School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Internet: office@ecs.vuw.ac.nz

A Machine Learning Approach to Binary Equivalence

Alix Schultze

Supervisors: Jens Dietrich, Andrew Lensen

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

Abstract

In 1984 Ken Thomson demonstrated the danger posed by malicious software compilers. Efforts to verify that compiled binaries have not been compromised by malicious build tools commonly involve building them in separate environments, and then comparing them to identify any discrepancies. However, comparing binaries is not straightforward. This project aims to identify binary equivalence of Java classes by applying the CodeT5+ large language model to disassembled bytecode. The resulting embeddings are classified as equivalent or not equivalent with an accuracy of 92%, showing that large language models can be used to identify binary equivalence with reasonable success. This report describes the methodology and findings of this research.

Contents

Chapter 1

Introduction

Compilers are used to convert source code into executable files known as binaries. In 1984 Ken Thomson demonstrated the potential danger of trusting a compiler to generate a binary that accurately reflects the original source code [1]. Thomson showed this by creating a compiler that could insert a backdoor into the binaries it produced. This compiler could also detect whether it was compiling itself and insert code to propagate its malicious behaviour into the new compiler. Inspection of the source code provided no defence against this because the compiler could build a compromised binary from completely clean source code.

Compromised build tools, such as compilers and linkers, can be used to carry out software supply chain attacks. Real-world cybersecurity breaches illustrate the stealth and extensive reach of these attacks. For example, the SUNBURST backdoor targeted SolarWinds Orion, a popular network monitoring tool. SUNBURST was compiled into Orion through a compromised build server and delivered to customers as a digitally signed update. The backdoor was unknowingly downloaded by 18,000 SolarWinds customers, including numerous government agencies and major organisations [2].

1.1 The Problem

Consumers of modern software often do not compile software from source code themselves but obtain pre-built binaries from third parties. An everyday example of this is installing a web browser or game on a home computer, but the practice extends well beyond that. Software vendors often depend on binaries obtained from other vendors or open-source repositories. For example, Google's Assured Open Source Software project provides open-source packages that have been built on their Cloud Build platform [3]. There are various package repositories that make it easy for contributors to share pre-built software: Maven Central for Java is an example [4]. Given Thomson's demonstration, how can we verify that these third-party binaries are true representations of their source code and do not contain any malicious behaviour? A way to verify pre-built binaries is needed.

Formal methods for verifying the equivalence of programs have limitations, and do not supply a general solution for non-trivial programs [5, 6]. A pragmatic approach involves compiling binaries in separate build environments and comparing them to detect any discrepancies. Users of open-source libraries may take this approach to verify that pre-built packages actually correspond to the source code they claim to originate from. This method relies on the premise that extra functionality injected by a compromised build environment would result in a noticeably different binary. Additionally, the ability to generate equivalent binaries using different build environments increases trust in their authenticity. This is

because compromising two distinct build environments is more challenging for an attacker than compromising just one.

However, a bit-for-bit or byte-for-byte comparison of binaries is not suitable for identifying discrepancies because many software-building tools have not been designed with this usage in mind, so it is usual for their outputs to vary. For example, compilers can exhibit non-deterministic behaviour arising from bugs[1] or multi-threading, producing different binaries from the same source code. There are efforts to make build pipelines deterministic, but this can be difficult to achieve and cannot address software that has already been built and distributed [7].

Artificial intelligence presents another possibility for comparing binaries. Large language models (LLMs) can operate at high levels of abstraction, which may allow us to identify discrepancies in behaviour regardless of details such as when and how binaries were built. Some downsides of this approach are the potentially high computational costs, the need for large amounts of training data and the difficulty of explaining how an LLM arrives at its outcomes.

This project aims to implement and evaluate the use of a large language model for examining pairs of binaries, with the goal of developing a method that can reliably classify pairs of binaries as equivalent (EQ) or not equivalent (NEQ). Such a method would be helpful in verifying that compiled binaries are true representations of their source code.

1.2 Binary Equivalence

We consider two binaries to be equivalent if they have equivalent behaviour, meaning that a given input should always produce the same output. This definition allows us to differentiate between normal binaries and binaries with malicious inclusions because they behave differently from each other. The idea of binary equivalence is discussed in [8], which gives the example of binaries compiled from the same source code. Two binaries compiled from the same source code should exhibit equivalent behaviour, even if they are compiled with different compilers (assuming the compilers implement the same standards).

Binaries compiled from different source code could also have equivalent behaviour. This might happen due to refactoring, where the source code is changed but the program's behaviour stays the same, or through coincidence. However, this project is only concerned with the equivalence of binaries compiled from the same source code.

The project makes use of a pre-existing 'BinEq' dataset, which consists of pairs of Java classes that are labelled equivalent (EQ) or not equivalent (NEQ) [8, 9]. The two groups can each be broken down into sub-groups as follows:

- EQ-DiffComp: pairs compiled with different Java compilers
- EQ-SameComp: pairs compiled with the same Java compiler, but different major version numbers; see listings 2.2 and 2.3 for two EQ bytecode samples compiled from the same source code
- EQ-SameMjCompVer: pairs compiled with the same Java compiler and same major version number
- NEQ1: pairs that differ due to breaking API changes
- NEQ2: pairs that differ due to changes applied by a mutation testing framework

[1] Example of a non-determinism bug in the Java compiler: https://bugs.openjdk.org/browse/JDK-8264306 Accessed 2024-10-13

- NEQ3: pairs that differ due to vulnerability patches

This project uses machine learning techniques to classify items in these categories. It takes pairs of Java classes from the dataset, creates bytecode documents for each class and uses a large language model to calculate embeddings for those documents. Finally, a classifier is used to identify which label a pair belongs to based on the embeddings. The solution is evaluated by first testing whether it can correctly classify pairs of binaries as EQ or NEQ, and then whether it can identify the subsets of EQ and NEQ.

Chapter 2

Background

2.1 Software Builds and Supply Chains

What does a software build environment look like? Turning source code into executable binaries may involve compiling, linking, fetching dependencies, testing, publishing, and more. Continuous integration and continuous deployment (CI/CD) products supplied by vendors such as GitLab and Jenkins are often used to automate these steps. Build environments can become complex, with many components to manage. Ensuring that every component of a software build environment is trusted can be challenging.

Significant work has been done to improve the security and assurance of software supply chains. The concept of a 'software bill of materials' (SBOM) has been developed collaboratively and is currently mandated for software provided to the United States Food and Drug Administration [10, 11]. An SBOM lists all dependencies of a software application, allowing users to see whether the software contains any potentially harmful components. Another measure to improve supply chain security is the Supply-chain Levels for Software Artifacts (SLSA) framework, which is an industry collaboration including enterprises such as Google, Intel and the Linux Foundation [12]. It aims to establish guidelines that can mitigate a wide range of supply-chain attacks, such as unauthorised source code changes and the use of compromised dependencies. Of particular interest to us are efforts to improve assurance for pre-built binaries, which is the primary focus of the Reproducible Builds [13] project, Google's Assured Open Source Software [3] service and Oracle's build-from-source project.

2.2 Verifiable Builds

A software build is considered verifiable if it includes sufficient information to reproduce it exactly, or near-exactly with any differences explained. This property allows independent parties to reproduce the binary and check whether the output matches expectations. It is intended to provide assurance that the binary is a true representation of the source code [7, 14].

2.2.1 Deterministic Builds

The primary approach towards verifiable builds is to make the build process deterministic, meaning that a specific source code input must always produce an identical binary output. However, there are barriers to implementing this. Build tools may exhibit non-deterministic behaviour or incorporate non-deterministic information directly into the binary; timestamps are a notorious source of non-determinism in software builds [14]. One

report describes some of the difficulties encountered by organisations implementing deterministic build pipelines [7]. The authors studied three large-scale commercial systems within Huawei. Some challenges they identified were the large resources needed to locate and resolve sources of non-determinism, non-determinism imposed by security mechanisms such as digital signatures, and non-determinism arising from third-party dependencies which were not within the organisation's control.

The Reproducible Builds project is a communal project that promotes deterministic build processes as a way to improve supply chain security. It aims to address attacks on build infrastructure through verifiable builds. Its website states that 'First, the build system needs to be made entirely deterministic: transforming a given source must always create the same result.' [13] However, producing deterministic builds is not simple. The website goes on to state that the build environment must be documented sufficiently so that a third party could recreate it: 'Second, the set of tools used to perform the build and more generally the build environment should either be recorded or pre-defined. Third, users should be given a way to recreate a close enough build environment, perform the build process, and validate that the output matches the original build.'

Given the above process, an independent party can verify a binary by setting up a build environment similar to the original build environment and checking whether it produces the same binary. This assumes that the independent party has access to build resources equivalent or 'close enough' to those used for the original build. However, there is still room for security holes; for example, in a situation where the original build environment includes a security flaw, and the verification environment reproduces that flaw. In this situation, a build could be verified without identifying the security flaw.

2.2.2 Explaining Non-Equivalences

A complementary approach to verifiable builds is to accept non-equivalences that can be explained and documented [7]. This approach deals with sources of non-determinism that are impractical or impossible to mitigate by creating provenance to track and record them. However, human judgement is needed to decide which non-equivalences should be allowed and explain the rationale for their acceptance. Identifying and explaining the root causes of non-equivalences can be difficult: one case study encountered non-determinism arising from the brand of CPU in the build machine [14]. Such cases make this approach difficult to automate and present scalability problems.

2.2.3 Verifiable Builds in Practice

The authors of [15] created a process for producing verifiable builds for Java. Their approach was to eliminate sources of non-determinism and explain any sources that could not be eliminated. This approach required specialised software, as well as human input to diagnose and explain the non-equivalences. A similar approach had been previously developed for C/C++, but could not be immediately applied to Java binaries due to Java's different sources of non-determinism and the different mitigations required to address them. For example, instruction interception techniques that operated at the kernel level had to be adapted to work on the Java Virtual Machine.

The authors applied their process to software available from the Reproducible Builds project. However, they were not able to produce verifiable builds for all of the items they tested due to non-determinism arising from third-party libraries.

2.2.4 Identifying Binary Equivalence

Approaches relying on deterministic builds present implementation difficulties that may place them beyond the reach of some software providers. Additionally, they do not address the replication of security flaws that could occur through the replication of build environments, or the fact that many pre-built binaries have already been distributed without verification in mind. The idea of binary equivalence provides an alternative to relying on deterministic builds.

Large language models have the potential to compare binaries without relying on a byte-by-byte analysis. They are capable of operating at high levels of abstraction, which may allow us to accept the non-deterministic nature of software builds and test for binary equivalence. This approach could provide an option for assessing binaries built without consideration for determinism or provenance, such as ad hoc and historical builds.

2.3 Large Language Models for Bytecode Classification

Large language models are examples of complex neural networks. They are trained on large amounts of data and demonstrate the ability to learn and understand abstract concepts. For example, Google's T5 model can translate between natural languages such as English and French [16]. If we consider abstraction as a way of reaching a concept regardless of its exact expression, then the ability to translate between languages demonstrates this: the meaning of a translated message remains the same, but its expression is changed. The capacity for abstraction is useful to us because we want to identify binary equivalence regardless of the exact bits and bytes used to express it.

Internally, an LLM expresses meaning in numerical form. The neural network maps aspects of the input to different dimensions, creating a high-dimensional vector. These vectors, referred to as 'embeddings', can be extracted from the LLM and used for further machine learning operations. For example, a vector representation enables distance calculations, which can be used to measure how similar two inputs are while disregarding the exact details of their original representations.

LLMs have been successfully applied to a variety of language-related domains, including the understanding of computer programming languages. LLMs specialised for source code, sometimes called 'code understanding' models, can be used to generate code based on natural-language requests and produce summaries of source code [17]. However, our question is whether a binary accurately represents its source code. This must be answered by inspecting the binary, not the source code.

Decompilation techniques can be used to recover source code from compiled binaries, but they cannot reliably restore the original text [18]. Source code contains information (such as comments, compiler directives and high-level code structures) that may be permanently lost during compilation. Additionally, transforming the source code to an executable involves processes, such as compiling and linking, that may make unauthorised changes to the output [1].

2.3.1 Preparing Built Binaries for Machine Learning

How can we prepare a binary for a machine learning pipeline without the source code? While it would be possible to analyse a binary byte-by-byte, raw bytes lack the semantic information that source code contains.

Bytecode is an intermediate representation that lies between source code and raw machine instructions. It is often used to run programs in a virtual machine instead of directly

on the hardware. Programming languages such as Java and C# use bytecode to make their built binaries more portable by removing the need to build different binaries for different varieties of hardware. Bytecode contains more semantic information than raw bytes: for example, it has instructions for specific operations such as integer addition.

Java bytecode fully captures the behaviour of the binary and can also be converted to text for input into a code understanding model. The Java Development Kit includes the tool 'javap', which can be used to disassemble a Java class. The `-c` flag can be used to obtain the bytecode representation of a Java class as human-readable text [19]. The output does not include private members by default, but they can be included by using the `-p` flag.

Two examples of javap output are shown in listings 2.2 and 2.3. These listings show the bytecode representations of the string concatenation function shown in listing 2.1, which has been compiled with two different versions of the Java compiler and disassembled with the command `javap -c` (the `-p` flag is not needed here because there are no private members). We would expect our machine learning solution to classify these two bytecode documents as EQ because they represent equivalent behaviour. However, we can see that the bytecode implementations are quite different: the Java 17 version makes a dynamic function call to 'makeConcatWithConstants' instead of using a StringBuilder to concatenate the strings.

Embedding provides a way of converting raw input, such as text, to a more suitable format for further machine learning operations. The embedding process can be applied to the output of javap to produce an embedding vector that represents a Java class. Once computed, the embedding can be stored for later use.

Listing 2.1: Java source code for listings 2.2 and 2.3.

```
class Concatenator {
    public String concat(String x, String y) {
        return x + y;
    }
}
```

Listing 2.2: Bytecode representation of listing 2.1 compiled with Java 17.

```
public java.lang.String concat(java.lang.String, java.lang.String);
  Code:
     0: aload_1
     1: aload_2
     2: invokedynamic #7,  0 // InvokeDynamic #0:
        makeConcatWithConstants:(Ljava/lang/String;Ljava/lang/String
        ;)Ljava/lang/String;
     7: areturn
```

Listing 2.3: Bytecode representation of listing 2.1 compiled with Java 8.

```
public java.lang.String concat(java.lang.String, java.lang.String);
  Code:
     0: new           #2      // class java/lang/StringBuilder
     3: dup
     4: invokespecial #3      // Method java/lang/StringBuilder."<
        init>":()V
     7: aload_1
     8: invokevirtual #4      // Method java/lang/StringBuilder.
        append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
    11: aload_2
    12: invokevirtual #4      // Method java/lang/StringBuilder.
        append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
    15: invokevirtual #5      // Method java/lang/StringBuilder.
        toString:()Ljava/lang/String;
    18: areturn
```

2.3.2 Applying Machine Learning to Bytecode Embeddings

Pre-trained models specifically for bytecode are not publicly available to the best of our knowledge. However, there are abundant resources for applying LLMs to source code, and models trained on source code show promise. The BinEq dataset has been used previously to conduct machine learning classification experiments; for example, in [20] pairs of binaries were classified as EQ or NEQ using two different GPT models from OpenAI, as well as neural networks CodeT5+ and CodeLLaMA with encouraging results. GPT-4 scored remarkably well despite not being specialised for code, which the author attributed to 'structural similarities between bytecode and other assembly languages.' This shows that machine learning models do not need to be specifically trained for bytecode in order to produce useful results. There is room to expand on this research by exploring other machine learning models and techniques, such as different classification methods, and by identifying the nature of binary non-equivalences by identifying the subsets of NEQ.

Related research includes the application of machine-learning techniques to JavaScript bytecode to identify unwanted behaviours such as tracking and advertising [21]. While this research addresses a different problem than ours, there is a clear parallel in using bytecode to infer information about a program's functionality. In their case, they wished to identify whether a program exhibited a particular type of behaviour, while we wish to identify whether two programs exhibit equivalent behaviour. Their use of artificial intelligence for both embeddings and classification validates our approach.

In summary, recent works show that machine learning techniques, such as using large language models to generate embeddings, can be successfully applied to bytecode for the purpose of classifying program behaviour.

Chapter 3

Methodology

This project aims to improve software supply chain assurance by identifying binary equivalence. To do this, a code understanding model is used in combination with a classifier to classify compiled Java classes as equivalent or non-equivalent. The work has been carried out in incremental stages and tests the ability of the code understanding model to identify equivalence for three sets of code: raw source code, bytecode obtained from a small hand-crafted set of Java classes and bytecode obtained from a large dataset made up of open-source Java libraries. The general process was to obtain the bytecode representation of each Java class using the javap tool, generate embeddings using the code understanding model, then classify pairs of binaries as EQ or NEQ based on the embeddings. This chapter describes the specific steps used to validate the process and produce the final results.

3.1 Datasets

The BinEq dataset was the primary dataset used for evaluation. However, due to its complexity and size, a smaller, hand-crafted dataset was created to be used while developing a proof-of-concept.

Toy Dataset: A small dataset was created to challenge the machine learning model in specific ways. The toy dataset included examples of four simple programming tasks: integer addition, integer comparison, string concatenation and conditional branching. These tasks were chosen to show the proof-of-concept's ability to discern the meaning of code structures in context. For example, a single operator can be the difference between integer addition and integer comparison, but they are fundamentally different tasks and should always be classified as NEQ. Each task had several different Java implementations, written so that some would have equivalent behaviour and others would not.

A set of examples used for testing is shown in table 3.1. Integer overflow is a common source of software bugs where numbers that grow larger than the maximum integer value 'wrap around' to the minimum value. The Java code in rows 1, 2 and 3 will exhibit this behaviour. However, the code in row 4 makes use of a function that detects integer over-

	Java code	**Relationship**
1.	`x + y;`	Base integer addition code
2.	`y + x;`	Equivalent code with operands reversed
3.	`Integer.sum(x, y);`	Equivalent code using a method call
4.	`Math.addExact(x, y);`	Non-equivalent code: ArithmeticException may be thrown

Table 3.1: Java snippets that may exhibit integer overflow.

	1.	2.	3.	4.
1.	EQ	EQ	EQ	NEQ
2.	-	EQ	EQ	NEQ
3.	-	-	EQ	NEQ
4.	-	-	-	EQ

Table 3.2: Expected classification of paired Java snippets in table 3.1.

flow and throws an ArithmeticException if it is encountered. Table 3.2 shows the expected classification of these examples. We expect our solution to classify examples 1, 2 and 3 as EQ to one another, but NEQ to example 4.

BinEq 1.1.0 Dataset: The BinEq dataset contains real Java classes obtained from open-source repositories. Pairs of classes are grouped as EQ, NEQ1, NEQ2 and NEQ3 [9].

The three NEQ groups represent different types of behavioural non-equivalence. NEQ1 contains pairs differing due to breaking API changes, NEQ2 contains pairs with differences applied by a mutation testing framework, and NEQ3 contains pairs differing due to vulnerability patches. The NEQ3 group is particularly relevant to the problem of identifying binaries with malicious inclusions.

The EQ group can be broken down into pairs built with the same compiler and major version number, pairs built with the same compiler but with different major versions, and pairs built with different compilers. Building classes with the same compiler makes it easier to compare the resulting binaries and aligns with the practice of reproducible builds, which encourages us to use similar build environments. Such builds provide strong evidence to support the assertion that two different binaries originate from the same source code. Building classes with different compilers makes them harder to compare but provides a different type of assurance. From a security point of view, proof that a compiler produces EQ binaries only leads us to ask whether the compiler is good or not. It does not provide any assurance that the compiler is not malicious. However, producing EQ binaries from different compilers does provide this assurance to some degree, simply because the small likelihood of two different compilers being compromised in the same way.

3.2 Inception

Initial experiments were conducted using source code instead of bytecode. The Salesforce codet5p-110m-embedding model was used to generate embeddings for Java source code files. Pairs of embeddings were then compared to each other using the cosine distance between the embedding vectors as a measure of similarity.

CodeT5+ is a family of pre-trained large language models from Salesforce. They are capable of a variety of programming tasks such as code generation, autocompletion and summarisation [17]. The codet5p-110m-embedding variant can be used to obtain embeddings rather than code outputs. CodeT5+ models are trained on nine different programming languages using publicly available source code obtained from GitHub: the point of this exercise was to establish the model's ability to identify equivalent behaviour using the medium it was trained in which is source code, not bytecode.

An early version of the toy dataset was used for the source code experiments. The dataset contained 18 code snippets with examples of addition, conditional branching and integer comparison. An embedding was generated for each file, then each embedding was paired with every other embedding for a total of 152 pairs ($18^2/2 = 152$).

Cosine similarity can be used to measure the similarity of text documents [22]. It is based

on cosine distance, which measures the cosine of the angle between two vectors. Small angles show that the vectors are pointing in similar directions. In the high-dimensional vector space represented by the embeddings, small angles show that the vectors have been assigned similar meanings by the embedding model. Cosine similarity inverts cosine distance so that values closer to 1 show greater similarity. This method was selected to provide a simple means of measuring the similarity of source code documents. Combining this measurement with a threshold would provide a way to classify pairs of documents for proof-of-concept purposes, with the intention of switching to a more sophisticated method later.

All the pairs were ranked by their cosine similarity scores. The results formed three groups. The most similar group, with scores ranging from 0.96 to 0.99, were pairs implementing the same task in slightly different ways. Examples of the differences include swapping the order of operands in a comparison, choosing to implement an 'else' branch as a ternary statement, and explicitly throwing an exception that would have been thrown implicitly anyway. The next most similar group of items had scores ranging from 0.91 to 0.95. It contained pairs implementing the same programming task, but with subtle differences in functionality: primarily that an exception could be thrown by one snippet but not the other. These exceptions would arise either from integer overflow checking (see section 3.1 for an example of this) or attempting to call a function on a null object. These pairs exemplify similar, but non-equivalent behaviour. The remaining pairs occupied a range well below the others (0.65-0.73) and consisted of pairs that did not implement the same programming task.

Using these rankings, the items could be successfully classified as EQ or NEQ by applying a threshold. All pairs scoring 0.96 or higher in cosine similarity would be classified as EQ, and all others as NEQ.

Similarity score range	**Relationship between pairs**
0.96-0.99	EQ
0.91-0.95	NEQ implementing the same tasks
0.65-0.73	NEQ implementing different tasks

The source code test showed that the embeddings produced by the Salesforce model contained useful information about program behaviour. The information captured in the embeddings was sufficient to rank the similarity of source code pairs in line with our expectations. The source code contained extremely simple functions and did not approach the complexity of programs in the BinEq dataset. However, we saw that the code understanding model could be used to distinguish between EQ and NEQ pairs, including pairs that were NEQ due to reasonably subtle differences.

3.3 Proof-of-Concept

Following on from promising source code tests, a proof-of-concept was needed to validate a bytecode-based approach and identify any potential issues as early as possible. The requirements were to take input generated by using tools such as javap, use a large language model to transform the inputs into embeddings, and attempt to characterise the input based on those embeddings.

The proof-of-concept operated on the toy dataset, not the BinEq dataset. The test platform was a Windows 10 machine with a Ryzen 7 3700X processor, 16GB of memory and a GTX 1080 graphics card.

Disassembly: The toy dataset consisted of 29 Java source code files, each containing a single class with a single method. To prepare for bytecode-based testing, these files were compiled and disassembled with javap.

Three of the 29 files were excluded from testing because they contained deliberate syntax errors and could not be compiled. Each of the 26 remaining files was compiled, then disassembled inside a Docker container using `javac` followed by `javap -c`. Three different versions of the Java compiler were sourced from OpenJDK Docker[1] images: openjdk8, openjdk11 and openjdk17. Java versions 8, 11 and 17 were chosen because they are well-established and commonly used due to having long-term support [23]. The use of multiple versions mirrored the use of multiple Java compilers in the BinEq dataset, though on a smaller scale. This was important because different versions of the Java compiler can produce substantially different bytecode documents from the same source code, as demonstrated in listings 2.2 and 2.3, but the proof-of-concept was expected to be able to identify them as EQ.

The three Docker containers were used to compile and disassemble three javap documents for each class. Each javap document was paired with every other document to create an overall dataset of 6,084 pairs ($26 \times 3^2 = 6,084$). Pairs of bytecode documents were labelled EQ if we expected them to have equivalent behaviour, otherwise they were NEQ. An example of equivalent behaviour is when different source code is used to express the same functionality as described in section 3.1. Another example is when the exact same source code is compiled with different versions of Java because Java compilers may use different strategies to accomplish the same behaviour. Different Java compilers were also used to create EQ pairs in the BinEq dataset.

Embeddings: These were generated using the Salesforce codet5p-110m-embedding model available on Hugging Face [17]. The model had been trained on source code in various programming languages, including Java. It was not known to have been trained on Java bytecode specifically, but the CodeT5+ family of models had been used to classify bytecode with some success [20]. Because the toy dataset's source code files and resulting javap documents were short, there was no need to consider any input limit imposed by the embedding model.

The embedding step was seen to produce different outputs on different machines. This was observed when generating embeddings on a Windows desktop, Mac laptop and Linux lab machine. When embeddings were repeatedly generated on a single machine the outputs were the same. For this reason, a single platform was used to generate all embeddings.

Classification: A simple classifier was created by combining the cosine similarity measure used in the previous experiment with a threshold value to determine whether two embeddings were similar enough to be classified as EQ.

3.3.1 Proof-of-Concept Outcome

To test the proof-of-concept, the embedding model was used to generate embeddings for each javap bytecode document in the toy dataset. Each pair of embeddings was classified as EQ or NEQ by computing the similarity of the two embedding vectors and comparing it to a threshold value. The following table shows the results of classifying 6,084 pairs with three different threshold values. A threshold of 0.97 results in no false positives (Incorrect EQ). We see that as the threshold value decreases the percentage of correctly classified items increases, but so does the number of false positives.

Threshold	Correct EQ	Incorrect EQ	Correct NEQ	Incorrect NEQ	Total Correct
0.97	406	0	5,202	476	92.18%
0.96	424	18	5,184	458	92.18%
0.95	480	36	5,166	402	92.80%

[1]OpenJDK on Docker Hub: https://hub.docker.com/_/openjdk/tags Accessed 2024-10-12

The most successful threshold value overall was 0.95 with 92.8% of pairs correctly classified. However, this threshold produced more false positives than the others.

Despite using a relatively unsophisticated method of classification, the proof-of-concept was able to correctly classify up to 92.8% of pairs. This showed that the embeddings captured enough meaningful information from the Java bytecode to enable this result. However, the test data was contrived and not sufficiently complex to represent real-world scenarios.

For all three threshold values, incorrectly classified pairs were always in the same functionality group (integer addition, integer comparison, string concatenation or conditional branching). This suggested that the proof-of-concept could reliably identify when pairs addressed the same general programming task, but had difficulty with the subtle differences present in various implementations of the task.

Overall, the proof-of-concept demonstrated the ability of a code understanding model to capture meaning from Java bytecode, and several opportunities for improvement were identified. The next step was to adapt the proof-of-concept to use the BinEq dataset, which also meant adapting the process used to generate embeddings. Additionally, the cosine distance classifier would be swapped for a more sophisticated one, which would make it easier to perform multiclass classification.

3.4 Application to BinEq Dataset

Upon completing the proof-of-concept and receiving encouraging output, the next step was to adapt it to operate on the BinEq dataset. This dataset contained hundreds of thousands of records drawn from real open-source projects. The scale and complexity of this data compared to the toy data presented some performance challenges related to the generation of javap documents and embeddings.

3.4.1 Data preparation

Disassembly: The BinEq dataset contained pairs of Java classes generated by a variety of compilers. The process for the proof-of-concept had been to use Docker containers to compile and immediately disassemble class files. This meant that the javap version used for disassembly always matched the compiler. Maintaining this approach with the BinEq data would mean identifying and obtaining the correct version of javap for each class, a potentially error-prone process. To simplify disassembly, this approach was abandoned in favour of using a single Java installation (openjdk 17.0.11) to run javap on the entire dataset. This treated all items the same way and had an additional performance benefit of removing the overhead of running various Docker containers.

The command `javap -c -p` was used to disassemble all classes referenced by the BinEq dataset. The `-p` flag was used so that private members would be included in the output. This was important to allow the code understanding model to identify non-equivalences implemented in private classes or methods. The resulting output was saved to a file, referred to from here as a javap document.

Model input limit: One of the main challenges was that the CodeT5+ model used for embeddings only accepted up to 512 tokens as input. This was anticipated to become a problem when moving from the toy dataset to the BinEq dataset, where tokens would number in the tens of thousands. A simple solution would have been to swap the CodeT5+ model for a more generous model that had also been trained on code. However, any model (supposing a suitable one was found) could reasonably be expected to impose some sort of input limit. Since Java classes can be arbitrarily long, this would only ever provide a partial solution.

One possibility was to reduce the overall size of the javap documents by removing redundant or irrelevant information. Javap documents were 'minified' by removing information that was implied elsewhere in the bytecode document. Listing 3.1 shows a method call represented in Java bytecode. The instruction 'invokevirtual' is followed by a constant pool reference '#5' and a phrase including a descriptor of the method's fully qualified name and return type. However, the text '// Method' contains information that can be inferred from context; for example, the 'invokevirtual' instruction tells us that this is a method call, so the word 'Method' may be considered redundant. This is an example of information that was removed from the bytecode documents.

Listing 3.1: Bytecode representation of a method call

```
15: invokevirtual #5                  // Method java/lang/StringBuilder.
    toString:()Ljava/lang/String;
```

However, during test runs with the classifier this had an overall negative effect on the classification accuracy. This effect suggested that the redundant information was providing some value to support the model's understanding, so the practice of removing text from javap files was abandoned.

Another possibility for reducing the size of bytecode documents was to pre-process them by breaking them up into small chunks. This would require the resulting embeddings to be combined somehow but could address the problem of overly large input at the cost of some loss of cross-chunk context. It was tested by splitting the bytecode documents based on method definitions. This was intended to avoid splitting in the middle of code structures where context would seemingly be more important. However, methods can also be arbitrarily long, and it was soon found that this did not reliably produce small enough units to use as input for the model.

The CodeT5+ model had an associated a tokeniser that was used to prepare any input by converting it into tokens. This tokeniser provided an option to return a specific number of tokens at a time. We used the tokeniser to prepare each javap document as a sequence of 512-token chunks. Each of these chunks was input into the code understanding model to produce one embedding vector. Finally, the vectors were combined by calculating their arithmetic mean, which is a way to combine information from multiple embeddings without increasing their dimensionality [24]. This resulted in a single embedding representing the full javap document: figure 3.1 visualises the process. Compared to the approach used for the proof-of-concept, where only the first 512 tokens were considered, this approach was more appropriate for the BinEq dataset. This is because the entirety of each javap document was used to generate the embedding, rather than just the first 512 tokens. The 512-token limit was acceptable when using the toy dataset, which consisted of short javap documents generated from Java classes that were written specifically for experimentation. Comparing embeddings based on only the first 512 tokens is obviously problematic in cases where differences exist beyond the 512-token boundary. The BinEq dataset consisted of Java classes taken from open-source repositories, which were not written with our use case in mind and in many cases went well beyond the 512-token limit.

Embeddings: Generating embeddings was the most time-consuming and resource-heavy step in the process. Once generated, embeddings could be stored and re-used without having to regenerate them. However, changes to the embedding generation process (such as supplying different parameters to the tokeniser) did require new embeddings to be generated for each bytecode document.

A significant challenge of calculating multiple embeddings per bytecode document was that the total number of embeddings being generated was much higher than before, because

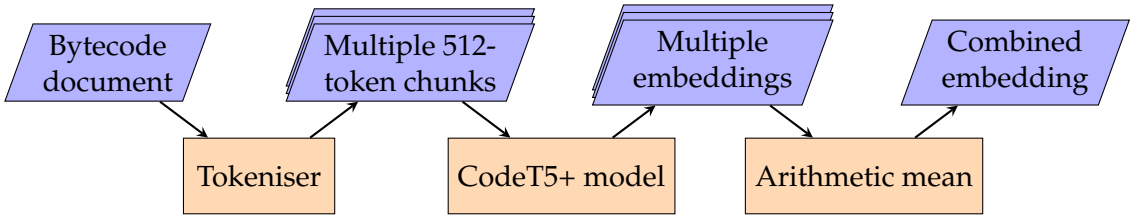

Figure 3.1: Flow chart showing the process of using a code understanding model to create an embedding for a Java bytecode document.

some classes required hundreds of embeddings to represent them. Many machine learning operations benefit from the use of a graphics card (GPU or Graphics Processing Unit) due to the GPU's ability to perform calculations in parallel and the prevalence of machine learning libraries that make use of this capability. This comes at the cost of having to send data to the GPU and back, but that is generally outweighed by the gain in performance. A GTX 1080 graphics card was used to reduce the time taken to generate the embeddings for each javap document. Once generated and combined, the final embeddings were stored to avoid the need to re-generate them.

It had been noted earlier that different embeddings could be produced from the same input depending on the environment in which the model ran. For this reason, all embeddings were calculated in the same way using the same hardware and operating system. This would prevent any irregularities arising from differences between machines and the resulting embeddings.

The CodeT5+ model's tokeniser provides the option to set a 'stride' value. When splitting an input into tokenised chunks, the stride can be used to help preserve cross-chunk context by including some surrounding context (overlapping tokens) in each chunk. Finding a suitable value for the stride option is a question in itself and the cost of generating embeddings would have made this relatively expensive to experiment with, so it was left unused. The results of this project could likely be improved through use of the stride option.

3.4.2 Classification

The proof-of-concept used a cosine similarity calculation in combination with a threshold value to classify pairs of binaries as EQ or NEQ. While this was sufficient to demonstrate preliminary results, the BinEq data contained more complex data and more granular labels, requiring a more sophisticated classifier.

Random forest is an ensemble classification algorithm, meaning that it combines multiple classifiers to determine an output. It works by generating decision trees based on randomised subsets of the data, then taking a 'majority vote' from among them [25, 26]. Compared to single decision trees, random forests have better overall classification accuracy and are less likely to overfit. This is because errors made by individual trees are balanced out by the ensemble. Random forests are robust to noise and outliers and also reasonably quick to train, making them a practical choice for various applications. The implementation from the scikit-learn library was used for classification [27].

An issue when moving from the cosine similarity calculation to any other method was that the classifier had to operate on pairs of embeddings. This was straightforward when calculating cosine similarity, but there was the question of how to combine embeddings so they could be treated as a single input. Possibilities included summing or concatenating the embeddings. Another alternative was to use a 'Siamese' architecture where the two embeddings were treated separately until the last step, where a contrastive loss function was

used to arrive at the final outcome. The approach had been used with success [20]. During development, the concatenation approach was taken because it was simple and avoided the potential data loss incurred through summing. This was implemented using NumPy's hstack function [28].

The random forest is an example of a supervised learning algorithm, meaning it must be trained on labelled data before use. Initial classification runs were performed by splitting the dataset into testing and training sets using an 80:20 ratio. The training set (80%) was used to train the random forest, and the testing set (20%) was used to evaluate its performance. Withholding a percentage of the dataset for evaluation purposes is a best practice in machine learning [29]. While it reduces the total amount of data available for training, it allows us to understand how well the model can generalise to unseen data. It is possible for a classifier to suffer from overfitting problems where it effectively recognises specific instances of the data and what the associated labels are, instead of learning patterns in the data and basing classification decisions on those. Such a model will be less effective at classifying data that is not in its training set. Withholding the test data from the model until evaluation time is a way of revealing such problems by showing how it performs on previously unseen data.

Label	**Training Set (80%)**	**Testing Set (20%)**	**Total Pairs**
EQ	14,404	3,596	18,000
NEQ	9,757	2,445	12,202
Total	24,161	6,041	30,202

Table 3.3: Example breakdown of dataset split into training and testing data (80:20).

Parameters for the random forest classifier were identified using a grid search. This technique searches a grid of options to find the set of parameters that produces the best outcome. A random seed parameter of 489 was used for repeatable results.

Data selection: The BinEq dataset contains 622,029 labelled pairs of classes. Due to the expense of calculating embeddings, only a small selection of the data was used. However, the selection was complicated by the fact that the subgroups within the dataset were imbalanced. Notably, the NEQ3 group contained only 202 pairs. It was vastly outweighed by the other groups: EQ with 465,858 pairs, NEQ1 with 14,384 pairs and NEQ2 with 141,585 pairs. An initial attempt to classify 10,000 randomly selected pairs resulted in zero instances of NEQ3 appearing in the test data. Since the NEQ3 group concerns vulnerability patches, it is particularly relevant to the issue of supply chain attacks. It is the group that most closely represents the scenario where malicious behaviour is introduced through the build environment. For this reason, it was important for this group to be represented in the selected data.

Consideration was also given to the subsets within the EQ group. As described previously, the EQ group contains pairs of Java classes built with compilers of varying similarity to each other. The classifier was expected to be able to easily identify pairs built with the same compiler as EQ. Identifying this relationship between two built binaries would provide some assurance that both originated from the same source code. However, the ability to identify binaries built with different compilers as EQ would provide a different type of value. For example, in cases where the integrity of a compiler is in question, verifying that it produces two EQ binaries is of little value compared to verifying that it produces a binary EQ to a known good (or even just separate) compiler. This is also more valuable in situations where the exact build environment used to produce a particular binary is unknown or not reproducible.

The final data selection was done by taking all 202 NEQ3 pairs and equal numbers from the other NEQ sets and three subsets of EQ. The makeup of the dataset is presented in table

Data subset	Number of pairs
EQ-SameMjCompVer	6,000
EQ-SameComp	6,000
EQ-DiffComp	6,000
NEQ1	6,000
NEQ2	6,000
NEQ3	202
Total	**30,202**

Table 3.4: Makeup of the dataset used for evaluation.

3.4. Aside from NEQ3, pairs were selected in a pseudo-random way by using a seeded random number to order all pairs before taking a specific number of pairs starting from a specific index. This approach was intended to randomise the selection but allow the dataset to be grown by taking further selections without selecting the same data twice.

Cross-validation: To address the imbalanced representation of the NEQ3 group, stratified 10-fold cross validation was used. K-fold cross validation is a way of splitting data into testing and training sets so that all data is used for both training and testing (though the training and test sets are always kept separate). In 10-fold cross validation, each 'fold' is generated using a 90:10 split for training and testing. The part that makes up the 10% testing data is different for each fold. The model is trained and tested on each fold for a total of ten runs: effectively, ten instances of the model are trained and evaluated. The final results are reached by averaging the results from all ten instances. This practice helps combat problems such as overfitting that can be less obvious when the test data does not vary. Stratification modifies K-fold cross validation so that classes are distributed evenly across the folds. This was done to ensure that NEQ3 data would be included in a similar amounts in every test set.

10-fold cross validation was trialled against 5-fold cross validation, which uses an 80:20 split for testing and training data. Providing more training data can generally be expected to improve the performance of a classifier, but this comes at the cost of limiting the amount of testing data and potentially the accuracy of performance scores. In this case the amount of data could simply be increased except for the NEQ3 label, for which only 202 instances were available. Since the classifier's performance on NEQ3 was relatively poor, using a 90:10 split was hoped to improve it by maximising training data. 10-fold cross validation was found to provide only minor improvement in accuracy over 5-fold cross validation.

3.4.3 Metrics

To get a more detailed view of the classifier's performance, the standard performance metrics of precision, recall and F_1 score were calculated. Precision and recall are closely related metrics that are calculated separately for each label. Each prediction made by the classifier is considered 'positive' if it predicts that label and 'true' if it is correct.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

Precision measures the classifier's ability to assign a specific label correctly; it asks, when the classifier makes a prediction, how often is it actually correct? A high precision score

shows that there are few false positives. Recall measures the classifier's ability to identify as many instances of a label as possible from among the other data points. A high recall score shows that there are few false negatives. Precision can be inflated by being more selective about assigning a label, but this can result in a lower recall score. Recall can be inflated by assigning a label more generously, but this can result in a lower precision score.

F_1 score is the weighted harmonic mean of precision and recall. A high F_1 score shows that a model is good at identifying both positive and negative cases. It is a metric that assumes we aim to maximise both precision and recall equally. The generalised form of the F_1 score is the F-beta score, where the value of β sets the balance between precision and recall [30]. We can favour precision by setting the β value (usually 1) less than 1, or to favour recall by setting it greater than 1.

$$F_\beta = \frac{(\beta^2 + 1) \cdot Precision \cdot Recall}{\beta^2 \cdot Precision + Recall}$$

Modifying β can provide a way to report certain labels more conservatively. A relevant scenario where this would be beneficial is in creating tools that raise alerts for human consideration. In this scenario, reporting too many false positives would reduce the user's trust in the tool and possibly result in them ignoring it, so it may be beneficial to let some false negatives slip through in order to retain the user's trust. This could be done by using a model with a high $F_{0.5}$ score, which prefers precision over recall by setting β to 0.5.

Chapter 4

Results

The classifier's performance was evaluated across several tasks. A binary classification task was used to test its ability to distinguish EQ from NEQ, and a multiclass classification task was used to test its ability to identify the different subsets within EQ and NEQ. Sequential tasks were used to identify NEQ pairs within the dataset, then extract them and test performance on classifying them into the subsets of NEQ. The same dataset was used throughout.

4.1 Binary Classification

The primary test was a binary classification task to label pairs of Java classes as either EQ or NEQ. This addresses the matter of detecting binary equivalence. The results are shown in table 4.1 and figure 4.1.

The classifier was able to correctly identify most pairs of classes, achieving an overall accuracy score of 92%. This shows that machine learning techniques can be applied to Java bytecode to identify binary equivalence with reasonable accuracy.

Label	Precision	Recall	F_1	$F_{0.5}$
EQ	0.94	0.94	0.94	0.94
NEQ	0.91	0.90	0.91	0.91

Table 4.1: Scores for the binary classification task: EQ vs NEQ.

4.2 Multiclass Classification

For a more granular test, the same dataset was used but with six different labels identifying the subsets of EQ and NEQ. This was a more difficult task that asked the classifier to identify different varieties of equivalence, such as whether pairs were compiled with the same compiler and major version number. As could be expected, the overall accuracy score was much lower at only 69%. However, this gave us more insight into what types of differences were easier or harder for the classifier to identify. Table 4.2 and the confusion matrix in figure 4.2 show the results from this test. We see that NEQ3 items are often labelled wrongly: the confusion matrix shows that the classifier could not easily distinguish between the different subsets of EQ. Its performance on the NEQ1 and NEQ2 subsets remained reasonably high, but the recall score for NEQ3 was very low, meaning that most NEQ3 instances were misclassified as something else.

Figure 4.2 shows that the classifier performed best on the NEQ1 (API changes) and NEQ2 (mutation testing) groups. As could be expected, it was relatively poor at distinguishing

Label	Precision	Recall	F_1	$F_{0.5}$
EQ-DiffComp	0.50	0.47	0.48	0.47
EQ-SameComp	0.52	0.47	0.49	0.51
EQ-SameMjCompVer	0.65	0.66	0.65	0.65
NEQ1	0.84	0.97	0.90	0.87
NEQ2	0.91	0.91	0.91	0.91
NEQ3	0.95	0.10	0.18	0.35

Table 4.2: Scores for the multiclass classification task: subsets of EQ and NEQ.

between the different EQ labels. For our research problem we do not value the ability to distinguish between subsets of EQ as highly as the ability to distinguish between EQ and NEQ. For security uses there may be additional value in being able to distinguish between subsets of NEQ, particularly NEQ3 (vulnerability patches), but the classifier often failed to identify items as NEQ3.

In the multiclass classification task, the NEQ3 pairs were more often assigned one of the EQ labels than their correct label. This can be attributed to the lack of training data for NEQ3 compared to the other groups. Because there are not many instances of this label, the classifier does not have as many opportunities to learn patterns associated with it and how it differs from the other labels. This could be addressed through collecting or generating new NEQ3 instances; however, NEQ3 data is collected through manual processes that make it particularly expensive to acquire.

4.3 Sequential Classification

Another test was run on the output of the binary classification test (EQ vs NEQ), to take the identified NEQ pairs and further classify them as NEQ1, NEQ2 or NEQ3. This effectively reduced the multiclass task to only three possible labels, which was expected make it easier for the classifier. However, its performance on NEQ3 was not greatly improved, with a large portion of NEQ3 items being filtered out at the binary classification step. Table 4.3 and figure 4.3 show the metrics from this test. In figure 4.2, which shows the results of the multiclass task with all six labels, we see that the NEQ3 group was often misclassified as one of the EQ labels. Figure 4.3 shows that when the EQ labels are not in consideration, the NEQ3 pairs are still misclassified. Even though the binary classification task could be considered easier than the multiclass task, NEQ3 pairs were classified (or misclassified) in similar numbers.

Label	Precision	Recall	F_1	$F_{0.5}$
NEQ1	0.88	1.00	0.93	0.90
NEQ2	0.94	0.99	0.97	0.95
NEQ3	0.94	0.43	0.59	0.76

Table 4.3: Further classification of pairs identified as NEQ.

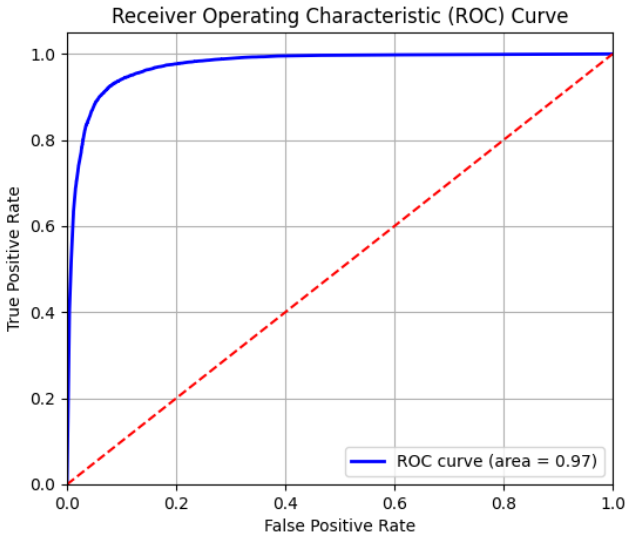

Figure 4.1: ROC curve for the binary classification task: EQ vs NEQ.

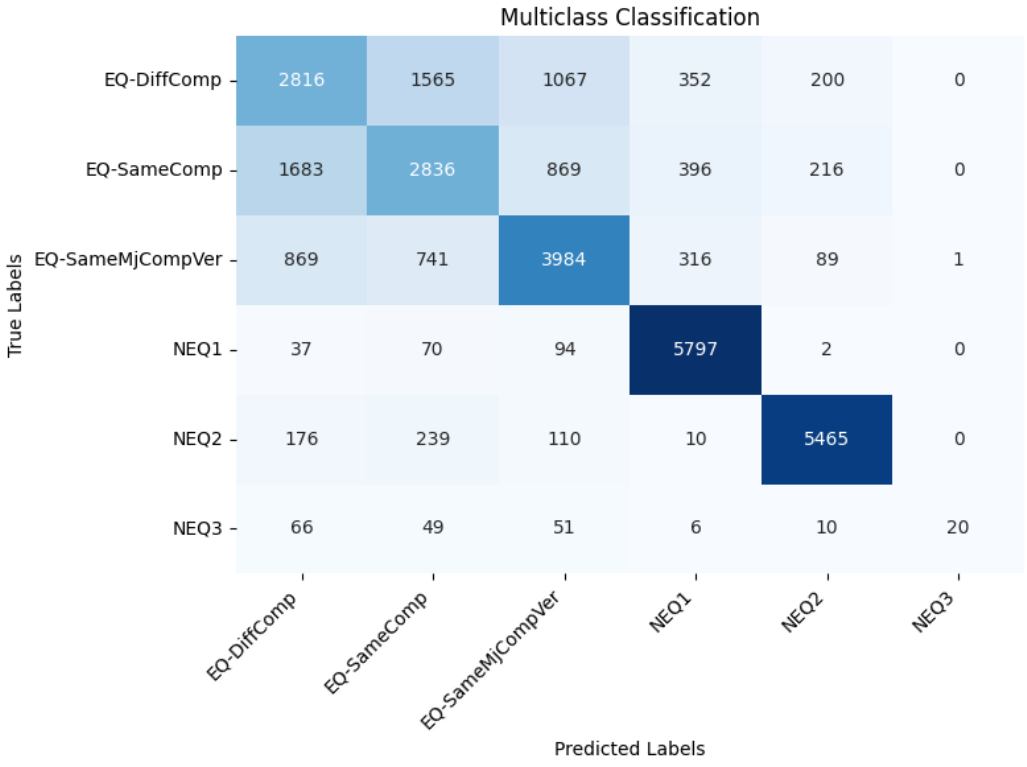

Figure 4.2: Confusion matrix for multiclass classification task: subsets of EQ and NEQ.

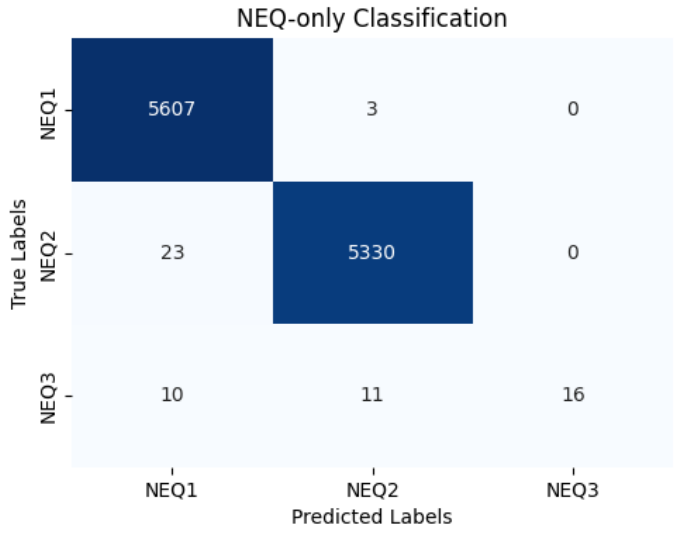

Figure 4.3: Further classification of pairs identified as NEQ.

Chapter 5

Discussion

This project applies machine learning techniques to the task of identifying binary equivalence. It is motivated by the issue of cybersecurity breaches arising from compromised build environments. While the results are generally positive, it is important to consider whether the solution is able to address binary equivalence from a cybersecurity perspective and what would be needed to progress towards a full solution.

5.1 Identifying Vulnerabilities

The classifier often failed to correctly identify pairs labelled NEQ3, which have behavioural differences due to vulnerability patches. We could see this because the recall scores for NEQ3 were low, and confusion matrices showed that it was often misclassified as some other label. However, the precision scores for NEQ3 were relatively high at 0.95 for the multiclass test and 0.94 for the NEQ subsets test. This means that when the classifier did identify a pair as NEQ3, it was usually correct. Given the small amount of data available for NEQ3, it is difficult to draw strong conclusions from this result. However, we can look to other evidence that machine learning models are able to identify code features relating to security vulnerabilities.

Claude 3 Sonnet is a multimodal LLM that can process both images and text and has been shown to be able to identify insecure code. In [31] the authors identified feature activations in Claude that related to certain topics. Topics ranged from specific (Amelia Earhart, the Golden Gate Bridge) to more abstract concepts including security vulnerabilities, bugs and backdoors. Claude was able to relate the concept of backdoors to images of covert recording devices and could be induced to write a program containing a backdoor by forcing the relevant feature activations well beyond their normal levels. We do not know whether the Salesforce CodeT5+ models have similar capabilities or whether any information regarding vulnerabilities was encoded in the embeddings. However, the capabilities demonstrated by Claude show that large language models can assist in identifying security vulnerabilities, so it is probable that they can be used to identify NEQ3 pairs.

5.2 Explainability

An explainable system is one where the reasoning behind a specific output can be identified. This property is highly valuable because it allows human beings to understand why and how a system makes decisions. As described previously, a major component of the verifiable builds approach to binary equivalence is to explain and document non-equivalences, a task which is carried out by humans. This is done to enable independent verification and

auditing [7]. Automated systems intended to supplement or provide an alternative to this process should also support verification and auditing. This could be achieved through high explainability.

A major obstacle for the use of large language models, including our use of embeddings, is their lack of transparency. LLMs are highly complex, making use of many-layered neural networks and being trained with millions or billions of parameters. This makes it difficult to trace or reconstruct their processes in a way that humans can understand; for example, to establish whether the output of an LLM is plagiarised. For comparison, consider the random forest that was used for classification. The individual decision trees that make up the random forest are highly explainable because they are essentially flow charts. An entire random forest is less explainable due to the high number of trees and their randomness, but the relative importance of input features can be calculated to provide some insight into the process. However, an LLM infers features from its training data rather than taking them as input the way a random forest does, so the same calculation cannot be applied. The explainability of LLMs is an area of active research: the previously mentioned paper on Claude is just one example [31, 32].

Our solution is currently not explainable, but developments in the explainability of large language models may resolve this going forward. Potential applications for the current solution could be to search large collections of binaries and flag any items of concern for further review. Automated software analysis techniques could be used to identify and prove the existence of non-equivalent behaviours. For example, an existing technique for automatically identifying malicious binaries is to run them in an isolated environment and observe their behaviour [33]. This could reveal the non-equivalent behaviours of flagged binaries. Another possibility is to use a test generation framework such as Randoop to automatically generate tests for flagged pairs of binaries [34]. If a test could be found to produce different results for each binary, it could serve as proof of non-equivalent behaviour. This would support verification and auditing without requiring the large language model to be explainable.

Chapter 6

Conclusion

This project tested the potential of machine learning techniques to identify binary equivalence in Java classes. To do this, we took the text representation of each Java class's bytecode and used a code understanding model to generate an embedding. A random forest classifier was used to classify the embeddings as EQ or NEQ.

The results showed that the Java classes could be identified as EQ or NEQ with an accuracy of 92%. Subsets of NEQ could be identified the majority of the time with good F_1 scores for the NEQ1 and NEQ2 groups. However, performance on the NEQ3 subset was relatively poor. This can be attributed to the low representation of NEQ3 data in the dataset. Aside from obtaining more data, it could potentially be addressed by adjusting the classifier parameters to penalise misclassification of NEQ3 items.

There are several steps that could be taken to improve the results of this research. Due to time and resource constraints, only a small amount of all the available BinEq data was used. It is likely that providing more training data to the classifier would improve its performance. The embedding generation process may benefit from use and fine-tuning of the tokeniser's 'stride' setting, as explained in section 3.4.1. There are many methods of classification that could be used as alternatives to the random forest algorithm and may produce better results. A more specialised code understanding model may exhibit better performance: while the CodeT5+ model produced good results, a model trained specifically on bytecode might perform even better. However, there are broader issues that would not be addressed by simply improving classification performance.

While the results show that the code understanding model can be used to identify binary equivalence, the outputs are not explainable at this time. Given the importance of transparency in current approaches to supply chain security, a lack of explainability limits the application of this solution. This would be helped by using an explainable code understanding model. There are ongoing efforts exploring this with other large language models such as Claude [31]. Alternatively, software analysis techniques could be developed to provide proof of non-equivalent behaviours without requiring explainability.

A different area for further research would be to find out whether the use of a code understanding model is as effective for other programming languages. This could be useful in domains where the Java programming language is not prominent, such as systems programming. There are languages that share the use of Java bytecode and the Java Virtual Machine, such as Kotlin and Groovy, where our approach could be expected to produce similar results. There are also languages that use the same general principle of running bytecode on a virtual machine, such as JavaScript and C#, and languages such as C++ and Rust that can be compiled to an intermediate representation using LLVM [35]. Future work could investigate whether code understanding models can identify binary equivalence for these languages. However, until there is a way to provide proof of non-equivalences, the

outcome of that research will also be limited by a lack of transparency.

The results from this project were not sufficient to show the reliable detection of compromised binaries, which were represented by the NEQ3 group in the dataset. It is likely that this can be done, given the ability of LLMs such as Claude 3 Sonnet to identify and create code vulnerabilities [31]. However, Claude's capabilities were shown in a limited way as part of a research effort into explainability, so we did not see how broad its understanding is. Further research could explore different classes of vulnerabilities to see which ones LLMs are able to detect.

In conclusion, this project demonstrates the potential of a large language model to identify binary equivalence in Java classes. However, further work must be done to enable explanation, documentation and verification of the output. With these elements in place, we will be able to improve the transparency and security of our software supply chains.

Bibliography

[1] K. Thomson, "Reflections on trusting trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, 1984.

[2] J. Martínez and J. M. Durán, "Software supply chain attacks, a threat to global cybersecurity: Solarwinds' case study," *International Journal of Safety and Security Engineering*, vol. 11, no. 5, pp. 537–545, 2021.

[3] "Assured oss." `https://developers.google.com/assured-os`. Accessed: 2024-04-22.

[4] "Maven central repository." `https://central.sonatype.com/`. Accessed 2024-10-13.

[5] A. Rybalov, "On the strongly generic undecidability of the halting problem," *Theoretical Computer Science*, vol. 377, no. 1, pp. 268–270, 2007.

[6] B. Godlin and O. Strichman, "Regression verification: proving the equivalence of similar programs," *Software Testing, Verification and Reliability*, vol. 23, no. 3, pp. 241–258, 2013.

[7] Y. Shi, M. Wen, F. R. Cogo, B. Chen, and Z. M. Jiang, "An experience report on producing verifiable builds for large-scale commercial systems," *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3361–3377, 2022.

[8] J. Dietrich, T. White, M. Abdollahpour, E. Wen, and B. Hassanshahi, "Bineq-a benchmark of compiled java programs to assess alternative builds," Sep 2024.

[9] J. Dietrich, T. White, M. M. Abdollahpour, E. Wen, and B. Hassanshahi, "Oracles for the equivalence of java bytecode," Dec 2023.

[10] "Software bill of materials (sbom)." `https://cisa.gov/sbom`. Accessed: 2024-04-22.

[11] U. S. Food and D. Administration, "Cybersecurity in medical devices: Quality system considerations and content of premarket submissions." `https://www.fda.gov/regulatory-information/search-fda-guidance-documents/cybersecurity-medical-devices-quality-system-considerations-and-content-premarket-submissions`, Sep 2023. Accessed 2024-09-28.

[12] "Supply-chain levels for software artifacts." `https://slsa.dev`. Accessed: 2024-04-22.

[13] "Reproducible builds." `https://reproduciblebuilds.org/`. Accessed: 2024-04-22.

[14] X. de Carné de Carnavalet and M. Mannan, "Challenges and implications of verifiable builds for security-critical open-source software," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, (New York, NY, USA), p. 16–25, Association for Computing Machinery, 2014.

[15] J. Xiong, Y. Shi, B. Chen, F. R. Cogo, and Z. M. J. Jiang, "Towards build verifiability for java-based systems," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ICSE '22, ACM, May 2022.

[16] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020.

[17] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. H. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint*, 2023.

[18] Z. Liu and S. Wang, "How far we have come: testing decompilation correctness of c decompilers," p. 475–487, 2020.

[19] Oracle, "javap." `https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javap.html`. Accessed 2024-05-30.

[20] M. M. Abdollahpour, "Semantic equivalence of java bytecodes," 2024.

[21] M. Ghasemisharif and J. Polakis, "Read between the lines: Detecting tracking javascript with bytecode classification," p. 3475–3489, 2023.

[22] T. Thongtan and T. Phienthrakul, "Sentiment classification using document embeddings trained with cosine similarity," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: Student Research Workshop* (F. Alva-Manchego, E. Choi, and D. Khashabi, eds.), (Florence, Italy), pp. 407–414, Association for Computational Linguistics, July 2019.

[23] Oracle, "Oracle java se support roadmap." `https://www.oracle.com/nz/java/technologies/java-se-support-roadmap.html`, Mar 2024. Accessed 2024-09-26.

[24] J. Coates and D. Bollegala, "Frustratingly easy meta-embedding – computing meta-embeddings by averaging source word embeddings," 2018.

[25] L. Breiman, "Random forests," *Machine Learning*, vol. 45, pp. 5–32, 10 2001.

[26] A. Parmar, R. Katariya, and V. Patel, "A review on random forest: An ensemble classifier," in *International Conference on Intelligent Data Communication Technologies and Internet of Things (ICICI) 2018* (J. Hemanth, X. Fernando, P. Lafata, and Z. Baig, eds.), (Cham), pp. 758–763, Springer International Publishing, 2019.

[27] scikit-learn developers, "Randomforestclassifier." `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`. Accessed 2024-09-26.

[28] N. Developers, "numpy.hstack." `https://numpy.org/doc/stable/reference/generated/numpy.hstack.html`. Accessed 2024-09-26.

[29] I. Muraina, "Ideal dataset splitting ratios in machine learning algorithms: general concerns for data scientists and data analysts," in *7th international Mardin Artuklu scientific research conference*, pp. 496–504, 2022.

[30] Y. Sasaki, "The truth of the f-measure," *Teach Tutor Mater*, 01 2007.

[31] A. Templeton, T. Conerly, J. Marcus, J. Lindsey, T. Bricken, B. Chen, A. Pearce, C. Citro, E. Ameisen, A. Jones, H. Cunningham, N. L. Turner, C. McDougall, M. MacDiarmid, C. D. Freeman, T. R. Sumers, E. Rees, J. Batson, A. Jermyn, S. Carter, C. Olah, and T. Henighan, "Scaling monosemanticity: Extracting interpretable features from claude 3 sonnet," *Transformer Circuits Thread*, 2024.

[32] H. Zhao, H. Chen, F. Yang, N. Liu, H. Deng, H. Cai, S. Wang, D. Yin, and M. Du, "Explainability for large language models: A survey," 2023.

[33] S. Jamalpur, Y. S. Navya, P. Raja, G. Tagore, and G. R. K. Rao, "Dynamic malware analysis using cuckoo sandbox," in *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*, pp. 1056–1060, 2018.

[34] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *OOPSLA 2007 Companion, Montreal, Canada*, ACM, Oct. 2007.

[35] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, (Palo Alto, California), Mar 2004.