



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Development of a data collection tool for the evaluation of a deductive verifier

Bachelor Thesis

Simon Hostettler

February 20, 2024

Advisors: Prof. Dr. Peter Müller, Dionysios Spiliopoulos
Department of Computer Science, ETH Zürich

Acknowledgements

Foremost, I express my gratitude to my supervisor Dionysios Spiliopoulos for his guidance and continued support throughout this thesis.

Furthermore, I would also like to thank my friends and family for their support during my bachelor's studies.

Abstract

Viper is a deductive program verification infrastructure and intermediate language. There are a variety of frontends using Viper to verify programs in different programming languages. Its verification performance can vary strongly depending on the programs it has to verify. The reduction of verification times is especially important for large projects.

In this thesis, we develop a data collection tool for the Viper infrastructure and frontends with the goal of building a dataset of user-written programs, their metadata, and verification benchmarks. This dataset will facilitate future profiling of the tool stack to identify performance bottlenecks. The tool consists of a database, a web server with a complementary frontend to access the data, and a processing algorithm that evaluates programs and filters submissions based on a program similarity algorithm we develop. To collect the data, we expand the frontends and verifiers of the tool stack with the capability to submit programs upon verification.

Contents

Contents	iii
1 Introduction	1
1.1 Outline	2
2 Background	3
2.1 Program Verification	3
2.2 Viper Tool Stack	4
2.2.1 Viper Intermediate Language	5
2.2.2 Silicon	8
2.2.3 Carbon	9
2.2.4 Frontends	9
2.3 Source Code Similarity Detection	10
3 Development	13
3.1 Database Design	13
3.1.1 Database Interaction in Scala	14
3.1.2 Schema Definition	14
3.2 Duplicate Data Reduction	17
3.2.1 Similarity Measurement	17
3.2.2 Fingerprinting Implementation	17
3.3 Features and Benchmarking	22
3.4 Submission Processing	23
3.4.1 Pipeline Overview	23
3.5 Query Frontend	26
3.6 Frontend Modifications	27
3.7 Extensions	28
3.7.1 Pattern Searching	28
3.7.2 Version Benchmarking	29

CONTENTS

4 Evaluation	31
4.1 Performance Measurements	31
4.1.1 Fingerprinting Performance	31
4.1.2 Processing Performance	33
4.1.3 Pattern Searching Performance	34
4.2 Similarity Detection Testing	35
5 Conclusion	39
5.1 Future Work	40
Bibliography	43

Chapter 1

Introduction

Program verification, the task of proving a program's correctness, has become increasingly important but also more challenging due to the growing complexity of software projects. Program verifiers, such as Viper[1], developed by the Programming Methodology group at ETH Zurich, are tools designed to automate large parts of the verification process. More precisely, Viper is a deductive verification infrastructure and intermediate language that has realized adoption in both academic research and practical applications globally. It is based on two backends, one using verification condition generation through an encoding to Boogie[2], and the other symbolic execution, both of which depend on an SMT solver. There are a variety of frontends that facilitate the compilation of annotated source code to Viper code. For large-scale projects like VerifiedSCION[3], reliable verification performance is essential. However due to the complexity and the various levels of abstractions of this tool stack, its performance can vary strongly, and finding bottlenecks can be challenging.

To get a proper understanding of these sources, a large dataset of programs to profile the various stages of the infrastructure would be helpful. This thesis addresses the creation of such a dataset through the development of a data collection tool for the Viper infrastructure. This tool can be used to collect and evaluate programs written by consenting users and is made up of two core parts.

The first is a backend made up of three parts. First, a database to store the programs, metadata and generated results. Second, a web server to query information about the stored programs and submit programs, and third a processing algorithm which verifies and benchmarks submitted programs. To reduce data duplication, we developed a similarity algorithm based on source code similarity to compare and filter submissions.

The second is a small tool implemented in the different frontends and veri-

fiers which enables users to submit their programs.

Additionally, we also developed a lightweight query frontend as a complement to the web server to facilitate future evaluation of our data. It contains methods to communicate with our web server and implementations of our common data types.

1.1 Outline

Chapter 2 gives an introduction to the theoretical concepts used during this thesis, by giving an overview of software verification, the Viper verification infrastructure and intermediate language and the concept of source code similarity detection.

Chapter 3 delves into the development of the collection tool and explains the choices made during this process, covering everything from the design of the database and the development of our code similarity algorithm to the modifications made to the frontends.

In chapter 4 we evaluate the performance of our tool to demonstrate its capability in handling real-life workloads. Additionally, we present the approach we used to define the behavior of the similarity algorithm.

In chapter 5 we conclude the thesis and provide insights into potential future applications.

Background

In this chapter, we will introduce theoretical and technical concepts used in this thesis. First, we will give an overview of software verification techniques, followed by an introduction to the Viper verification infrastructure. Lastly, we will explore current research into source code similarity detection.

2.1 Program Verification

Proving the correctness and security of programs has become more relevant than ever, with ever more complex software and large concurrent computing infrastructures. Program Verification is the subject of proving the correctness of programs given certain specifications. Generally, there are two types of verification - dynamic and static. Dynamic verification is performed by executing the software and trying to induce unwanted behavior, usually known as software testing. Static verification or analysis on the other hand tries to prove that a program has certain properties according to mathematical models. Analysis often results in either an under or over-approximation of programs that fulfill a property, i.e. either only no false positives or no false negatives can be guaranteed. There are multiple techniques commonly used in static analysis.

Model checking[4] is a technique involving exhaustively checking a mathematical model, either finite or infinite. Abstract interpretation is a common model checking technique, in which the state is approximated by ordered sets, and statements are used to transform the state with monotonic functions. Approximations are however often too imprecise or the state search too expensive.

Deductive Verification[5] is another commonly used technique. It consists of the generation of proof obligations which imply the correctness of a program in the given model and a search for proof of these obligations. Deduc-

tive verifiers employ a variety of proof search strategies and often depend on either interactive theorem provers or automatic solvers like satisfiability modulo theories (SMT) solvers. The Viper infrastructure is based on deductive verification.

SMT solvers[6] are tools designed to solve satisfiability modulo theories problem, determining whether a formula in first-order logic is satisfiable over some given theories such as arithmetic, arrays and bit vectors. As a generalization of the boolean satisfiability problem, this task is also NP-complete. The approach many solvers take is therefore to prune the search space as efficiently as possible by employing various heuristics and learning techniques. As such these procedures are rather variable in their efficiency of finding a solution. Users often need to make informed decisions about the approach based on the specific problems they are attempting to solve.

2.2 Viper Tool Stack

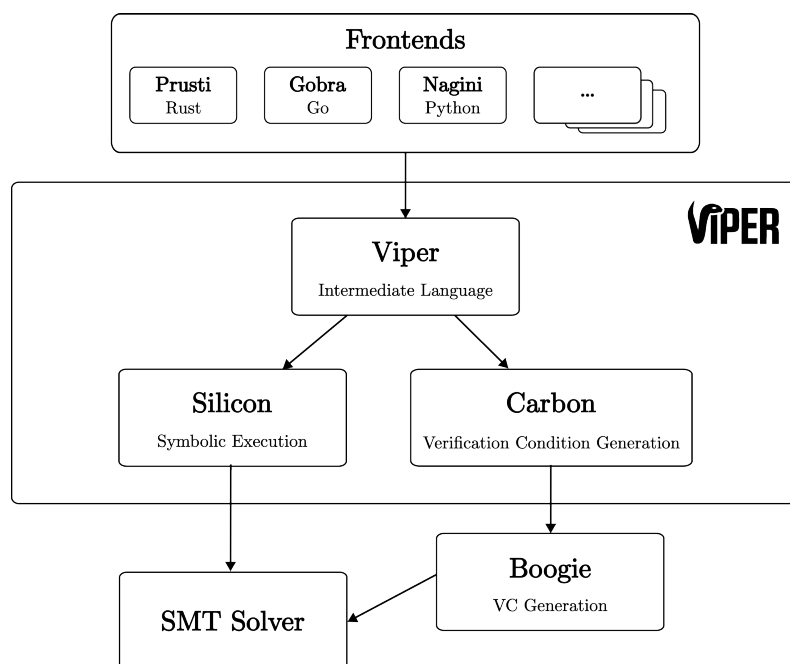


Figure 2.1: Viper tool stack, logo sourced from the official Viper homepage[7]

Viper[1] is a verification infrastructure natively supporting permission logics like implicit dynamic frames. This can be used to reason about heap-manipulating programs and thread interactions. It is intended for the development of program verifiers and prototyping of verification techniques. Viper is also the name of the provided intermediate language, which can

be used by the frontends. Verification is achieved through two backends: Silicon[8], based on symbolic execution, and Carbon[9], based on verification condition generation. Both backends ultimately depend on Z3[6], an SMT solver developed by Microsoft Research.

2.2.1 Viper Intermediate Language

The Viper language is sequential, object-based and imperative[10]. Even though originally developed as an intermediate language, it is also useful for manual encoding and verification of verification problems, due to its human-readability and high-level features.

Viper natively supports simple types such as Integers, Booleans and References, but also more complex data types such as Sequences, Sets and Maps, and the usual unary and binary operators for these types. It also allows for the definition of custom types.

Permission logic upon which Viper operates is based on implicit dynamic frames, which is inspired by separation logic[11]. Separation logic itself is an extension of Hoare logic and facilitates reasoning about the manipulation of pointer data structures, concurrency, and ownership. Separation logic operates on states consisting of stores and heaps. It can however not reason about heap-dependent expressions in assertions, a problem which implicit dynamic frames solve[12]. Compliance with these specifications can be reduced to first-order logic.

To reason about permission logic Viper uses the concept of permissions. Permissions control access to the program heap, which simplifies framing, i.e. proving that a heap modification does not violate an assertion or arguing about concurrent heap accesses. They are defined by access predicates like $\text{acc}(x.f)$, which gives exclusive read and write access of the field f of the object x to the current method. These permissions can also be fractional and defined by a variable, i.e. one can hold a part of a permission p . For $0 < p < 1$ this gives the holder read permissions only, for $p = 1$ read and write permissions and for $p = 0$ no permissions. Permissions can be held by method or loop executions. Accessing a heap location requires holding the permission for this location. Permissions can be transferred between executions using pre- and postconditions. Acquiring a permission is also called *inhaling* and releasing *exhaling*.

A Viper program is generally made up of six top-level declaration types[10], which we will shortly explain here. Listing 1 shows the basic syntax for these declarations.

Fields represent a heap location and are global, each object in Viper has the same fields.

Methods are an abstraction over a sequence of statements. The method body is opaque to callers, to the outside methods are exclusively defined by their signature, preconditions and postconditions. Methods can have side effects, i.e. modify the program state. To call a method, its precondition has to be fulfilled and results in the caller obtaining the assertions of the postconditions.

Functions are an abstraction over a sequence of expressions. Function calls result in a check of the precondition, then the result value is set to the expression in the function body, then its post-conditions are assumed. Functions cannot have any side effects.

Predicates are an abstraction over assertions. They are often used to specify recursive data structures. Predicate bodies are assertions, which are however not automatically inlined. Calls to `unfold` and `fold` have to be made to replace the predicate with its assertions and vice versa.

Domains are used to define new types and mathematical systems. They contain abstract function and axiom definitions, which can be used to derive assertions.

Macros are used to reduce code duplication. Before verification, they are syntactically inlined into the program similar to C-style macros. A key difference however is that Viper macros must have a well-typed body.

Higher-level Features

In addition to these basic features, Viper also supports more advanced high-level language features to increase its usefulness as an intermediate language. This functionality can however also lead to performance degradation, which is not easy to pinpoint, as the backends can decide to use more costly verification techniques to support these features. Here we introduce some of these features, which we took a closer look at in our development.

Recursive predicates are a common way in separation logic to define recursive data structures such as lists and trees. Like permissions, predicates can be held by method and loop executions and transferred between them. The ability to define recursive predicates, i.e. using the predicate in its own body leads to the `unfold` and `fold` statements, which have to be called manually. Otherwise, the prover might decide to indefinitely unfold a recursive predicate.

Magic Wands are a binary operator in separation logic. $A \text{ --* } B$ intuitively means that if the current heap is extended with a disjoint heap defined by the assertion A , the resulting heap satisfies B . When reasoning about recursive data structures, an assertion describing the already visited part of the structure is often needed to get back an assertion about the entire structure

```
1  field val: Int
2
3  method m(x: Int) returns (y: Int)
4      requires ... // precondition
5      ensures ... // postcondition
6  {
7      // body (optional)
8  }
9
10 function f(x: Ref, a: Int): Int
11     requires ... // precondition
12     ensures ... // postcondition
13 {
14     ... // body (optional)
15 }
16
17 predicate p(this: Ref) {
18     ... // body (optional)
19 }
20
21 domain D[A, B] {
22     function getFirst(d: D[A, B]): A
23     // other functions
24
25     axiom ax_1 {
26         ... // axiom body
27     }
28     // other axioms
29 }
30
31 define plus(a, b) (a+b)
```

Listing 1: Syntax of the top-level declarations in Viper, shortened version of examples from the Viper tutorial[10]

in the future. This leads to the definition of auxiliary predicates, which in turn require additional methods to permute between these different representations. Magic wands simplify this reasoning. B could describe an assertion defining the entire data structure and A the permissions to the left-over part that we are currently traversing. By applying the magic wand, A and the wand itself are given up to obtain B. Viper includes many heuristics to verify assertions using magic wands.

Quantified Permissions represent another way to specify unbounded heap structures. They use point-wise specifications, especially useful for data structures that aren't limited to being traversed in a single way such as generic graphs or arrays. Quantified permissions consist of a quantified variable and a resource assertion, i.e. $\text{forall } x: \text{Ref} :: x \text{ in } S \implies \text{acc}(x.f)$. Additionally, Viper allows users to provide trigger expressions for quantifiers to aid the verifiers in finding quick solutions by restricting the instantiation of a quantifier to the provided expressions. The trigger $\text{forall } i: \text{Int} :: f(i)$ implies that the quantifier will only be instantiated for expressions matching $f(i)$, for some $i: \text{Int}$.

2.2.2 Silicon

Silicon[8], developed by Malte Schwerhoff, is one of the two verifiers used by the Viper Infrastructure. Taking Viper programs as an input, it tries to prove given assertions using symbolic execution.

Symbolic execution is a form of static analysis, i.e. reasoning about program behavior without executing the program. Instead of assuming concrete values, symbolic execution keeps a symbolic state which describes possible values and path constraints, which are conditions necessary to reach a certain point in a program execution.

Silicon's symbolic state is made up of a symbolic store and heap. The store maps local variables to their symbolic values, whereas the heap keeps track of the currently accessible locations in the program and their labels. During execution, Silicon passes through the program and after each statement updates its state according to its execution semantics. If a branch is encountered in the program, such as an if-else statement, all feasible paths are executed and the path conditions are updated with the assumptions needed to take this specific path.

Once an assertion is reached, Silicon queries Z3, an SMT solver, to check whether the assertion holds given the current state and collected conditions.

If all possible paths have been executed and all assertions were confirmed to hold, Silicon can assume program correctness.

2.2.3 Carbon

The second verifier used by the Viper infrastructure is Carbon[9]. Carbon reasons about program correctness using verification condition generation.

Verification condition generation is another form of static analysis, utilizing formal systems such as Hoare logic to reason about program correctness. The goal is to compute a formula that if valid implies the correctness of the entire program.

Carbon achieves this by encoding the permission-based logic into Boogie[2], a verification language and tool for first-order logic verification. Boogie then generates verification conditions and checks their satisfiability through Z3. In general, Boogie calls Z3 far less often than Silicon, which has to check all assertions in a program separately. Since first-order logic is undecidable with the use of quantifiers and the generally more complex formulas produced by Carbon than Silicon, some programs that are verifiable through Silicon are not verifiable by Carbon, at least in a reasonable timeframe. This is also the case the other way around, as Carbon is for example better at verifying programs using quantified permissions.

2.2.4 Frontends

Viper allows verification of programs through manual encoding in its intermediate language due to its readability and high-level features. Users can then verify the Viper program through Silicon or Carbon directly. However, for end-users, it is often simpler and less arduous to simply annotate assertions and conditions in the original program. This also avoids human errors in translation between languages or missing language-specific quirks. To solve this, the Programming Methodology group developed several frontends for different programming languages, which compile programs and their annotations down to Viper code. Here, we shortly introduce the most relevant frontends, which we will also collect programs from.

Gobra

Gobra[13] is a frontend for verifying Go programs. It was developed for and is mostly used to verify the correctness and security of the VerifiedSCION[3] project. It is therefore often used to generate very large Viper programs, which can strongly vary in performance, which is why the analysis of these programs is very important.

Prusti

Prusti[14] is a frontend for verifying Rust programs. It is special in that it uses the strong guarantees given by the Rust type and ownership system to

simplify reasoning about the permission logic used by Viper.

Nagini

Nagini[15] is a frontend for the verification of Python programs. It supports the verification of a large subset of statically typed Python. Nagini was also originally developed for the VerifiedSCION project and was the basis for 2vyper, a verifier for Ethereum smart contracts.

ViperServer

ViperServer[16] is not a frontend in itself, it is however a server that multiplexes verification requests of tools to the Viper backends. It is foremost used for Viper IDE[17], a Visual Studio Code extension that facilitates automatic verification and caching of Viper programs written in VS Code. Most end-user code is verified through Viper IDE instead of manual calls to Silicon or Carbon.

2.3 Source Code Similarity Detection

The measurement of source code similarity is a well-researched topic with many applications such as plagiarism detection, malware analysis and code duplication detection. A broad overview of the topic is given by the literature review paper by Morteza et. al[18]. There is no general definition for similarity, however, it is usually associated with a similarity score s over two programs $c1$ and $c2$, where the higher s is, the more similar $c1$ and $c2$ are. For a given measurement, a similarity scope and threshold usually have to be defined. The scope describes the granularity of elements that are compared and the threshold describes a similarity point at which two programs are near enough to count as clones. Similarity can be viewed from two different angles, as in the similarity between the source code syntax and its behavior. From this, there are four different types of widely accepted definitions of code clones.

- Type I: The two program texts differ only in whitespace and comments.
- Type II: The program texts may additionally differ in identifier names and types.
- Type III: Part of the program texts may additionally be removed, added, or updated.
- Type IV: The program texts are entirely different but have the same functionality.

There is a multitude of different similarity measurement techniques in use today.

Text-based Techniques

The simplest form of similarity detection is treating the source code as simple text and using string comparison to find overlapping parts. An advantage of this is the relative efficiency of string comparison and the fact that it is language agnostic, thus simple to implement. There exist various implementations of this technique, some even implementing advanced features such as lexical and AST analysis, however, they are mostly able to detect type I and II clones, sometimes III, but not IV.

Token-based Techniques

A bit more advanced form of similarity detection is token-based. Here programs are converted to streams of tokens which are then compared to find common subsequences. This gives an advantage in being more robust against small changes, such as in identifiers, and is relatively efficient. Token replacement analysis however increases the processing time significantly. Token-based techniques are usually also good at identifying type I and II clones, but type III only for larger edit distances.

Tree-based Techniques

In tree-based techniques, the source code is converted to an abstract syntax tree using a language parser. The similarity problem is then reduced to the problem of finding common subtrees. This can be computationally expensive and a different parser has to be used for every programming language. These techniques however are very good at detecting type I, II and III clones accurately. They also allow for the abstraction over node types or groups of nodes to reduce the granularity of the similarity comparison.

Graph-based Techniques

In graph-based techniques, the programs are usually converted to data and control-flow dependency graphs. These techniques have the potential to detect all four types of clones accurately, however, comparison boils down to finding graph isomorphisms, which is an NP-complete problem and therefore difficult to implement efficiently.

Development

The data collection tool developed during this thesis is made up of three core parts. First, a database instance running in a Docker container, to simplify its setup and potential restoration, to store our collected data. Second, a web server which is used to submit programs or query information about the dataset. Third, a processing script that converts submissions into entries for the database. Figure 3.1 gives an overview of the interactions between these parts. The tool is intended to run indefinitely on a dedicated server and once correctly set up, will require no further user interaction. In this chapter, we will explain the process of developing this tool and justify design choices made along the way.

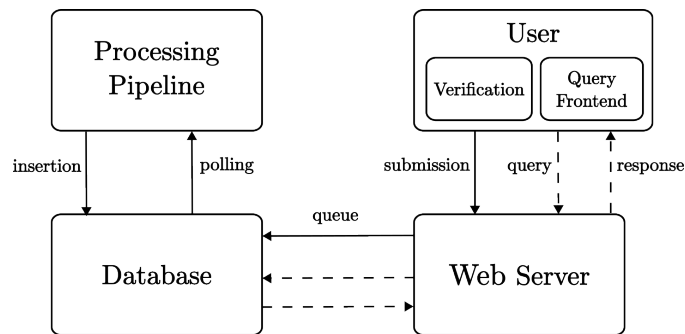


Figure 3.1: Overview of the data collection tool

3.1 Database Design

To construct our dataset, an efficient method for storing and retrieving programs, their associated metadata and benchmarking results is important. The common approach to storing large amounts of data like this is to use either a relational or non-relational database. We opted for a relational ap-

proach due to its advantages for our use case. In a relational model, all our data could be organized into predefined schemas with clear relationships between the tables. Since our data representation is unlikely to vary much, this gives us a consistent view of our data and greatly simplifies the Object-Relational mapping (ORM) between the stored data and its representation in our code heap. Schemas also facilitate more complex queries, enabling easy joining and aggregation between entries. For the specific database management system (DBMS), we selected PostgreSQL for several reasons. Firstly, being open-source, it incurs no licensing costs. PostgreSQL also supports advanced data types such as JSON and arrays, offering versatility compared to what is commonly available in other larger DBMSs like MySQL. The large and active community surrounding PostgreSQL was also a compelling reason for its choice, making it easier to find solutions to common problems during development

3.1.1 Database Interaction in Scala

To facilitate data storage and retrieval within our codebase, we chose *Slick*[19], a database access library. Being built on top of the Java Database Connectivity (JDBC) library, *Slick* abstracts away many of its low-level intricacies. It provides a type-safe and composable Domain Specific Language (DSL) in Scala for writing queries. *Slick* automatically manages ORM, compiling queries into SQL statements at compile-time. This design enables treating the database like any other collection type in Scala. Due to its DBMS implementation agnosticism, *Slick* also allows changing out the underlying DBMS in the future, without having to modify any significant parts of our codebase. The combination of these features makes *Slick* an attractive solution for handling database operations in our codebase.

3.1.2 Schema Definition

Each database table requires a corresponding case class in Scala, allowing us to easily store and retrieve instances of these classes. In the following subsections, we show the layout and implementation of our tables and their uses. We will explain some of the fields, however, most should be self-explanatory. Figure 3.2 shows an overview of our schema.

UserSubmissions

This table contains submissions sent in by users from either frontends or one of the verifiers (see section 3.6). This table is used as an input queue where we store submissions until they can be processed to be stored in the dataset. In Scala, these rows are represented by the case class `UserSubmission`.

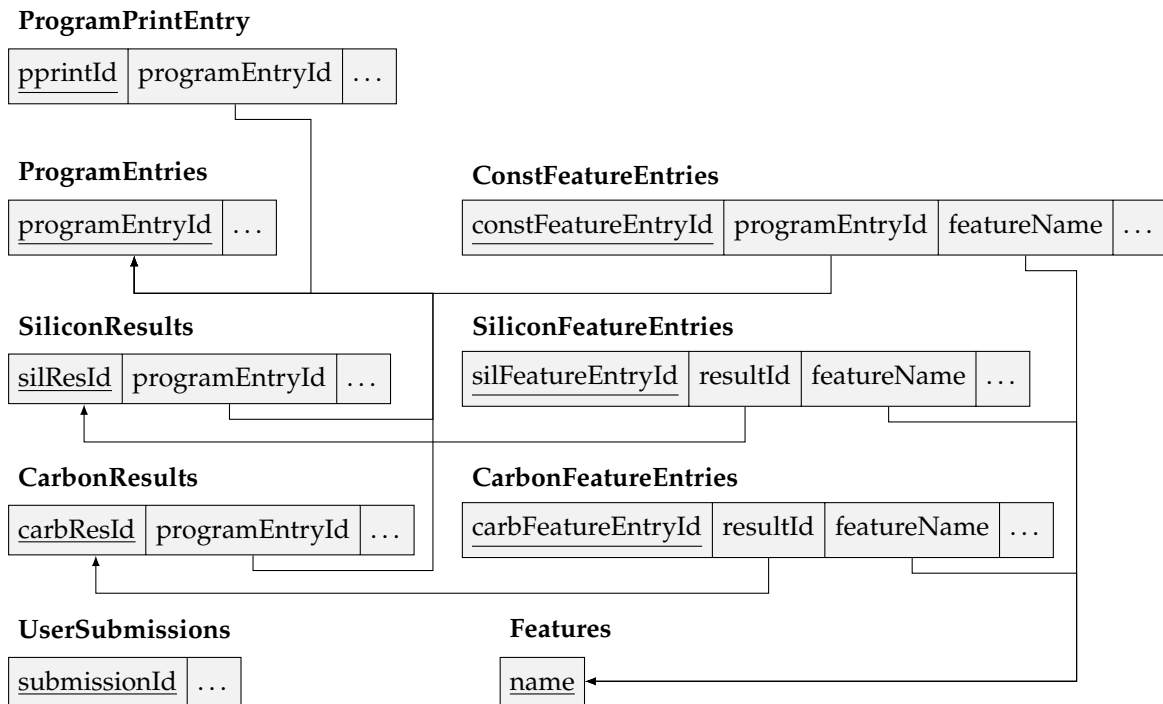


Figure 3.2: Relational diagram of our database schema. Arrows represent a foreign key relationship and underlined fields primary keys.

ProgramEntries

This table contains submitted programs and some of the original metadata gathered during the submission. These entries are created after processing the submission.

UserSubmissions	ProgramEntries
<u>submissionID</u>	<u>programEntryId</u>
submissionDate	submissionDate
program	program
loc	loc
frontend	frontend
args	originalVerifier
originalVerifier	args
success	originalRuntime
runtime	parseSuccess

Figure 3.3: UserSubmissions and ProgramEntries table layouts.

SiliconResults and CarbonResults

The SiliconResults and CarbonResults tables are both defined by the same layout. They contain the results of the local verification and benchmarking of the submitted program. The field `phaseRuntimes` contains more detailed runtimes of the different phases of the verifier, which are currently Parsing, Semantic Analysis, Translation, Consistency Checking and Verification. In Scala, these rows are represented by the case class `VerResult`.

ProgramPrintEntries

This table contains the binary serialized program fingerprint described in subsection 3.2.2. This allows us to efficiently load program prints back into memory to compare against a new submission, without having to recompute the fingerprints for all programs. In Scala, these rows are represented by the case class `ProgramPrintEntry`.

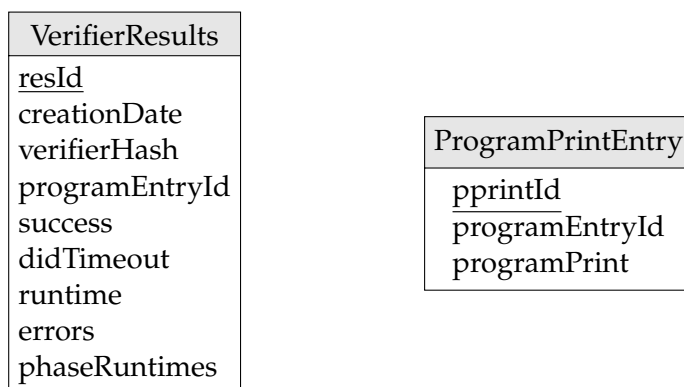


Figure 3.4: SiliconResults, CarbonResults and ProgramPrintEntry table layouts.

Features

We want to be able to store various features obtained during the verification of our programs. The Features table contains the names of all measured features. The `ConstFeatureEntries`, `SilFeatureEntries` and `CarbFeatureEntries` tables are all based on a `FeatureEntry`, where an entry is associated with either a verification run of a program where this feature was measured or a row in `ProgramEntries` if the feature is not verification-dependent. Feature values are stored as strings since we cannot predict what type a feature would have.

FeatureEntries
featureEntryId
featureName
referenceId
value

Figure 3.5: FeatureEntries table layout

3.2 Duplicate Data Reduction

One of the core goals of this thesis is to reduce duplicate data in our dataset. To solve this problem, we developed a way to measure similarity between programs to avoid storing nearly identical ones. Furthermore, we took into account some results of the evaluation to keep similar programs with very different behavior. In subsection 2.3 we introduced contemporary research of source code similarity detection, which we will use in this section to explain the development of our similarity algorithm.

3.2.1 Similarity Measurement

For our use case, we wanted a measurement technique that would perform efficiently, since every new program needs to be compared to all existing ones in the database, that can detect code clones up to type III reliably and that can store an intermediate representation to reduce processing time in the future. We therefore decided to go with a tree-based approach. This also gave us the freedom to abstract over node types.

In the paper *"Syntax tree fingerprinting for source code similarity detection"* [20], Chilowicz et. al. propose an algorithm that fulfills our prerequisites and was used as the base for our fingerprinting algorithm. Although not explicitly mentioned, their core data structure seems to be based on a Merkle tree. The basic idea of the algorithm is to parse a program to obtain its abstract syntax tree and convert this to a tree of fingerprint nodes. A fingerprint node is made up of its subtree weight and a hash value, indicating its structural properties. This allows finding exact subtree matches rather efficiently. The authors additionally store a pointer to the relevant AST node to cross-reference the two trees, however, for our purposes, this was not needed.

3.2.2 Fingerprinting Implementation

Data Structure

To store information about the structural properties of a program, we use fingerprint nodes, called `FPNode` which store a fingerprint `Fingerprint` and a list of the node's children. A fingerprint is made up of the weight of the

```
1  case class Fingerprint(weight: Int, hashVal: String)
2
3  case class FPNode(
4      fp: Fingerprint,
5      children: Seq[FPNode]
6  )
7
8  case class ProgramPrint(
9      domainTree: FPNode,
10     fieldTree: FPNode,
11     functionTree: FPNode,
12     predicateTree: FPNode,
13     methodTree: FPNode,
14     extensionTree: FPNode,
15     numMethods: Int,
16     numFunctions: Int
17 )
```

Listing 2: Case class signatures of the fingerprinting datastructure

node’s subtree and its structural hash value. A fingerprint of a program, represented by the case class `ProgramPrint`, is split into multiple trees representing the top-level declarations of a Viper program, such as custom domains, fields, functions, methods, predicates and extensions. This allows us to only compare relevant parts of a program to each other, speeding up the comparison.

Building the Fingerprint Tree

To build the program fingerprint the algorithm proceeds as follows: As an input, it receives the program’s AST. The AST is then split into the above-mentioned trees, each of which is fingerprinted individually. First, the fingerprinting method is applied to the node’s sub-nodes recursively. The fingerprint hash value of the current node is then obtained by hashing the node’s type-hash, taken from a lookup table, concatenated with the children’s hash values. Lastly, the children’s weights are summed up and increased by 1 to obtain the weight of the current subtree. Figure 3.6 shows a simplified version of this process.

Structural Hash Function

The hash function used for this algorithm should represent the tree’s structural properties and to guarantee scalability should be calculated incremen-

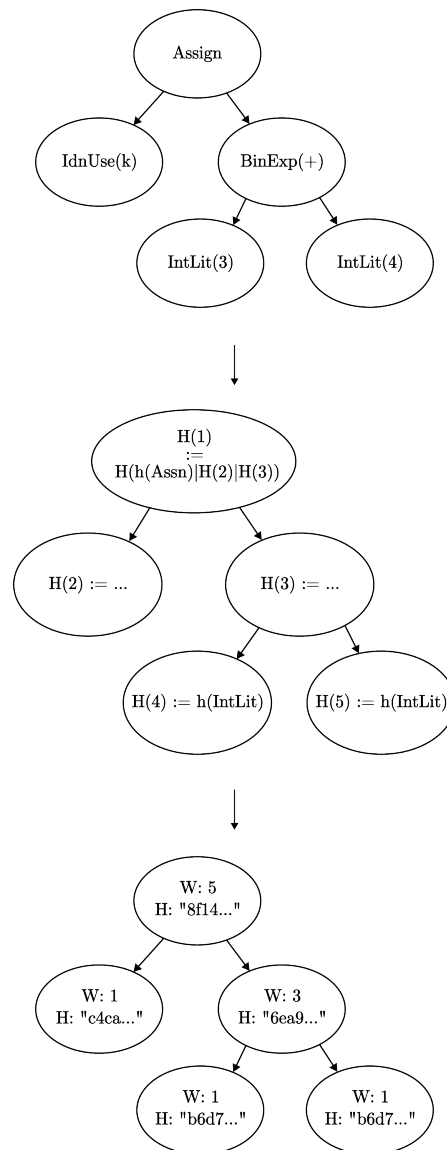


Figure 3.6: Simplified process of the fingerprinting algorithm applied on the statement $k := 3 + 4$. The first tree is the AST produced by the Viper parser. In the last tree, W denotes the weight of the subtree and H the fingerprint node's hash value.

tally, to avoid having to go through each node's entire subtree every time the hash is calculated. Chilowicz et. al. propose multiple different ways such a hash function could be built.

- **Sum hashing:** This method consists of counting the number of node types in a given subtree, resulting in a vector where the i -th entry is the count of nodes of type i . This allows comparison by the Euclidean distance of two vectors, however, ignores a lot of structural properties

of a tree, resulting in irrelevant matches.

- **Dyck word hashing:** For this method, a Dyck word is built for each node, a serialized form of the subtree obtained from depth traversals. This word is then processed using a Karp-Rabin hash function, which can be incrementally computed.
- **Cryptographic hashing:** For this method, nodes are indexed bottom-up. The hash value of a node is the hash of the concatenation of the node type hash with its children's hash values. This process is linear in the number of nodes in the tree and is completely incremental.

For our implementation, we went with cryptographic hashing, due to its strong performance and simplicity. To get the hash value for a node type, we created a lookup table with a random string for each Viper node type. For some node types, such as identifiers or binary operations, the type identifier or the binary operator is added to the hash such that nodes with the same type but different behavior can be differentiated. To hash a node, the node's type-hash is concatenated with the children's hash values. An MD5-hash of this string is then created and the resulting value is shrunken by selecting the first 16 hexadecimal digits. According to the author's experiments, this should be more than enough information to make the chances of accidental collisions infinitesimal. We end up with the following hash function:

$$H(n) = MD5(h(n) \parallel H(c_0) \parallel \dots \parallel H(c_n))_{[0:15]}$$

where c_i is the i -th child of the node n and h is the lookup table for the random string for n 's node type. Accidental collisions should not pose a large problem using this function. After the trimming described in 3.2.2 even larger programs in the order of 10^5 lines of code have a fingerprint tree containing less than 10^6 nodes. Since the hash values consist of 16 hexadecimal digits, each value contains 64 bits of information. We can approximate the chance $p(n)$ of at least one collision between any two nodes out of n nodes for a hash function with an output space of size H with

$$p(n) \approx \frac{n^2}{2H} \implies p(10^6) \approx \frac{10^{12}}{2^{65}} \approx 2.71 * 10^{-8}$$

In the unlikely event that two structurally different nodes accidentally have the same hash value that would result in a vastly different matching score, we still account for differing metadata and features of the programs, as described in 3.4.1.

Optimizations

This basic implementation is decently accurate, however, in some outlier cases, the matching results were unexpected or rather slow. These are some of the optimizations we added to the fingerprinting algorithm.

- **Trimming:** Leaf nodes make up a significant portion of the tree, increasing the storage size and all have their base hash value from the lookup-table. These lead to a lot of accidental matches due to programs containing the same nodes, but not in any structurally similar way. Therefore, all nodes in trees with a weight smaller than 4 are dropped, drastically increasing the accuracy of our matches.
- **Exploiting commutativity:** Some node types are semantically equivalent when their sub-nodes are in a different order, such as binary operations with commutative operators. A node's sub-nodes are therefore split into commutative and non-commutative parts, the former of which are sorted before concatenation to receive the same hash no matter the order of the sub-nodes.
- **Condition flattening:** The `&&` operator in Viper corresponds to the `*`-operator in separation logic, not purely a logical *and* and is as such not commutative. However for our purposes, pre- and postconditions that contain several *anded* clauses are almost semantically equivalent to conditions that list these clauses as multiple assertions. We therefore flattened any conditions with *anded* clauses into multiple separate clauses.

Matching

To determine an overall match percentage given two program fingerprints, we need to find exact matches across subtrees. Due to the fingerprint node's immutability, a new fingerprint node implementation was needed, a comparable fingerprint node. This additionally stores a boolean indicating whether it was already matched to a node in the other tree. To match two programs, a matching method is applied to all the program trees separately. This method goes through the tree in a depth-first-search order. For every node, the other programs' respective tree is searched, also in a depth-first search order. To speed up matching, the search of a given subtree is aborted if its weight is lower than the nodes, or if that subtree was already matched. If a node with the same fingerprint is found, that node is marked as matched and the weight of the current node is returned. At the end, the weights of all matched nodes are summed up. The return value is a tuple of the amount of matched nodes and the total nodes in the tree. This operation is therefore not commutative between two trees, as one program could be entirely contained in another, but not the other way around. The match percentage we use from here on is the ratio between the matched and total nodes.

3.3 Features and Benchmarking

Certain Viper features can cause drastic changes in verifier behavior and performance by changing solver approaches. We deemed it important to store which of these are used by a program. For this, we added the `FeatureEntry` tables, one each for Silicon, Carbon and constant features. Silicon and Carbon features are unique to the verifiers and can change across verification runs, constant features however should never change and are therefore only stored once. These tables store the name of a feature and any value associated with it as a string. Many of the chosen features are binary, either a program uses them or not. For those, the strings "true" and "false" are used as placeholders for booleans. To generate these feature entries, we implemented a `FeatureGenerator`, which is invoked during each verifier run and returns a list of tuples with the name and value for each feature. Splitting the table into the three categories and storing values as strings was decided on for flexibility and future-proofing. If any additional metadata or features of programs should be tracked in the future, the `FeatureGenerator` can simply be extended with this behavior. Here we briefly explain some of the features we are currently tracking.

- Whether the program uses any collection types. That is sets, sequences, maps, or multisets.
- Whether the program uses magic wands. These can be very costly if the verifier's heuristics fail.
- Whether the program uses any permissions. There are multiple keywords indicating permissions, two of which are added as separate features due to their potential performance impact. The first one are wildcards, which specify any positive permission amount. Wildcard amounts are randomly chosen each time the keyword is encountered. Successive exhaling and inhaling of wildcard permissions might therefore not restore the original permissions and behave in a non-deterministic way. The second are for-permissions. These indicate a quantifier over all objects of which the current method has access to the specified field.
- Whether the program uses recursive predicates or functions.
- Whether there are quantifiers with missing triggers in the program. Not restricting possible quantifiable expressions can result in significantly more work for the verifier.
- Whether the program might use quantified permissions. Since this is up to the verifier, there is no simple and guaranteed way to determine this, therefore this feature is an overestimation. It simply expresses

whether the program contains a quantifier with an access predicate in its expression.

- Whether the program type-checks.
- More detailed information about the time spent at each stage in Silicon compared to the simple measurements stored in a `SiliconResult`. Arqunt Linard developed a tool `SymbExLogger` which can be used to output information about a Silicon verification run upon the visit of each member of a program. Members can be any part of a program, such as methods, predicates, or functions. This logger is the basis of a benchmarker which outputs this more detailed runtime breakdown. Once it is officially merged into Silicon, we have the capability of storing this output as a feature.

Most of these features are of a syntactic nature and constant, which allows us to use the RegEx matcher we developed in subsection 3.7.1 to quickly determine their presence. Others can be found by a simple analysis of the program's AST. Any additional features that should prove useful in the future can be tracked very easily due to the way we designed the tables and `FeatureGenerator`.

3.4 Submission Processing

When a user submits a program to our backend, several steps have to be taken to prepare the data to be entered into our dataset. In this section, we will explain the pipeline submissions go through until they are stored step by step.

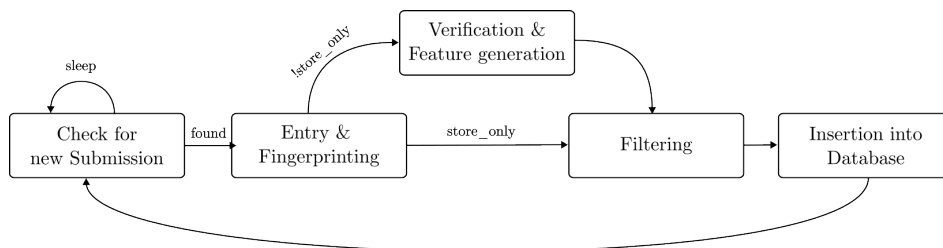


Figure 3.7: Submission processing overview

3.4.1 Pipeline Overview

When a submission is sent, it is automatically inserted into the `UserSubmissions` table of our database. A script is invoked on a regular interval to check for new submissions and to process them for insertion into the database. To avoid taking up resources and spin-locking when no submissions are available, the script goes to sleep for a few seconds in case there is nothing to

do. If a new submission is found, it goes through several stages, which will be explained in the following subsections. To guarantee consistency in measurements, a global lock is acquired at the beginning of the processing to avoid sharing resources with multiple instances of the pipeline. If any stage fails, local temporary data is removed and the submission is deleted, to avoid getting stuck on an invalid submission.

Entry creation and Fingerprinting

The submission is first removed from the `UserSubmissions` table. It is then converted into a `ProgramEntry`, which is stored locally for the moment. The program is then fingerprinted using our fingerprinting method and the resulting `ProgramPrint` stored in a `ProgramPrintEntry`.

Verification and Benchmarking

Verifier performance is a key aspect for any future evaluation of our programs. To double-check the submission results and measure performance reliably, any submission is therefore verified locally. This gives a relative consistency in our measurements, since local resources are relatively stable compared to the vastly different hardware users verify the programs with. Programs are verified through both Silicon and Carbon since we are interested in the performance and capabilities of both backends. This is done through two custom frontends, which measure runtime for each phase of the verifier and implement a `FeatureGenerator` which measures and generates the feature values described in section 3.3. The verifiers use caching for program parts they have already seen, which is rather difficult to completely disable. For consistency, the verifiers are therefore run through new JVM instances to guarantee an identical initial state. This increases the verification time by a few seconds generally but is needed. To ensure identical verification behavior, the arguments passed to the original verifier by the user are also passed to the corresponding local verifier, excluding a few flags referencing local files. The other verifier is executed without any additional arguments, as there is no direct correspondence between most of Silicon's and Carbon's flags. Since it is not possible to know whether the verification of a program will terminate, as the submission might have time-outed or been manually terminated, all programs are run with a timeout estimated as a multiple of the original runtime. Upon completion of the verifications, the measured results and features are stored locally in a `SiliconResult`, `CarbonResult` and multiple `FeatureEntry` instances.

Filtering and Insertion

In this stage, programs are checked to determine whether they are worth including in the dataset. For this, the previously generated case classes are

compared against any possible matches in the entire dataset. If a program is determined to be too similar to any existing entry it will be discarded. To be defined as *too similar*, we defined multiple criteria which all have to be fulfilled. Due to the amount of criteria, it is relatively easy for any submission to pass these checks, as they only have to differ in one aspect to all other programs. This was decided on since we predict that programs that are similar in most aspects yet have a few glaring differences, such as with respect to performance, would be valuable to keep for any future evaluation. Here we give a short explanation of these criteria.

- Two `ProgramEntry` instances are similar *iff*
 - The length in lines of the programs are within 20% of each other.
 - They were verified using the same verifier.
 - They were generated by the same frontend or both handwritten.
 - They both succeed or fail to parse.
- Two `SiliconResult` or `CarbonResult` instances are similar *iff*
 - Verification succeeded for both or both generated the same error types.
 - Their runtimes are within 50% or a small constant amount of each other.
- Two `ProgramPrint` instances are similar to each other *iff*
 - One program is a subset of the other, i.e. a 100% match of nodes.or
 - They have the same amount of methods and functions.
 - If the programs are handwritten, both match each other in at least 70% of all nodes.
 - If the programs are generated by a frontend, one of the two matches the other in at least 70% of the method and function nodes. Other nodes are ignored since many frontends generate large preambles containing the same domains and fields, which would inflate the match percentage between vastly different programs.

If no entry is found that is determined to be too similar, all the generated case classes are inserted into the database and the processing script starts from the beginning.

Store Only

A config flag called `STORE_ONLY` can be enabled in case constant resources can't be guaranteed by the server. If this flag is set to true, the verification and part of the filtering stages are skipped. Only the `ProgramEntry` and `ProgramPrintEntry` are created. Similarity checking is then reduced to only those two tables. If this is passed those two case classes are again inserted into the database.

3.5 Query Frontend

For future evaluation of the collected data, a simple way to request specific information is a necessity. Collection of the data also requires some way to address our backend from the user frontend instances.

To address this, we implemented a web server using *Cask*[21], a Scala HTTP framework. This server provides a set of end-points that can be queried for information or start specific jobs on our backend. Queried and response data is serialized and deserialized through JSON. Here we give a non-exhaustive overview of some of the queries and services the server provides.

- Request a list of `ProgramEntries`, filtered by metadata such as program length, the frontend used to generate the program, whether it has any parse errors and more.
- Request a list of IDs for programs that fulfill different conditions, such as programs that have a `SiliconResult` / `CarbonResult` with specific verification results, errors, runtimes, or programs that have a specific feature value. For any list of IDs, the corresponding `ProgramEntries` can also be requested.
- Request more information about specific `ProgramEntries`, such as their `VerResults`, or feature values.
- Request a summary of differences between two verifier versions (more detailed explanation in subsection 3.7.2) or invoke benchmarking of specific verifier versions.
- Match a `RegEx` string on all programs in the database (more detailed explanation in subsection 3.7.1)
- Submit a new program to the database. This end-point is used by all submitters referenced in section 3.6

JSON deserialization of the requested data requires some effort on the side of the end-user and leads to code duplication. To solve this, we implemented a query frontend.

This is a lightweight project intended to be included in any future project analyzing our data. Instead of writing their own serializers and query functions or downloading the larger backend, users can simply add this frontend as a dependency. It contains pre-written request functions that query the corresponding end-points and handle data serialization and deserialization. Any relevant custom datatypes used in our backend are also made available.

3.6 Frontend Modifications

To collect user programs and metadata from the various frontends to populate our database, we needed to modify the frontends with the capability of collecting and submitting this information. For this, we developed a trait `ProgramSubmitter` with multiple implementations, whose job it is to collect the information required for a submission query during and after the run of a verification. This code is implemented in Silver, the project for the Viper intermediate language, since it is a dependency of all other projects. These implementations can be called directly in most of the frontends and verifiers. Here we explain some of the changes we made to the frontends to enable submissions.

User consent is vital and no data should be collected without the explicit agreement of the program author. No identifying information apart from the plain program text is stored either. We therefore added a flag called `--submitForEvaluation` (exact name depending on the respective programming language naming conventions) to all frontends and verifiers, without which no submission occurs.

Silicon has a frontend `SiliconRunnerInstance` which is invoked when a Viper file is manually verified. Carbon has a corresponding frontend called Carbon. The only modification taken was the addition of a `ProgramSubmitter` instance to their main methods, which utilizes the frontend's utility functions to collect the necessary metadata.

ViperServer has its own backend implementation which invokes a custom `SiliconFrontend` or `CarbonFrontend` respectively on each verification request. The only modification is the addition of a `ProgramSubmitter` to this backend, which gathers its information from the custom frontends.

In Gobra, verification is commonly invoked on multiple packages, i.e. multiple Gobra files. The verifier starts a job to verify each package, all of which are run in parallel. For each job, a `ProgramSubmitter` instance is added to a global map, identified by the package identifier. Once results are returned, the submitter is updated, sends a submission and is removed. Gobra supports *chopping*, a verification method where each Viper file is split into multiple parts. Due to the asynchronous way Gobra operates, there is no

single synchronization point during verification where all necessary information for our submitter, such as verification results for each chopped file, is available. If chopping is enabled, submissions are therefore disabled.

Nagini, although written in Python, provides access to a JVM instance through a compatibility layer, through which it invokes Silicon or Carbon. This gives us the ability to directly call a `ProgramSubmitter` instance in Python, removing the need for further modifications.

In Prusti JVM access is also supported, however a bit more involved. For simplicity, we therefore implemented an equivalent `ProgramSubmitter` in Rust itself. This is invoked on each verification request unless the result was cached to reduce load on our backend, since a cached program would most likely be declared as a duplicate anyway.

3.7 Extensions

After completion of the core goals, we further implemented two extensions for the collection tool, which we describe in this section.

3.7.1 Pattern Searching

One of the extension goals aims to develop a tool to search for specific code patterns in the programs in our database, providing summarized information about the identified programs. The primary purpose of this tool would be identifying specific code generation patterns from the various frontends or finding potential computationally expensive statements in programs. To develop this tool we mainly considered two approaches, which we will shortly explain here.

Fingerprinting

Our first idea was to repurpose the fingerprinting algorithm from subsection 3.2.2. We could sanitize the program ASTs by removing nodes that do not represent structural properties and only keep nodes like control-flow statements or declarations. One could then submit a pseudo-Viper program also only containing these node types, which would then be compared to all stored programs as before. An advantage to this approach is that we could reuse a large portion of our code, however, there are also several drawbacks. Due to the low number of remaining nodes, our comparisons could report a lot of false positives. Choosing exactly which nodes are relevant for pattern detection was also difficult at this point in time, since we did not have any explicit examples yet that we could conform to. Furthermore, this would require storing an additional fingerprint, increasing the size of our database.

Regular Expressions

Our second idea and the approach we implemented in the end was RegEx. Matching regular expressions on our stored plaintext programs posed several advantages. Regular expressions are more powerful than simple node matching, since any pattern that could be expressed as a regular language (or even a non-regular language using backtracking) could be searched for. Due to the fact that regular expressions can be pre-compiled to state machines using the standard Java RegEx library, matching is also incredibly efficient and requires no additional storage space. This approach also requires no further processing of the programs on our side. The only downside of regular expressions is the unintuitiveness of their syntax, requiring more effort from the users writing the requests.

To access this tool, we added end-points to our web server, allowing users to query a RegEx string and options. Upon reception of such a request, a state-machine is compiled from the RegEx string which all programs in the database are matched on. Then either a list of IDs of programs containing matches or a more detailed summary containing all match indices per program is returned.

3.7.2 Version Benchmarking

The second extension goal aimed to measure performance differences resulting from changes made to the Viper source code. This could give insights into whether changes or additions to the verifiers resulted in performance improvement or degradation.

To implement this feature we added another end-point to our web server, which allows a user to submit a specific Git version hash for either Silicon or Carbon. Upon reception, the local verifier is switched to the specified version and recompiled. All programs for which no SiliconResult or CarbonResult with this hash is present in the database then go through the verification stage again. The generated VerResults and Features are then inserted into the database. At the end, the verifier is switched back to the previous version. Since we want to benchmark submissions using the newest verifier versions, any further submission processing is stopped during this process.

For comparison, users can request version difference summaries from the web server, given two specific verifier versions. The summary is generated for programs that have VerResult entries in the database for both version hashes. Summaries contain the following information:

- A list of IDs of programs that have a different verification result across versions.
- A list of IDs of programs whose runtimes vary by more than 50%.

3. DEVELOPMENT

- A list of IDs of programs with different verification errors.
- The average runtime of programs per version.
- The average runtime variance per version.

Chapter 4

Evaluation

Given the planned continuous operation of our tool, ensuring its capability to deal with incoming tasks in a timely manner is vital. In this chapter, we first take a look at parts of the tool prone to creating bottlenecks by measuring their performance under a probably slightly overestimated anticipated workload. Afterward, given the inherently subjective nature of deciding similarity between programs, we take a look at the process we used to ensure the correctness and define the behavior of our fingerprinting algorithm developed in section 3.2.

4.1 Performance Measurements

Our data collection tool is designed for uninterrupted use over an indefinite period. As such it is important to ensure a sufficient throughput to handle incoming submissions without creating a large backlog of tasks. Since certain aspects like the speed of database operations or the time taken by the verifiers are beyond our control, it is even more important to minimize the overhead introduced by our code. In this section we will analyze the performance of some components we suspect are most likely to create performance bottlenecks. All measurements were generated on a base-model MacBook Pro (16-inch, 2023) with the following specifications: M2 Pro chip with a 12-core CPU and 19-core GPU, 16GB unified LPDDR5 memory and a 512GB SSD. Performance on a dedicated server will likely be better or at least match these results.

4.1.1 Fingerprinting Performance

Before a processed submission will be inserted into the database it has to be matched against all potential similar entries. This process is linear in the size of the dataset and as such it is of importance that this check can still be performed in a reasonable timeframe when the dataset grows to tens

4. EVALUATION

of thousands of entries. The most resource-intensive part of the similarity check is the `ProgramPrint` comparison. To measure the performance of our implementation, we built two datasets. The first one, referred to as *Viper*, consists of 500 handwritten *Viper* programs, collected from a web instance provided by ETH of the *Viper* tool stack. The second one, referred to as *Frontend*, consists of 100 programs generated by different frontends, taken from the Silicon test set. For reference, the average file in *Viper* has a length of 37 lines and 1435 in *Frontend*.

First, we measured the time to generate a `ProgramPrint` for each program in the datasets. Figure 4.1 shows a histogram of the measured runtimes. The median runtime for the *Viper* dataset is 0.522ms and 7.019ms for *Frontend*. Some outliers took up to half a second, however, since the `ProgramPrint` only has to be created once, this is a non-factor.

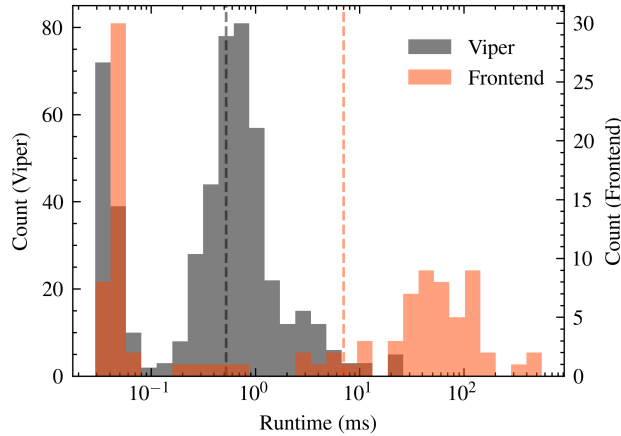


Figure 4.1: Histogram of the runtime taken to create a `ProgramPrint` of 500 handwritten *Viper* programs and 100 frontend generated programs. Vertical lines represent the dataset median.

Second, we measured the time to match all programs in a dataset to every other program. This does not include the generation of the respective `ProgramPrints`, these were pre-generated and stored locally. Figure 4.2 shows a histogram of the measured runtimes. The median runtime for the *Viper* dataset is 0.004ms and 0.147ms for *Frontend*. Extrapolating from this, if an average frontend-generated program would have to be compared against 10'000 programs of similar size, the `ProgramPrint` matching part of this operation would take around 1.5 seconds. In practice, this number is probably far lower, as potential matches are filtered by multiple criteria before any comparison even begins.

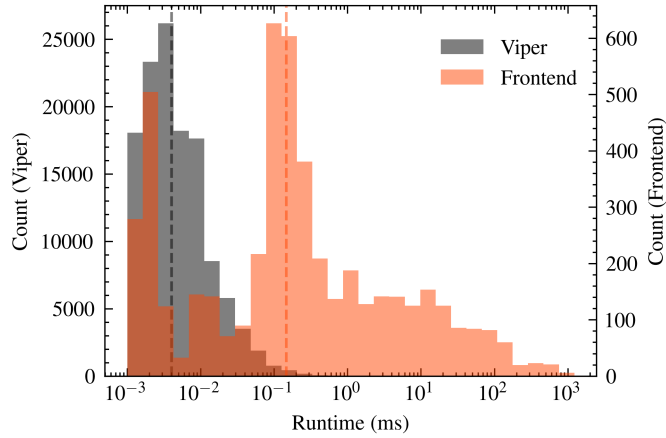


Figure 4.2: Histogram of the runtime taken to all-to-all compare ProgramPrints of 500 hand-written Viper programs and 100 frontend generated programs. Vertical lines represent the dataset median.

4.1.2 Processing Performance

A low overhead of the entire processing pipeline is key to preventing accumulation of submissions. The time taken to verify the program through Silicon and Carbon is inevitable, therefore the overhead introduced by the first and last stage is of special interest.

To measure this, we devised a test using the *Viper* dataset. Any overhead is most likely to disproportionately show up during the processing of these programs since they are very short with an average length of 37 lines. To measure performance we selected 10 programs that are verifiable through both Viper and Carbon. The other 490 were then inserted into the database manually to create entries to compare against during the filtering stage. The 10 programs were then submitted to a local instance of the web server and the time spent in each stage was measured. Figure 4.3 shows the measured results. On average the verification stages took up 77.2% of the time, far more than the other two stages. Since the *Viper* dataset was created by users and not manually curated it contains a lot of duplicate programs. This results in the filtering stage being very short, since a duplicate can be found rather quickly. However, even if this was not the case, the previous subsection shows that the runtime of this stage would not increase by more than a few seconds. The entry and fingerprinting stage is deceptively long, as over 95% of that stage is spent on setting up a connection to the database, querying the submission and its deletion. This duration stays almost constant for programs of any size and should therefore not be of significance. This leads us to the conclusion that even in the worst case of a very small program, the overhead introduced by our processing pipeline is more than acceptable.

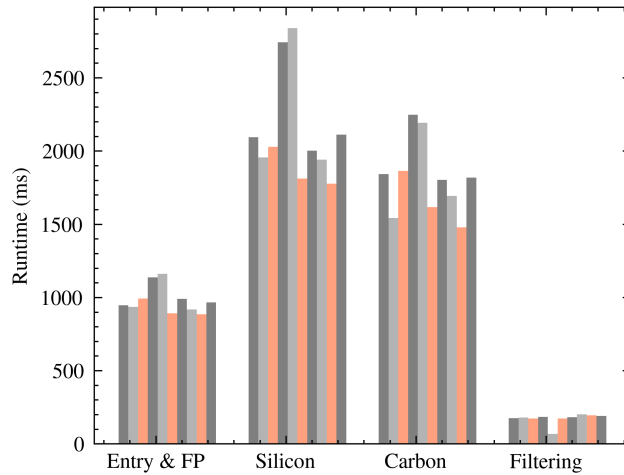


Figure 4.3: Time spent in the processing stages for 10 different handwritten Viper programs.

4.1.3 Pattern Searching Performance

For the web server it is key to minimize response times. Most queries are limited by the speed of database operations since they are simple wrappers around an SQL query. The only further processing taken is the JSON serialization of the data, which is linear in the amount of memory used by the data.

The queries introduced in subsection 3.7.1 however perform a local search of the RegEx pattern against the entire database. Since RegEx with extensions such as backtracking is theoretically unbounded in its temporal complexity, we can give no guaranteed upper bound for response times. Nevertheless, a simple pattern should still return a response in a reasonable timeframe. To check this, we performed two tests on 50 copies of the *Frontend* dataset, i.e. 5000 programs with an average length of 1435 lines.

The first test was performed on the dataset alone. `"{.*\\(.*\\).*}"`, an expression that matches any substring containing parentheses enclosed in curly brackets, was matched on the entire dataset. This operation resulted in 241'500 matches and took 3.156 seconds or 0.631ms per program, averaged over 10 runs. Since this process is parallelized, measuring runtime per program is not possible without affecting the overall performance. Therefore we cannot create a histogram of singular programs here.

The second test was performed as an integration test. The 5000 programs were inserted into an instance of the database and the actual query endpoint was used to perform the matching operation using the same RegEx as before. This operation took 7.450 seconds or 1.490ms per program, averaged over 10 runs, indicating that database operations are responsible for around

57.7% of the runtime. Since we do not expect this pattern searching query to be used with a high frequency, this response time should be acceptable.

4.2 Similarity Detection Testing

During the development of the tool we continuously wrote unit tests to ensure the correct behavior of our code. Most of these are basic and not worth showcasing on their own. To ensure correct behavior of our fingerprinting and over-arching similarity algorithm however, we designed a more extensive “specification” of behavior it should display. This is in the form of a collection of 30 program pairs, each of which was marked as either *matching* or *non-matching*, which describes the output expected by applying the similarity rules for `ProgramPrints` described in subsection 3.4.1. These pairs are further split into code that was generated by frontends, usually multiple thousand lines long, and shorter handwritten Viper programs. The former will from here on be referred to as *Frontend* in labels and the latter as *Viper*. Apart from correctness, this collection was also used to pick subjective values in the algorithm, such as the minimum matching percentage of 70%. The algorithm currently classifies all programs correctly and table 4.1 shows the average minimum results of matching the `ProgramPrints` of all pairs. We choose to show the minimum of the two match percentages since to count as a match, both percentages have to be above the threshold.

	Frontend			Viper		
	Amount	Mean	SD	Amount	Mean	SD
Matching	6	82.4	18.5	9	86.1	18.1
Non-Matching	6	43.6	27.2	9	30.4	23.1

Table 4.1: For each combination of test-cases this table shows the amount of program pairs we defined, the minimum mean matching percentage given by the fingerprinting algorithm and the standard deviation.

Each program pair was constructed to reflect a change in syntax and behavior which we either deem to be too similar or different enough that it shouldn’t be marked as a match. Here we show some of these pairs to give an idea of the properties we deemed important. Programs generated by frontends are too long to be displayed here. Figure 4.4 shows that our fingerprinting technique is completely indifferent to syntactic changes in identifiers or whitespace. Figure 4.5 shows a program pair used to test the condition flattening described in subsection 3.2.2. Further matching program pairs include changes such as subsets, reordering of commutative nodes, line deletions, line insertions and more.

4. EVALUATION

Figure 4.6 shows a pair that we do not want to match. Even though they look similar at a glance, they operate on different types with different binary operators. Further non-matching program pairs include changes such as code repetition, keeping the same domain but changing methods, variable type changes and more.

```
1  method sum(n: Int) returns (res: Int)
2    requires 0 <= n
3    ensures res == n * (n + 1) / 2
4  {
5    res := 0
6    var i: Int := 0;
7    while(i <= n)
8      invariant i <= (n + 1)
9      invariant res == (i - 1) * i / 2
10   {
11     res := res + i
12     i := i + 1
13   }
14 }
```

```
1  method x(z: Int) returns (y: Int)
2    requires 0<=z
3
4    ensures y==z*(z+1)/2
5  {
6    y := 0
7    var w: Int := 0;
8
9    while(w<=z)
10   {
11     invariant w<=(z+1)
12     invariant y==(w-1)* w/2
13   }
14   y := y+w
15   w := w+1
16 }
17 }
```

Figure 4.4: Two programs that should be a match. The second program is semantically identical, only whitespace and identifier names were changed. According to the fingerprinting algorithm, the programs are 100% identical

```
1  field f: Int
2
3  method inc(x: Ref, i: Int)
4    ...
5  {
6    ...
7  }
8
9  method client(a: Ref, b: Ref) {
10   ...
11 }
12
13 method copyAndInc(x: Ref, y: Ref)
14   requires acc(x.f)
15   requires (x != y ==> acc(y.f))
16   ensures acc(x.f)
17   ensures (x != y ==> acc(y.f))
18   ensures x.f == old(y.f) + 1
19   ensures y.f == old(y.f)
20 {
21   x.f := y.f + 1
22 }
```

```
1  field f: Int
2
3  method inc(x: Ref, i: Int)
4    ...
5  {
6    ...
7  }
8
9  method client(a: Ref, b: Ref) {
10   ...
11 }
12
13 method copyAndInc(x: Ref, y: Ref)
14   requires acc(x.f) && (x != y ==> acc(y.f))
15   ensures acc(x.f) && (x != y ==> acc(y.f))
16   ensures x.f == old(y.f) + 1 && y.f == old(y.f)
17 {
18   x.f := y.f + 1
19 }
```

Figure 4.5: Two programs that should be a match. Pre- and postconditions that are *anded* in the second program are listed separately in the first. According to the fingerprinting algorithm, the programs are a 89.8% match. Triple dots “...” represent lines left out for brevity.

4.2. Similarity Detection Testing

```
1 method intOp(a: Int, b: Int) returns (c: Int) 1 method boolOp(a: Bool, b: Bool) returns (c: Bool)
2   requires b >= 0;                          2   requires b != false;
3   requires a > 0;                            3   requires a == true;
4   {                                          4   {
5     c:= b == 0 ? 1 : a * a * (b-1)          5     c:= b == false ? true : !a || a && !b
6   }                                          6   }
```

Figure 4.6: Two programs that should not be a match. Although structurally similar, they operate on different types and use different binary operators. According to the fingerprinting algorithm, the programs are a 0% match.

Chapter 5

Conclusion

At the beginning of this thesis, we set ourselves five core goals to strive after during the development of the data collection tool. In this chapter, we will address whether we achieved these objectives and summarize the contributions made during this thesis. After that, we will provide some potential future applications of this tool and data.

The first goal was the reduction of duplicate data in our dataset. We achieved this through the development of our similarity algorithm, which takes into account the programs' structural properties as well as verification results and metadata to decide on whether a submission is worth keeping or filtering.

The second core goal was to ensure the variety and usefulness of the dataset. We originally planned to achieve this by filtering out programs based on the relative prevalence of their features in the dataset and by excluding programs that do not parse or type-check. However during the development and upon request from group members, we decided on keeping these submissions as they might still provide utility in the future. Through the features and metadata we collect, undesirable programs are still easy to filter out in case they will not be needed for a certain application.

The third core goal was to improve the usefulness of our data by keeping track of additional features of the programs that might impact verification performance. Here we focused mostly on syntactic features as described in section 3.3. We also implemented code to track more detailed Silicon benchmarks once the benchmarker is ready. Through the design of our database tables and the FeatureGenerator, tracking additional features in the future is very simple.

The fourth and fifth core goals focused on the implementation of the backend and a frontend to submit programs. The backend we implemented consists of a docker container hosting our database, the web server for querying

data and submitting programs, and the processing pipeline. For the frontend, we added a `ProgramSubmitter` class to Silver, which we implemented in both verifiers and the Gobra, Nagini and Prusti frontends. In addition, we also developed a query frontend to facilitate searching and filtering the data which will be collected.

After completion of the core goals, we additionally implemented two extension goals.

The first was to make the programs searchable for code patterns. We achieved this by adding a `Regex` matching functionality to the backend, which can be queried through the web server.

The second was to find a way to measure performance differences across Viper versions. For this, we extended the backend with a version benchmarker which can be invoked through a query to the web server. This verifies and benchmarks all programs with a specific verifier version. A difference summary in verifier versions can then be requested from the web server.

5.1 Future Work

In this section, we will address ways to expand the functionality of the tool developed in this thesis in future work and potential uses for the data that will be collected.

The features measured and generated by the `FeatureGenerator` are currently mostly focused on the syntactic properties of a program. For Silicon, only the benchmarking results are additionally collected. Although probably not the primary focus, these tracked features can be expanded in future work in case more information about the behavior of a specific backend is needed. The `SymbExLogger` mentioned in section 3.3 should be of great use for extracting more information about the Silicon verification process.

Another use case and the main reason for the development of this tool is the identification of performance bottlenecks in the Viper infrastructure. We can currently imagine a number of approaches for which our data would be helpful to achieve this.

- Looking at badly performing programs, analysis of the generated features could be used to find such sources. One might especially want to look at correlations between different features and the overall runtime, as certain combinations of these could disproportionately affect verification performance.
- One could use the data generated by the benchmarker to find a correlation between the performance of specific verification stages and the

rest of the data stored about a program.

- The pattern searcher could be used to find common code patterns generated by specific frontends that might be non-optimized and which introduce performance degradations.

Bibliography

- [1] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016.
- [2] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.
- [3] VerifiedSCION. <https://www.pm.inf.ethz.ch/research/verifiedscion.html>. Accessed: 2024-02-06.
- [4] Peter Müller. Lecture slides 5: Modeling - formal methods and functional programming, 2022.
- [5] Peter Müller. Building deductive program verifiers, December 2018. Accessed: 2024-01-29.
- [6] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

- [7] Viper Logo. https://www.pm.inf.ethz.ch/research/viper/_jcr_content/pageimages/imageCarousel.imageformat.lightbox.883770256.png. Accessed 2024-02-12.
- [8] Malte Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. PhD thesis, ETH Zurich, Zürich, Switzerland, 2016.
- [9] Stefan Heule. Verification condition generation for the intermediate verification language sil. Master’s thesis, ETH Zurich, Zürich, Switzerland, 2013.
- [10] Viper Tutorial. <http://viper.ethz.ch/tutorial/>. Accessed 2024-02-12.
- [11] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.
- [12] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, 2012.
- [13] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. Gobra: Modular specification and verification of go programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 367–379. Springer, 2021.
- [14] Vytautas Astrauskas, Aurel Bílý, Jonás Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. The prusti project: Formal verification for rust. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *Lecture Notes in Computer Science*, pages 88–108. Springer, 2022.
- [15] Marco Eilers and Peter Müller. Nagini: A static verifier for python. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 596–603. Springer, 2018.

- [16] ViperServer. <https://github.com/viperproject/viperserver>. Accessed: 2024-02-19.
- [17] Viper IDE. <https://github.com/viperproject/viper-ide>. Accessed: 2024-02-19.
- [18] Morteza Zakeri Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *J. Syst. Softw.*, 204:111796, 2023.
- [19] Slick. <https://scala-slick.org>. Accessed: 2024-02-06.
- [20] Michel Chilowicz, Étienne Duris, and Gilles Roussel. Syntax tree fingerprinting for source code similarity detection. In *The 17th IEEE International Conference on Program Comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17-19, 2009*, pages 243–247. IEEE Computer Society, 2009.
- [21] Cask. <https://com-lihaoyi.github.io/cask/>. Accessed: 2024-02-06.



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

- I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies¹.
- I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies².
- I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies³. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

Development of a data collection tool for the evaluation of a deductive verifier

Authored by:

If the work was compiled in a group, the names of all authors are required.

Last name(s):

Hostettler

First name(s):

Simon

With my signature I confirm the following:


- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date

8046 Zürich, 20.02.2024

Signature(s)



If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

¹ E.g. ChatGPT, DALL E 2, Google Bard
² E.g. ChatGPT, DALL E 2, Google Bard
³ E.g. ChatGPT, DALL E 2, Google Bard