verifies that the file is present in both jars. It then compares the sources for equality and equivalence<sup>15</sup> (columns 3 and 4). In the example shown in Figure 3, there are two resources in `META-INF/maven` that are present in the *gaoss* but not in the *mvnc* built jar. Interestingly, the sources used to create the jars are not identical. A closer inspection of the respective diffs reveals that this is caused by altered legal comments. But sources are still shown to be equivalent by the respective analysis. The [Δ](#) link references a page showing the diff generated during the analysis.

The same file comparison (column 5) then compares the content of the respective files, this reveals some changes in the metadata files `META-INF/LICENSE.txt` and `META-INF/MANIFEST.MF`.

The last two columns present the results of two bytecode analyses: one based on the standard disassembler (i.e., comparing the output of `javap -c -p`) and the other from the *daleq* comparison. *Daleq* generates several reports to facilitate

explainability, including links to the EDBs and IDBs used in the analysis.

Notably, *javap* shows equivalence for all classes except `XAException.class`, where the disassemblies differ. Details are provided in the linked diff created from the disassemblies. However, *daleq* can still establish equivalence between the two versions of this class.

### B. Explainability

For *daleq* equivalence results, *advanced-diff* reports are generated and made accessible through links in the main report. These reports include all extracted and derived facts, along with explanations of how custom rules were applied to establish equivalence. An example of this section of the report is shown in Figure 3. Derivation trees are visualized using CSS, and users can click on the facts and rules involved in the derivation.

### C. Adding Additional Analyses

The *daleq* tool features an extensible design, allowing new analyses to be added via the `io.github.bineq.daleq.cli.Analyser` interface, which can be implemented by third parties. Analysers are discovered and loaded through the Java service loader mechanism, enabling them to be used as plugins.

## V. EVALUATION

The evaluation is guided by the following research questions:

<sup>14</sup>In testing, often *skip* is used instead of *N/A*

<sup>15</sup>Using an AST-based analysis that ignores formatting and comments

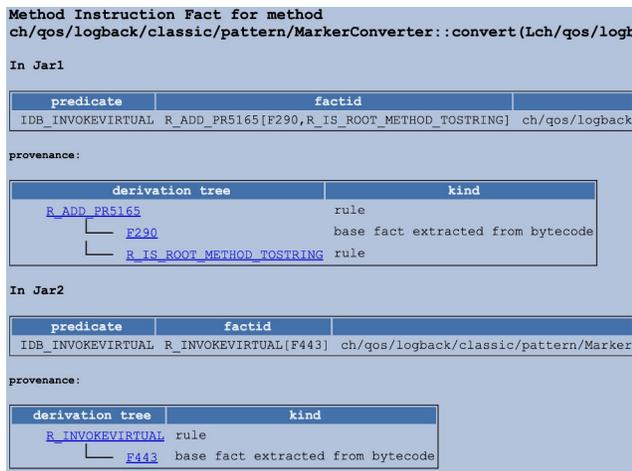


Fig. 4: Provenance user interface

- RQ1** How common are bitwise differences in Java class files between developer-built artifacts (*mvnc*) and independently rebuilt artifacts from trusted providers such as *obfs* and *gaoss*?
- RQ2** How effective is *daleq* in classifying non-bitwise-equal classes as equivalent compared to existing tools such as *javap* and *jnorm*?
- RQ3** What are the most frequent causes of non-equivalence in rebuilt Java class files, and to what extent can they be explained through recurring bytecode-level patterns?
- RQ4** How scalable is *daleq*?

#### A. Methodology

To evaluate the performance of *daleq*, we used the dataset from [12]<sup>16</sup>. In particular, we compared jars built by the developer (deployed on Maven Central *mvnc*) with alternatively built jars from Oracle’s Build-From-Source (*obfs*) and Google’s Assured Open Source (*gaoss*) projects. We ensured that the respective sources (released with the jars) are equivalent, using the AST-based comparison also used in [12]. The number of jars and classes compared are summarised in Table I. Both datasets *mvnc-vs-obfs* and *mvnc-vs-gaoss* contain a similar number of classes (i.e. compiled class files, \*.class), ca 130,000. The numbers of jars significantly differs, *obfs* contains more smaller jars, whereas *gaoss* contains fewer but larger jars. In both cases, we found a small number of jars without .class files. These jars contain only resources and meta-data, and were therefore excluded from the analysis.

We have created scripts that summarise the results for RQ1 and RQ2. They also produce a file structure, which is used as input for additional analysis scripts that answer RQ3 and RQ4. The corresponding data is available at <https://zenodo.org/records/16628896>.

<sup>16</sup>Available at <https://zenodo.org/records/14915249>

#### B. RQ1 Results

Results for RQ1 can be found in Table I. We found a significant number of non-equal classes across alternative builds, 2,158 (1.58%) for the *mvnc-vs-obfs* comparison, and 33,470 (25.69%) for the *mvnc-vs-gaoss* comparison.

Interestingly, different processes and design goals by alternative providers lead to significantly different rates of classes that are bitwise identical.

#### C. RQ2 Results

We then identified classes that are not bitwise equivalent (i.e., level 1), but can still be shown to be equivalent at higher levels. To evaluate the performance of *daleq* relative to other tools, we used the standard Java disassembler<sup>17</sup> and *jnorm*<sup>18</sup>. *jnorm* was the best-performing tool in a previous study [12]. Note that in [12] *jnorm* was also used with the aggressive normalisation option. We did not use aggressive normalisation here as it includes some normalisation patterns that we consider unsound, and even unsoundy, such as N16 described in [37]. *javap* was also used in [12] and performed reasonably well. In particular, it was able to analyse all classes, whereas some of the other tools like the *fernflower* decompiler (used in [12]) and *jnorm* (used both in [12] and this study) sometimes resulted in errors. *javap* is also a tool that is trusted by engineers as it is well-maintained, and part of the standard Java Developer Kit. It therefore forms a suitable baseline of the evaluation of tools to establish equivalence. Using the `-c -p` configuration provides some abstractions, whereas using `-v` (verbose) would have resulted in a very detailed representation (including the line number table and the constant pool) not suitable to establish equivalence between different classes.

Equivalences are established by comparing bytecode transformations generated by the respective tool. Those are then diffed, and if the diff is empty, the respective classes are considered equivalent. Otherwise the diff is stored, those diffs are then used later to answer RQ3.

The results are shown in Table II. *Daleq* can infer that 85.91% of non-equal classes in the *mvnc-vs-obfs* comparison and 90.80% of non-equal classes in the *mvnc-vs-gaoss* comparison can be shown to be equivalent. This indicates a substantial reduction in manual effort required to assess these cases. *Daleq* outperforms both the *javap*- (45.18% / 35.08%) and the *jnorm*-based (65.06% / 80.90%) equivalences. There were also cases where *jnorm* encountered errors, which are reported in the last column and counted as *jnorm*-non-equivalent. In contrast, *daleq* successfully analyzed all classes in the evaluation dataset.

#### D. RQ3 Results

While *daleq* generally performs well and significantly reduces the manual effort required to assess non-equivalent classes, a substantial number of such classes remain in the report. We expect that additional patterns can be discovered

<sup>17</sup>`javap -c -p`

<sup>18</sup>version 1.0.0 with the normalisation option `-n`.

provider1	provider2	jars	jars without classes	classes	equal classes	non-equal classes	non-equal classes (%)
<i>mvnc</i>	<i>obfs</i>	1,922	22	135,425	133,267	2,158	1.59%
<i>mvnc</i>	<i>gaoss</i>	792	13	130,265	96,795	33,470	25.69%

TABLE I: Non-equal classes

provider1	provider2	non-equal classes	equiv. (javap)	equiv. (jnorm)	equiv. (daleq)	errors (jnorm)
<i>mvnc</i>	<i>obfs</i>	2,158	975 (45.18%)	1,404 (65.06%)	1,854 (85.91%)	5 (0.23%)
<i>mvnc</i>	<i>gaoss</i>	33,470	11,741 (35.08%)	27,079 (80.90%)	30,390 (90.80%)	461 (1.38%)

TABLE II: Equivalent classes

and implemented using *souffle* normalisation rules, and we make no claim that the current set of rules is exhaustive.

To gain a better understanding of the nature of non-equivalent classes, we analysed them using the following approach. If two `.class` files are not equivalent, a diff file `daleq-diff.txt` is generated showing the differences of the respective IDBs in standard diff format. There are 3,384 such files<sup>19</sup> in the results. We analysed these files to identify common patterns in changes. Pattern detection was implemented using a text analysis of added and removed lines. For instance, to check for changes in loaded constants, we collected lines starting with `+IDB_LDC` and `-IDB_LDC`, respectively. Then we extracted the values loaded by collecting the last tokens from those lines (the lines represent database records, using a tab as separator), compared those to sets, and report an instance of this pattern if those sets are different. This can potentially produce some false negatives, e.g. if constants loaded are swapped between different methods. We consider this unlikely.

The following are the analysed patterns:

**CHECKCAST** A checkcast statement is removed or added. In general, checkcasts can change the behaviour of programs – when checkcasts fail, a runtime exception is created, changing the control flow of a program. Casts can therefore not be ignored, unless behavioural equivalence can be inferred from context<sup>20</sup>. An example is shown in Listing 6. This issue is also discussed in [37] (N16). The authors argue that this is not used in normalisation, but only in aggressive normalisation mode.

```

1 --- version1
2 +++ version2
3 @@ -582,6 +582,7 @@
4   IDB_GETFIELD ..
5   IDB_ALOAD ..
6   IDB_INVOKEVIRTUAL ..
7 +IDB_CHECKCAST   projected org/apache/curator/shaded/com/
   google/common/graph/StandardNetwork::incidentNodes (
   Ljava/lang/Object;) Lorg/apache/curator/shaded/com/
   google/common/graph/EndpointPair; -1 org/apache/
   curator/shaded/com/google/common/graph/
   NetworkConnections
8   IDB_INVOKESTATIC ..
9   IDB_CHECKCAST ..
10  IDB_ALOAD ..
11

```

Listing 6: Non-equivalence caused by an additional checkcast instruction (some lines abbreviated).

<sup>19</sup>This is the sum of non-*daleq* equivalent records reported in Table II, i.e.  $(2,158-1,854)+(33,470-30,390)$

<sup>20</sup>As is the case in P2 discussed in Section III-E

**CONSTANT** A constant value in a LDC instruction is changed. We found instances where different values point to different paths. This can have security implications, for instance, differences pointing to file system locations may indicate path traversal (CWE-22) attacks. We found several path references that differ between *mvnc* and *gaoss* builds of *dev.zio:zio-test\_3:2.1.0-RC2*, those are references to scala source files, the paths seem to point to the build environment being used, starting with `/home/runner/work/` and `/workspace/shared-workspace/build/upstream-repo`, respectively.

**SBINIT** `StringBuilder` initialisation. There are different sequences being generated to initialise `java.lang.StringBuilder`, using different descriptors `()V` and `(I)V`, respectively.

**SIGNTR** Missing signature attributes in methods. There are cases where one class misses the signature. This is a candidate for another “soundy” rule, those changes may effect program behaviour if reflection is used.

**SYNMET** Naming of synthetic methods. Different compilers sometimes name synthetic classes differently. Ignoring them would make the analysis unsound as those methods encode program behaviour.

**SYNFLD** Naming of synthetic fields. Different compiler sometimes name synthetic classes differently. Ignoring them would make the analysis unsound as those methods fields have an impact on program behaviour.

**ANNO** Changed annotations. In particular, we found subtle differences in the parameters of checker framework’s `@Nullable` annotations. We consider annotations critical for security as modern frameworks heavily rely on them to define the semantics of a program. For instance, both *springframework* and *micronaut* use annotations to define entry points for web applications.

**ACCESS** Changed access. While we capture a particular pattern (P6), there are other cases where the flags associated with class, methods and fields generated by different builds differ.

The results of this analysis are summarised in Table III, column 2. We also counted the number of classes where non-equivalence has more than one cause in the last row.

At least two of those patterns (SIGNTR and ACCESS) are candidates for additional soundy rules. It is not clear whether normalising synthetic method names is expressible in *daleq*, if so, some of the differences in the SYNFLD and SYNMET categories could also be covered by additional rules.

cause	all	javap non-equiv.	jnorm non-equiv.
total	3,384	49	523
CHECKCAST	635 (18.76%)	0 (0.00%)	17 (3.25%)
CONSTANT	152 (4.49%)	0 (0.00%)	0 (0.00%)
SBINIT	650 (19.21%)	0 (0.00%)	6 (1.15%)
SIGNTR	396 (11.70%)	0 (0.00%)	385 (73.61%)
SYNMET	309 (9.13%)	0 (0.00%)	61 (11.66%)
SYNFLD	612 (18.09%)	0 (0.00%)	0 (0.00%)
ANNO	75 (2.22%)	45 (91.84%)	60 (11.47%)
ACCESS	630 (18.62%)	4 (8.16%)	19 (3.63%)
multiple	926 (27.36%)	0 (0.00%)	20 (3.82%)

TABLE III: Analysis of pairs of classes not *daleq*-equivalent

CONSTANT is category that could indicate a compromised binary, and warrants further investigation.

We then analysed *daleq* non-equivalent classes flagged as equivalent by either *javap* or *jnorm*. The results are also included in Table III, columns 3 and 4, respectively. We note here that the *javap* equivalence is based on running `javap -c -p` which produces a high-level representation of bytecode with some abstractions. Had we used `javap -v`, those values would have been much lower, possibly zero. On the other hand, the results for *javap* as reported in Table II would have been worse, too. Those choices reflect the classic tradeoff between precision and recall. There are actual cases where the *javap*-based equivalence becomes unsound, such an example (changed value of constant not used in the defining class) was detected in discussed in [12].

For *jnorm*, we note here that *jnorm* has not been designed to establish bytecode equivalence, but for *code similarity analysis*. For instance, the issues caused by synthetic methods (SYNMET) are a result of *jnorm* ignoring those methods (normalisation rule N1 in [37]), which we consider unsound. We also found that the *jimple* representation used by *jnorm* sometimes omits information like signature headers, leading to the high numbers in the SIGNTR category.

### E. RQ4 Results

To assess the scalability of *daleq*, we have computed some statistics from the data captured in the *computation-time-in-ms.txt* files. Those files record the time taken to extract the EDB, compute the IDB and project it. The experiments were performed on an Apple M1 Pro running MacOS 15.5 with 32GB memory. The JVM used was the *OpenJDK Runtime Environment Corretto-21.0.5.11.1*, the *soffle* version used was 2.4.1.

We analysed 71,256 such timestamps. This number is twice the sum of the number of non-equivalent classes (see Table I, column 7). For equal (i.e. bitwise identical) classes, *daleq*-equivalence is not computed but inferred. This hinges on the assumption that *daleq* and the tools it relies on (*ASM*, *daleq*) are deterministic, i.e. the same EDBs and IDBs are computed from identical input.

The mean time it takes to compute the projected IDB for a class is 1,691 ms. To put this in context, consider a larger Java library like *guava-33.4.0-jre.jar*, containing 2,018 .class files. Further assume that of those classes 20% are not bitwise identical. This means that for 808 classes ( $2 * 0.2 * 2,018$ )

*daleq* has to compute the projected IDB. This would take under 23 mins. Some additional time would be required to compare the generated reports for equality, and diff resources and metadata, but those operations are very fast. Most libraries are significantly smaller than this, and a typical *daleq* analysis takes only a few minutes.

## VI. CONCLUSION

In this paper, we have introduced *daleq*, a tool designed to compare Java bytecode and establish equivalence, facilitating the assessment of rebuilt artifacts. *daleq* operates at level 3 equivalence, with a particular focus on providing provenance for both equivalence and non-equivalence statements. This is achieved through a datalog-based implementation of bytecode normalisations that records derivations and exposes them to users, enhancing transparency and traceability. The evaluation on a large real-world dataset further confirms that *daleq* performs well and outperforms the state-of-the-art *jnorm* tool.

We have demonstrated the impact of *daleq* in an industrial context by evaluating artifacts built from source by two vendors. The results show a significant reduction in the manual effort required to assess non-bitwise equivalent artifacts, which would otherwise require intensive human inspection. *daleq*'s ability to automatically establish equivalence and explain the reasoning behind these results offers substantial time savings, especially when dealing with large-scale datasets.

*Daleq* is publicly available. By releasing it to the public, we invite contributions to enhance its functionality. We anticipate that this will lead to the development of additional rules and integration of other analysers into the tool.

### A. Tools and Data Availability

tool:	<a href="https://github.com/binaryeq/daleq/">https://github.com/binaryeq/daleq/</a>
evaluation scripts:	<a href="https://github.com/binaryeq/daleq-evaluation">https://github.com/binaryeq/daleq-evaluation</a>
input dataset:	<a href="https://zenodo.org/records/14915249">https://zenodo.org/records/14915249</a>
evaluation result data:	<a href="https://zenodo.org/records/16628896">https://zenodo.org/records/16628896</a>

## ACKNOWLEDGMENTS

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

## REFERENCES

- [1] Reproducible builds. <https://reproducible-builds.org/>.
- [2] CVE-2024-3094 (xz), 2024. <https://nvd.nist.gov/vuln/detail/CVE-2024-3094>.
- [3] Anthony Andreoli, Anis Lounis, Mourad Debbabi, and Aiman Hanna. On the prevalence of software supply chain attacks: Empirical study and investigative framework. *Forensic Science International: Digital Investigation*, 44:301508, 2023.
- [4] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. Porting doop to sofflé: a tale of inter-engine portability for datalog-based analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 25–30, 2017.
- [5] Rahul Bajaj, Eduardo Fernandes, Bram Adams, and Ahmed E Hassan. Unreproducible builds: Time to fix, causes, and correlation with external ecosystem factors. *Empirical Software Engineering*, 29(1):11, 2024.
- [6] Brenda S Baker and Udi Manber. Deducing similarities in java sources from bytecodes. In *USENIX Annual Technical Conference*, pages 179–190, 1998.

- [7] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30(19), 2002.
- [8] Simon Butler, Jonas Gamalielsson, Björn Lundell, Christoffer Brax, Anders Mattsson, Tomas Gustavsson, Jonas Feist, Bengt Kvarnström, and Erik Lönröth. On business adoption and use of reproducible builds for open and closed source software. *Software Quality Journal*, 31(3):687–719, 2023.
- [9] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186, 2014.
- [10] Ian J Davis and Michael W Godfrey. From whence it came: Detecting source code clones by analyzing assembler. In *2010 17th Working Conference on Reverse Engineering*, pages 242–246. IEEE, 2010.
- [11] 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.
- [12] Jens Dietrich, Tim White, Behnaz Hassanshahi, and Paddy Krishnan. Levels of binary equivalence for the comparison of binaries from alternative builds. In *Accepted for ICSME'25*. IEEE, 2025.
- [13] Jens Dietrich, Tim White, Valerio Terragni, and Behnaz Hassanshahi. Towards cross-build differential testing. In *Proc. ICST'25*. IEEE, 2025.
- [14] 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.
- [15] Joshua Drexel, Esther Hänggi, and Iyán Méndez Veiga. Reproducible builds and insights from an independent verifier for arch linux. In *Sicherheit 2024*, pages 243–257. Gesellschaft für Informatik eV, 2024.
- [16] 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.
- [17] 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.
- [18] Marcel Fourné, Dominik Wermke, William Enck, Sascha Fahl, and Yasemin Acar. It's like flossing your teeth: On the importance and challenges of reproducible builds for software supply chain security. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1527–1544. IEEE, 2023.
- [19] Elnar Hajiyev, Mathieu Verbaere, and Oege De Moor. Codequest: Scalable source code queries with datalog. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2006.
- [20] 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.
- [21] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, Samuel Ou, and Kelly Blincoe. Understanding breaking changes in the wild. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 1433–1444, 2023.
- [22] Jeong-Hoon Ji, Gyun Woo, and Hwan-Gue Cho. A plagiarism detection technique for java program using bytecode analysis. In *2008 third international conference on convergence and hybrid information technology*, volume 1, pages 1092–1098. IEEE, 2008.
- [23] Iman Keivanloo, Chanchal K Roy, and Juergen Rilling. Sebyte: Scalable clone and similarity search for bytecode. *Science of Computer Programming*, 95:426–444, 2014.
- [24] Mehdi Keshani, Tudor-Gabriel Velican, Gideon Bot, and Sebastian Proksch. Aroma: Automatic reproduction of maven artifacts. *Proc. FSE'24*, (FSE), 2024.
- [25] Chris Lamb and Stefano Zacchiroli. Reproducible builds: Increasing the integrity of software supply chains. *IEEE Software*, 39(2):62–70, 2021.
- [26] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015.
- [27] Christian Macho, Stefanie Beyer, Shane McIntosh, and Martin Pinzger. The nature of build changes: An empirical study of maven-based build systems. *Empirical Software Engineering*, 26(3):32, 2021.
- [28] 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.
- [29] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. Fixing dependency errors for python build reproducibility. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, pages 439–451, 2021.
- [30] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *Proc DIMVA'20*. Springer, 2020.
- [31] Steven Raemaekers, Arie Van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 215–224. IEEE, 2014.
- [32] Georges Aaron Randrianaina, Djamel Eddine Khelladi, Olivier Zendra, and Mathieu Acher. Options matter: Documenting and fixing non-reproducible builds in highly-configurable systems. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 654–664. IEEE, 2024.
- [33] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. Automated localization for unreproducible builds. In *Proceedings of the 40th International Conference on Software Engineering*, pages 71–81, 2018.
- [34] Zhilei Ren, Shiwei Sun, Jifeng Xuan, Xiaochen Li, Zhide Zhou, and He Jiang. Automated patching for unreproducible builds. In *Proceedings of the 44th International Conference on Software Engineering*, pages 200–211, 2022.
- [35] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspán. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.
- [36] Jean-Guy Schneider and Sung Une Lee. An experimental comparison of clone detection techniques using java bytecode. In *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*, pages 139–148. IEEE, 2022.
- [37] Stefan Schott, Serena Elisa Ponta, Wolfram Fischer, Jonas Klauke, and Eric Bodden. Java bytecode normalization for code similarity analysis. 2024.
- [38] Aman Sharma, Benoit Baudry, and Martin Monperrus. Canonicalization for unreproducible builds in java. *arXiv preprint arXiv:2504.21679*, 2025.
- [39] Yong Shi, Mingzhi Wen, Filipe R Cogo, Boyuan Chen, and Zhen Ming Jiang. An experience report on producing verifiable builds for large-scale commercial systems. *IEEE Transactions on Software Engineering*, 48(9):3361–3377, 2021.
- [40] Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In *International Datalog 2.0 Workshop*, pages 245–251. Springer, 2010.
- [41] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundareshan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224, 2010.
- [42] John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. Using datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems*, pages 97–118. Springer, 2005.
- [43] 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.
- [44] Dongjin Yu, Jiazha Yang, Xin Chen, and Jie Chen. Detecting java code clones based on bytecode sequence alignment. *IEEE Access*, 7:22421–22433, 2019.