ETH zürich

Program Verification in Continuous Integration Workflows

Bachelor's Thesis

Johannes Gasser

March 17, 2022

Advisors: Prof. Dr. P. Müller, L. Arquint, J. C. Pereira

Department of Computer Science, ETH Zürich

Abstract

Despite the strong correctness guarantees formal verifiers provide, they are not widely integrated into development workflows. A possible integration is to verify projects as part of the continuous integration process. Like with unit tests, developers could gradually add specifications to their software, which automatically gets checked on new contributions.

In this thesis we propose a configurable, yet streamlined way to integrate Gobra, a verifier for Go programs, to continuous integration workflows of Go programs. To speed up the verification of unchanged project parts, we cache verification results for future verification runs. Additionally, we generate reports, containing useful summaries of verification runs.

The changes presented in this thesis provide support for efficient caching and for creating useful summaries of verification runs. Additionally, caching is shown to significantly decrease the verification time, especially when changes to a codebase are small.

Contents

Chapter 1

Introduction

Continuous integration (CI) has become an important part of software development processes. Pipelines are run on contributions to a project to ensure the quality of the software. Typically, tests are used to assess the reliability of programs. Ideally, new tests are introduced when new features are developed or when new bugs get identified. However, tests are not always enough to ensure the desired reliability of software since they only explore a subset of the possible software behavior.

Contrary to testing, formal verification techniques provide strong correctness guarantees. In this setting, programs are annotated with specifications, such as pre- and postconditions. These annotations describe the correct behavior of the program. Then, program verifier automatically check whether the code meets its given specification. Unfortunately, program verification is seldom used in continuous integration.

This process, integrated in CI workflows, would make sure that a program stays in a verified state after every contribution. In this thesis, we focus on Gobra [1], a program verifier for the Go language. We developed a straightforward way to integrate Gobra into CI workflows of Go projects on GitHub. Based on our previous experience with verifying large programs, we identified issues when using Gobra in a CI context that we want to address in this thesis. Firstly, the verification of the whole project may take up to multiple hours to complete. Long workflow times like these, stall the development process, since this will increase the time it takes to publish software or to identify issues. Secondly, Gobra currently does not support all of Go's language features. It can not be run successfully if any part of the code relies on a yet unsupported feature. As such, code that relies on these features, has to be commented out before verification. This is cumbersome if the same code is expected to be compiled after verification. And lastly, developers who use Gobra would benefit from informative reports and statistics for their verification runs.

In this thesis, we developed a configurable, yet easily integratable solution that

addresses all aforementioned problems. To demonstrate the effectiveness of our solution, we evaluate it on the VerifiedSCION project, the biggest codebase currently verified with Gobra. Our experiments show that our changes significantly decrease the verification time in a CI context, especially if changes to a codebase are small.

The contributions of this thesis are summarized as follows:

- We provided a GitHub action that uses Gobra to verify Go programs on GitHub.
- We extended Gobra to be able to cache verification results on a function level that are reused for subsequent verification runs.
- We introduced the trusted flag to Gobra's specification. Functions annotated with it are not verified by Gobra, but their specification is still used for other functions' verification. This way unsupported features don't need to be commented out anymore.
- We introduced a statistics feature for Gobra that collects data about the state of the verification in a project and prints it in a report. Additionally, we export all the data we collected to a file, to allow detailed analysis of a verification run.
- We deployed our changes to the CI workflow of the VerifiedSCION project.

Chapter Overview

In Chapter 2, we provide the necessary background information about the various technologies and concepts we use. We then take a deeper look at the problems we are facing and offer solutions to them in chapter 3. These solutions are discussed in more detain in chapter 4, where we provide additional information about the implementations and the limitations. We evaluate our contributions with the VerifiedSCION project in chapter 5. Lastly, we draw conclusions about our work and discuss possible future work that could build on top of this project in chapter 6.

Chapter 2

Background

In this chapter we provide the technical background for this thesis. Firstly, we introduce Viper [2] and Gobra, the verification infrastructure we will focus on in this thesis. Then, we shortly introduce continuous integration and present the tools used during the project. Lastly, we take a look at VerifiedSCION, which serves as motivation and evaluation of this thesis.

2.1 Viper

Viper is a permission-based verification infrastructure. It consists of an intermediate language, which allows the definition of specifications as annotations for methods and functions. These specifications are then verified by a Viper backend, which leverages a Satisfiability Modulo Theories (SMT) solver to check whether the program behaves according to its specification. Since SMT is an undecidable problem, such solvers may take an arbitrary time to return a result, depending on the input. The last part of the tool chain are Viper frontends, which translate different source language to the intermediate language.

Gobra

Gobra is a frontend of the Viper verification infrastructure for the Go programming language. It takes a Go program annotated with assertions such as pre- and post-conditions and loop invariants. Listing 2.1 shows such a Go function, annotated with specifications. This Go program then is translated to a Viper program. If this Viper program is correctly verified, then the original Go program is guaranteed to behave according to the provided specification. If Viper finds an error while verifying the program, this error is backtranslated and presented by Gobra at the level of the original Go code

```
1 //@ requires n > 0 && m > 0 // precondition
2 //@ ensures sum > 0 && sum == n + m // postcondition
3 func positiveSum(n int, m int) (sum int) {
4   return n + m
5 }
```

Listing 2.1: Go program annotated with pre- and postconditions

2.2 Docker

Docker is an open source platform that allows to pack applications with all their dependencies into images. These images can be run as independent containers on every platform that also runs Docker. Docker images are defined in Dockerfiles, which consist of a list of commands that lead from a base Docker image to the finished image. One should note the difference between Docker images and Docker containers. When we reference Docker images, we mean the software package resulting from a Docker build. When we reference Docker containers, we are talking about running instances of a Docker image.

2.3 Continuous Integration

Continuous integration (CI) describes the process in software engineering, in which developers simultaneously and frequently add to a shared repository. To ensure quality of a repository, automatic pipelines are run on contributions to run builds or execute tests and report the results. This way, build or test failures can be tracked to the contribution that introduced them. Ideally, pipelines should be reasonably fast, so the development process does not get stalled by long build and test cycles. GitHub, one of the largest public source code hosts, allows developers to design CI pipelines in the form of workflows. Workflows are composed of multiple jobs that run in parallel.

2.3.1 GitHub Actions

GitHub Actions provide a way of defining work steps for jobs. They are predefined, configurable tasks that may be published to the GitHub marketplace, such that other developers can benefit from them. Jobs can be defined by using a GitHub action instead of writing it on your own. Therefore, it makes integrating common CI tasks such as automated builds or tests easier. There exist three types of GitHub Actions:

- **Docker Actions** A Docker image can be provided as a base to this action that is used to execute a script or a series of commands in a container generated from it. Therefore, users can use arbitrary Docker images, which

can satisfy almost any software dependencies, one can have during CI processes.

- **JavaScript Actions** JavaScript Actions provide a way of solving CI tasks in a high level programming language. They are generally faster than Docker Actions, but provide a limited tool set since they are bound by the capabilities of the language.
- **Composite Actions** This form of action can be used to bundle multiple workflow steps into a single action.

2.3.2 Related Work

There exists some related work [3] in which formal methods are integrated in CI processes as part of an industrial experiment. They integrated formal methods into the workflow of developers by adding specifications directly to the source code. Additionally, proofs are reviewed and discussed as part of the normal development workflow. Lastly, they provided a CI integration of their proof system that makes sure proofs are always executed after changes. Their findings suggest, that integrating formal verification into CI processes and even developers workflows can be practically done. Moreover, because formal methods were integrated into the development workflow, developers started taking part in writing and reviewing specifications as they became more accustomed to the process. Additionally, it increased the rate at which bugs were found and fixed. These results serve as a motivation for this thesis, as Gobra may also benefit from integration into the development process of Go developers.

2.4 VerifiedSCION

VerifiedSCION is a project with the goal of verifying the network protocol SCION from the high-level design all the way down to the implementation. SCION is a clean-slate Internet architecture that provides secure routing and forwarding, alongside a number of other desirable properties. Currently, Gobra is used in a GitHub workflow to verify parts of the flagship implementation of the SCION protocol, which is written in Go. The previous approach provided only limited feedback, did not support caching of verification results and built the Gobra executable as part of its CI.

We use this project to evaluate how our changes impact the usability of Gobra in a CI process as it is currently the largest codebase that is verified using Gobra. Moreover, it provides a broad range of packages, for which the verification time ranges from a few seconds, up to hours. Therefore, it is an ideal candidate to evaluate caching strategies.

Chapter 3

Approach

As described in Chapter 1, the overall goal of this project is to create a GitHub Action that runs Gobra to verify Go projects on GitHub. This way, we can make sure projects that were once verified stay in a verified state by continuously verifying them. We extend Gobra with the following three improvements to make continuous verification more practical. To increase its CI performance, we add a caching mechanism for verification results. Since Gobra currently does not support all language features of Go, we extend Gobra's specifications with a trusted flag. This flag allows a user to assume that a member's specification is correct without checking the member's body. We also extend Gobra's reporting system to provide statistics and warnings about a verification run. While this does not influence the CI directly, it helps to understand which parts of the codebase take longer to verify, which in turn indicates which parts of the code are worth optimizing.

3.1 GitHub Action

Of the existing variants of GitHub Actions, the Docker action is the one that fits our needs the most. That is because it allows us to build a Gobra Docker image in a central repository, which saves us the build time for Gobra when the action is executed. Additionally, we can provide different Gobra versions, packed in different images. This makes testing of new features or versions of Gobra inside of GitHub workflows very easy.

The action consists of shell scripts that invoke Gobra and a configuration file which specifies the interface that is available for workflow configurations. It is built on top of a Linux based Docker image that contains the Gobra executable and tools, such as Z3 [4], the underlying SMT solver for Gobra. A job of a GitHub workflow is meant to combine several GitHub actions or small scripts to achieve its goal. Thus, our action only performs the actual verification. Other actions should be used to set up the correct project structure or persist artifacts. Therefore, users

have maximum flexibility as they can combine GitHub actions in arbitrary ways and configure each of them.

Configurable Options

To provide a high level of freedom to a user, we make most aspects of the Action configurable. Hence, the following configuration options are offered:

- **Caching** Caching can be turned off for very short verifications for which the caching overhead outperforms the benefit.
- **Docker Image** Instead of using the Docker image based on the latest available version of Gobra, the user can manually specify an image. Therefore, containers built from particular branches or forks of Gobra can be used.
- **Timeout options** Since Gobra's verification is powered by an SMT solver, verification might not terminate. For this reason we provide configuration options for timeouts on a package and project level.
- **JVM options** We provide options to limit the resource consumption of Gobra. Since Gobra is executed in the JVM, we can make use of JVM options to specify resource limits.

3.2 Caching

Viper verification backends typically verify programs modularly. They produce verification results for individual parts of the program such as methods, functions and predicates. As long as such a program part and its dependencies do not change, their verification result will stay the same. Therefore, these verification results can be stored for later runs. If a specific program part and its dependencies have been verified before and are up for verification again, the already stored verification results will be presented, instead of recomputing them. Additionally, if a project follows the main CI principles, contributions to it should be frequent but small. Therefore, much of the verification done, can be stored and retrieved again for subsequent verifications. For these reasons we provide a caching strategy for Gobra, which stores the verification results on a Viper level.

There already exists a caching solution for Viper methods in ViperServer. ViperServer is a service that manages verification requests to different backends from the Viper tool stack. Moreover, it provides the functionality to cache verification runs. Thus, to enable caching in Gobra, we added ViperServer as a middleware between Gobra and the Viper backends. In the previous caching solution in ViperServer, the results were stored in memory and applied to subsequent verifications. As this requires a constantly running ViperServer instance, this is not directly applicable in CI settings because the processes are terminated after each workflow run. Thus, we store the cache inside a file, which we reload in subsequent CI runs.

Serialization

To store the cache content inside a file, we have to serialize it and then deserialize it on subsequent runs. We chose JSON as the file format because it is on the one hand lightweight and on the other hand a good fit for key value data like our cache content. There already exist libraries to convert a Scala Object to a JSON String.

The cache content consists of a list of errors in case of a verification failure or an empty list in case of a verification success. These verification errors contain nodes of the Viper abstract syntax tree (AST), which are needed to backtrack an error to the right position. The AST nodes cannot be translated to JSON directly, since they contain positional information which may vary on a new verification run since the Viper program potentially changed. Therefore, we translate AST nodes to a unique identifier that doesn't change between two verifications. To achieve that, we serialize a Viper AST node to its access path. The access path of a node consists of a list of hash sums for all nodes in the path from the AST's root to the node. The calculation of the access path was already implemented in ViperServer, as it was already be done previously to find equal nodes over two different ASTs.

3.3 Trusted Flag

Currently, Gobra does not support all language features of Go. This leads to errors if a project uses unsupported features since they can neither be parsed nor translated to Viper code. To enable verification of such projects, we introduced the `trusted` flag. This flag can be used to assume that the implementation of a function or method follows its specification. Gobra then ignores its body, by letting the parser skip it. Therefore, Gobra won't encounter unsupported features. An example of the trusted flag can be seen in listing 3.1, where the code makes use of closures, a yet unsupported Go feature.

```
1 //@ requires ...
2 //@ ensures ...
3 //@ trusted
4 func main() int {
5     sum := func(a, b, c int) int {
6         return a + b + c
7     }
8     return sum(3, 5, 7)
9 }
```

Listing 3.1: Example of a function annotated with the trusted flag. Without this flag, Gobra will throw an error because it does not yet support closures.

A drawback of the `trusted` flag is that a user can assume arbitrary specifications, even if they do not hold. Therefore, unsound verification results are possible. To limit the disadvantages while still offering the trusted flag, Gobra prints a warning

for every member that depends on such `trusted` flags to make users aware. Besides the aforementioned use-case of overcoming unsupported language features, the `trusted` flag can also be used for functions/methods that should be assumed for now and will be verified at a later point in time.

3.4 Reporting

To make verification results more easily accessible, Gobra prints an informative summary in the end. This summary includes a list of errors and warnings that occurred per package. Additionally, it contains useful statistics about the verification. These statistics include the number of members that were cached, the number of specified members, the number of verified members and the number of errors and warnings. To collect this information, we analyze the call graph of the Viper members that are being verified, after the Viper AST was generated. We are assuming that the dependencies of generated Viper members match the dependencies of the Gobra members they originated from. With this call graph, we can analyze dependencies on trusted or abstract members for which we then print warnings.

During verification, we make use of Viper's reporting system. In this system, messages are passed from the verification backend to the frontend for each verified Viper member. These messages contain information about the relevant members on a Gobra and Viper level, the verification time, the verification result and a flag that indicates whether the result was cached or not. We make use of this information to gradually collect statistics about the verification run.

Visualization

The statistics that we collect are exported to a data file at the end of a Gobra run. Again, we make use of the JSON format as a straightforward way to make this data available to as many platforms as possible. The exported JSON file, contains a graph with nodes for each Gobra and Viper member with all the data we collected attached to them. This graph can be used for visualization of the call graph and analyzing the verification run in detail.

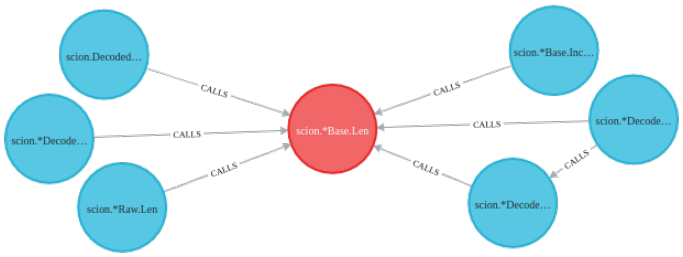

Figure 3.1: Visualization of Gobra members, depending on an abstract Gobra member(red).

As a proof of concept, we use Neo4j Desktop[5], a tool used to analyze and visualize graph data. On the resulting graph database, we performed dependency analysis at the Gobra level. This is done by isolating nodes of interest such as trusted or abstract Gobra members and visualizing their neighborhood, as can be seen in figure 3.1. Additionally, we could analyze which members at the Viper or Gobra level caused long verification time or did not verify successfully.

Chapter 4

Implementation

In this chapter we present details about the implementation of the GitHub action and Gobra extensions described in Chapter 3. Furthermore, we discuss limitations of our solutions.

4.1 GitHub Action

The GitHub action consists of three parts. Firstly, the Docker image in which the action is run. Secondly, the action configuration which defines the interface available in workflow configurations. And lastly, the task that gets executed inside the Docker container.

4.1.1 Docker Image

Before this thesis, there already existed a Docker image for Gobra. This image was built by installing all build dependencies of Gobra and then assembling it. This leads to large images because Docker images are built in layers that contain all changes that happen in a build step. Therefore, steps that don't clean up dependencies, which are not needed at runtime, bloat the Docker image. Because of this, the resulting image was around 1.3 GB large.

As an alternative to cleaning up every build step, Docker provides multi-stage builds [6]. The key idea is to use multiple `FROM` clauses. The Docker build process then starts with a clean image on each `FROM` clause and allows copying artifacts from previous steps. As can be seen in listing 4.1, we use the Java Development Kit (JDK) as the base image of our build which has a size of roughly 215 MB. However, for the final Docker image, we then use the Java Runtime Environment (JRE) as the base image, which has a size of roughly 76 MB. This way, we build Gobra in a build environment and then copy the created artifacts to a slim runtime environment. Therefore, we save a lot of space as the resulting image contains

neither build dependencies nor the source code. The size of the final Docker image for Gobra was reduced this way by roughly two thirds, to 422 MB.

```
1 FROM openjdk:11-slim as build # <- use larger build environment
2 # Steps to build Gobra
3 FROM openjdk:11-jre-slim # <- use runtime environment
4 # Steps to prepare the runtime environment and copy the build
    artifacts from previous stages
```

Listing 4.1: Excerpt of a Multi-Stage Docker build configuration

Another issue of the previous approach was the deployment of the finished image. Previously, the image was exported with the `docker save` command as a `tar` archive at the end of the GitHub workflow in which it was built and published as an artifact. To then use this image, one has to download the archive and load it into Docker with the `docker load` command. This works around Docker's caching mechanism that stores image layers and only downloads layers that did not change since the last download. Therefore, the Docker image would have had to be fully downloaded on every action run which would lead to unnecessary delays. Docker registries provide a powerful alternative as a central hub, where Docker images may be stored. When using registries, images can be downloaded with the `docker pull` command, which downloads the image layer by layer. Therefore, Docker can cache layers that did not change since the last download. Fortunately, GitHub provides a Docker registry for public repositories [7]. Therefore, we decided to publish Gobra's Docker image in this registry. As a positive side effect, this provides an easy way of managing different Gobra versions thanks to Docker image tags.

4.1.2 Configuration

We previously described the configuration options we want to include in the Gobra action. In this subsection, we discuss relevant implementation details for some of them.

Global timeout

We implemented the global timeout by wrapping the Gobra invocation inside a bash timeout call. When the specified timeout passes, a `TERM` signal is sent to the Gobra process to kill it gracefully. However, GitHub has an own timeout limit for jobs, which is six hours per default. Jobs running longer than that will be killed by GitHub. If longer verifications should be performed, it can be split into multiple jobs. This can be achieved by using the `packages` and `excludePackages` configuration options of the action. They allow a user to select which packages should be verified or which packages should be excluded from verification in a job. Each of these jobs will again have a timeout of six hours. Additionally, these jobs are executed in parallel, something that Gobra in its current state is not able to do.

This can be done multiple times until a workflow reaches the time limit of 72 hours [8].

Parameterized Docker Image

Currently, GitHub does not support variable Docker images or passing build arguments to a Docker build in Docker actions [9]. Therefore, it is impossible to use varying Docker images in a convenient way. Different projects might want to use different versions of Gobra. For example, VerifiedSCION that previously used its own Gobra version since it depended on some features that were not yet released to Gobra's main version. Therefore, we built a workaround for this issue. We changed the action to run in a clean Docker image that contains Docker. Inside the resulting container, we have access to the action's `imageName` and `imageVersion` configuration options. This allows us to pull the desired Docker image and running it inside our intermediate container. We should note here, that all arguments defined in the action's configuration have to be explicitly passed to the Gobra container, as otherwise they will not be available in the container.

This approach allows us to successfully specify the image name and tag of the Gobra image that is run in our intermediate container. However, it comes with two downsides. Firstly, while providing the freedom of choosing to a user, it also allows a user to let the action run with malicious Docker images. We minimize this risk by specifying Gobra's official GitHub repository and the latest image tag as default values. This way, a user would have to explicitly specify a malicious Docker image. Therefore, we have the same security expectations about the Docker image as GitHub has for workflows in general. There, it is expected that a user is aware of what he runs in a workflow. Secondly, pulling the image inside a Docker container that has no access to the host's Docker cache, causes the cache to be cleared after every workflow run. Therefore, the image has to be downloaded again on every workflow run. However, the impact of this has been shown to be in the range of a few seconds. This could be, since the official GitHub workflow runners have a fast connection to their Docker registries.

4.2 Caching

In this section, we focus of the extensions we made to the existing caching implementation of ViperServer. We added functionality to serialize and deserialize the cache and write it to a file. This helps us to store and restore the cache between two CI runs, since ViperServer gets shutdown after every CI run. Apart from that, we did not change much of the existing implementation, because the main focus of this thesis is to enable caching in Gobra and CI. The top level of the cache structure maps a `programId` to a nested map, containing results for a particular verification task. A `programId` is a string, identifying a particular verification task. It is provided by a Viper frontend and should not change between

verification runs, as this would lead to cache misses. The nested map maps a Viper method to a list of `CacheEntry` objects. Previously they consisted of a dependency hash and a `CacheContent` object. We extended it to also contain timestamps that specify when a cache entry was last accessed and when it was created as can be seen in listing 4.2. This information can be used in the future to create a cache eviction policy. The `CacheContent` objects contain the cached verification result.

```
"programId":{
  "`methodHash`":[
    {
      "dependencyHash": "...",
      "content": "...",
      "created": "...",
      "lastAccessed": "..."
    }
  ]
}
```

Listing 4.2: JSON representation of the top level cache structure

4.2.1 Serialization

We decided to use the `net.liftweb.list-json` library for this task because it allows easy serialization and deserialization of arbitrary objects with the help of type hints. Type hints are strings, stored inside the resulting JSON object, which identify the class of the serialized object. The deserializer uses this information to determine to which class the JSON object should be deserialized. Additionally, this library allows implementing custom `serialize` and `deserialize` methods. They can be used to provide custom ways of serializing and deserializing objects. Cached error messages contain nodes of Viper's abstract syntax tree (AST). They cannot be serialized directly, as they contain positional information that changes if the nodes position in the program changes. This positional information is used to determine the position of the error which is why we have to find a workaround. Therefore, we overwrite the `serialize` and `deserialize` methods to transform AST nodes to unique identifiers. They should on the one hand identify a node uniquely while on the other hand be resistant to program changes. As described in section 3.2, we use a node's access path as an identifier. It is the path from the AST's root to a node and will not change between subsequent verification runs as long as the program structure will not change significantly. Therefore, it is ideal as an identifier since it contains enough information to identify a node uniquely while also being resistant to program changes.

The cache structure is deserialized on the startup of ViperServer. This allows to continue with the same cache, the previous run stopped. Since we don't have the AST for a particular verification run available at startup, we can't deserialize

cached errors at that time since they depend on AST nodes. Therefore, we decided to split the deserialization into two steps. In the first step, at startup, the whole cache structure is deserialized except for the `CacheContent` objects. A particular `CacheContent` object is later deserialized once its accessed. Until then, it stays in a serialized state. Since they are only accessed when a program is being verified, we have access to that program's AST. Therefore, we can find the correct AST nodes for the new program and deserialize the errors. As soon as the program's verification is finished, the deserialized `CacheContent` objects are serialized again and stored in the overall cache structure.

4.2.2 Usage during a workflow

GitHub workflows should not take too much time to complete, so they do not stall the development process. Therefore, they will profit from caching verification results. To share the cache between GitHub workflow runs, the generated cache file has to be saved at the end of a verification run and loaded again on the next run. This can be achieved with auxiliary GitHub actions. GitHub provides a default caching action. The problem with this action, is that a cache entry is evicted after seven days. While this will not be an issue for repositories with frequent changes, repositories with infrequent changes will not profit from caches stored this way at all.

Alternatively, the serialized cache can be uploaded as a workflow artifact. GitHub's default artifact action does not allow downloading artifacts of previous workflow runs. However, there exist third party actions that do exactly that [10]. This increase the retention period of the cache to the retention period of artifacts, which is by default 90 days. We did not test this approach during this thesis, but the extensive configuration of the specified action should allow storing and restoring of the correct cache over multiple branches. Ideally the latest cache related to the current branch should be taken for a CI run. One way could be to iterate over the current branch's commit history to find the first commit that a cache artifact was generated for. Thus, always the latest cache file related to the current branch is taken for a verification.

4.2.3 Limitations

Before we can talk about the limitations of the current caching solution, we need to understand how ViperServer retrieves a `CacheEntry` objects for a given Viper member. For an entry to be successfully restored from the cache, two conditions must be fulfilled.

First, there must be an entry for the method's hash code in the map under the `programId` the frontend has specified. This hash code is calculated by taking the string representation of a method and applying a collision resistant hash function to it. Therefore, changing a method without changing its semantics, for example by

renaming it or reordering its statements, will result in cache misses. Additionally, the `programId` must not change between verification runs.

Secondly, the resulting `CacheEntry` object's dependency hash must match the dependency hash that is calculated from the currently verified program. This is done by collecting all Viper methods, functions and predicates, the method is calling. Additionally, all domains and global variables, regardless of whether they are used or not are collected as well. For each of them a hash code is calculated again by using their string representation with the only exception of methods, for which the body is omitted when calculating the hash code. All of these member hashes, joint together and hashed again, result in the final dependency hash. Because of this, the order of top-level Viper structures matter, since their hashes are joint together in the order they are encountered. Moreover, for domains the ordering of their axioms and methods matter as well, since their string representation will change if they are reordered. The ordering of top-level Viper members as well as the ordering of domain members are only syntactic differences to a program that should not result in cache misses.

Gobra does not satisfy all conditions that are needed for the cache to work reliable at the moment. Currently, there exist two big issues that interfere with caching, a Gobra user can't influence. The first one is, that certain Viper member names contain positional information of Gobra members, Viper members generated for interfaces or structs. Therefore, when these members' position in the program changes between two verification runs, Viper members generated for them are renamed which leads to cache misses. The second one is, that the top-level Viper members are not ordered. Thus, the dependency hashes may not be calculated reliably over multiple verification runs. It would make sense to solve the second issue on a Viper level, because the ordering of top-level declaration will not change the program behavior. This could be done by changing the way the dependencies are calculated. Like this, everything depending on this functionality profits since the dependency hashes are more resistant to code changes.

Another issue, which can be controlled by a Gobra user is that the cache depends on package names and their relative location in the project. This, because we build the `programId` with this information. Therefore, if a package is moved or renamed, ViperServer will not be able to use previous cached results for the entire package. Additionally, many names generated in the Viper code also contain this information as well. Thus, when renaming or moving a package, Viper methods generated for it will be renamed, which leads to cache misses for them and members that depend on them.

Currently, the caching also does not check in which Viper or Gobra version a program was verified. This is of concern to us, since verification results that resulted from a bug in Gobra or Viper may be cached indefinitely since the stored cache files can always be reused. Previous to this thesis, this was much less of an issue, since the cache was short-lived, because it was cleared once

ViperServer shutdown. However, this could easily be worked around by changing the `programId` to contain this information. We decided to not include this in the current state, since Gobra doesn't have version information available yet.

Lastly, one has to be careful when dealing with members that would be correct but output errors on some verification runs. Non-deterministic verification results violate the assumption caching is built under that verification results will not change if the program does not change. Once such a non-deterministic error is cached, ViperServer won't try to verify the method again until it has changed.

4.3 Trusted Flag

For the trusted flag, we added a specification keyword to the `PFunctionSpec` definition inside Gobra's AST. The parser first parses the `PFunctionSpec`, then checks whether the trusted flag was set and decides whether it parses the body or not. Because of how Gobra's parsing library handles positioning of nodes, we had to manually update the position of the resulting method or function node since it did not cover the `PFunctionSpec` clause.

With this approach, we completely ignore the contents of a trusted member. This leads to cases where a parsed program would not compile but is still accepted by Gobra, as shown in Listing 4.3. However, this is the desired behavior, since neither Gobra nor its parser can handle these features at the moment.

```
1 //@ trusted
2 func functionWithNonSupportedFeature() int {
3   some non-supported go feature
4   return "Not an Integer"
5 }
```

Listing 4.3: Trusted function with arbitrary body

In the future, once Gobra can parse and type check unsupported features, this behavior will change such that the body gets parsed and type checked, but is ignored for the verification. On the Viper level, a trusted Gobra member gets translated to an abstract Viper method, meaning it has no method body. This allows Viper to include the method's specification for verification of other members that depend on it while not verifying the method itself.

4.4 Statistics Collector

Before this thesis, Gobra did not output much information about a verification run. However, developers integrating formal verification in their project would profit from statistics, describing the current state of the verification. These statistics could include for example information about how many methods have specification or

how many depend on assumptions imposed by trusted or abstract functions in Gobra. To collect statistics such as these, we built the `StatsCollector`, a custom `Reporter` implementation. `Reporter` objects are used to handle messages in the Viper tool-chain. These messages contain information about a verification run and may be sent by Gobra directly or may be passed from a Viper backend. Other parts of Gobra also depend on this messaging system. Therefore, the `StatsCollector` had to be built, such that it does not disturb the flow of messages in Gobra. To achieve this, we implemented it as a wrapper around a different `Reporter` implementation. This way we can capture all messages, extract the data that we need for statistics and then pass the message to the next `Reporter`.

The `StatsCollector` collects statistics by capturing verification result messages we receive. These are received for each Viper method, function and predicate that were generated by Gobra. In addition to the information about the verified Viper member, they contain backtrack information about the internal Viper representation in Gobra and the Gobra member they originate from.

As soon as Gobra finishes generating the Viper AST, a message containing said AST is sent. We use it to build an empty skeleton, in which we will store the data about the verification once it is available. This way we know before the real verification starts which verification results we can expect to receive. Therefore, we can decide after a verification run, for which members the verification did not terminate, since no data will be available for them. Storing whole Viper or Gobra AST nodes in our data would defer garbage collection of the entire AST. Therefore, we do not store them directly, but only extract and store information relevant to the statistics. The data is collected in multiple steps and contains information on a Gobra and Viper level. Firstly, we add all Gobra functions, methods and predicates, for which at least one of the Viper members in the AST was generated. Secondly, we add all Viper methods, functions and predicates, which come from such a Gobra member, as children to the member's data entry. Lastly, we analyze the call graph on a Viper level and add dependencies we found this way to the Gobra entry, by checking to which Gobra members two Viper members with a dependency belong to. This way we can reflect the dependencies that are relevant for verification on the Gobra level.

Most of the interesting data, like the verification time, information about caching or the verification result is stored in the Viper entries of our data. We decided to do this, because it provides a lot of details about the verification, which otherwise would be lost because we would have to aggregate information of multiple Viper entries in a single Gobra entry. This is particularly relevant for Gobra functions that generate multiple Viper members. For example, a Gobra function for which we want to proof termination will result in multiple Viper methods. There, we couldn't determine which of the Viper member's caused a particularly long verification time if we didn't keep the information on the Viper level. However, since Gobra entries contain all Viper entries as children, it is still possible to aggregate this data at a

later time to the Gobra level.

Today, when Gobra crashes or is terminated there is no detailed information about the verification available. However, collected statistics could prove especially useful when Gobra unexpectedly crashes or is terminated. For example, when Gobra has to be terminated because of a time-out, information about which member timed out could be useful to identify the problematic Gobra member and apply a fix. Therefore, we added a JVM shutdown hook that exports the statistics to a file when the JVM shuts down.

4.4.1 Filter Built-In Features

Not all the received messages are relevant for our statistics. Some of them originate from Go features that are directly built into Gobra. They contain functions from the Go standard library for example and are treated as abstract functions with specification. For verification, they are attached to generated Viper programs and also passed to the verification backend. Reporting information about them would lead to unnecessary cluttered reports that are not very useful to a Go developer since they cannot control these built-in features. Thus, we filter these messages out. Since the built-in members are either coming from Gobra files bundled together with Gobra or are directly implemented in Gobra, we have to mark these members at two different locations: Firstly, when collecting Go code that is bundled within Gobra, we mark them with a `isBuiltin` flag that is passed to the package that they result in. During data collection, we check if the reported Gobra member belongs to a package with this flag and filter them out. The second is to filter out all verification result messages, containing internal nodes that extend the type `BuiltInMember` since these are features that are directly implemented in Gobra.

4.4.2 Input handling in Gobra

To collect statistics for multiple packages, we had to extend Gobra to handle multiple packages in a single invocation. Before this thesis, Gobra could only be run on a list of files, belonging to the same package, or a whole package. It was not possible to handle a project that consists of multiple packages in a single Gobra invocation. This made collecting statistics over multiple packages hard since Gobra would have had to be invoked for each of them and the resulting data would have had to be merged afterwards. Thus, we introduced changes to Gobra's input handling such that it can verify multiple packages. We extended the existing input option to accept a directory as input, which is then scanned for all files relevant to Gobra. This is by default done only in the specified directory. Alternatively, there exists an option to traverse the directory recursively and collect all Gobra files in all subdirectories as well. These files are then grouped by a `packageId`, which should uniquely identify a Go package. It consists of the package directory, paired with the package name. The package directory is a relative path, which depends on the

configuration that Gobra is invoked with. It is either relative to a specified project root, if that is not specified, the first of the package include paths, or if that is also not specified, the path in which Gobra is executed from. This way, the `packageId` is independent of the file system except for the last method, where it depends on the location Gobra was invoked from. The way this identifier is generated reflects how packages are organized in Go: A package consists of a collection of files in a single directory with the same package clause. The grouped files are then verified package by package. So overall we didn't change anything about how Gobra verifies files and packages, but enabled Gobra to handle verification of multiple packages in sequence. In addition to these changes, we added options to only verify certain packages or to exclude certain packages in a project by filtering found packages by their name.

4.4.3 Data merging

While collecting statistics we will encounter some Gobra and Viper members multiple times. This happens, because Gobra currently runs a verification task for each package separately. Members of imported packages are imported as abstract Gobra members and then translated to abstract Viper members, such that their specification can be used during verification. Therefore, we will receive messages for these Viper members each time we verify a package that imports the Gobra members they belong to. Because of this, we had to decide if we want to merge the received data into a single data point leave them as it is. To get the most out of the data while keeping the dataset as small as possible, we decided to merge the resulting Gobra entries into a single entry. Since we do not store any statistical information in the Gobra entries, merging them is trivial except when deciding if a Gobra function or method is abstract. Currently, this is done by checking, if that members body is empty. However, since imported members always appear with an empty body and there is no other way to tell if a member is abstract, we had to under approximate this attribute. We only declare a Gobra entry as abstract, if it appears once as a non-imported member without a body. Therefore, abstract members for which the package is not verified but still appear as imported members will not be reported as abstract. To still make users aware of this, we print a warning if a package is imported but not verified. For Viper entries, we decided to not merge them together as it better reflects the behavior of the Viper backend. There, they cause a small overhead in the range of 100 ms, because they are passed to the verifier even if they do not have to be verified. Therefore, we can still capture this overhead and display it in the statistics as this might be relevant to a user.

Chopper Support

Gobra has a feature called chopper that chops a Viper AST into smaller parts. In these chopped parts, the same Viper member might occur multiple times.

However, it is guaranteed that each function or method only occurs once with its body, meaning all other occurrences are abstract. Thus, we have to deal with equal Viper members appearing multiple times during a single verification task. In this case we do not gain much if we keep the information for each Viper member since this only provides information about how chopper split the AST while cluttering the data unnecessarily. Therefore, we merge the data of equal Viper entries, received during the same verification task together into a single data object. Thus, we can still guarantee that a Viper entry only occurs once per verification task. We have taken the following design decisions for merging Viper entries:

- To decide if a Viper entry is not abstract, we check whether it occurs with a body at least once.
- We regard a Viper entry as verified, once we receive a verification result message for it.
- For a Viper entry to be marked as successfully verified, all received verification result messages must report a success.
- A Viper method entry is considered cached, when we receive a cached verification result once for it. This is sufficient in this context, because all other instances of this method have to be abstract and only non-abstract Viper methods are cached.
- While merging a Viper entry, we add up the reported verification times in all result messages for the corresponding member.

4.4.4 Visualizations

In this subsection, we focus on the visualization we performed with Neo4j Desktop. Before we could start with visualizing data, we had to allow Neo4j to import JSON files. To do this, the APOC plugin[11] has to be installed. Additionally, we have to allow APOC to import files. Therefore, we add the following line to the configuration file of the Database:

```
apoc.import.file.enabled=true
```

After that, the data can be imported. We provide a basic import script that imports Gobra and Viper entries as nodes. Moreover, it creates edges for dependencies between Gobra entries. Also, Viper entries get an edge to the Gobra entries they belong to. Lastly, it adds tags for trusted and abstract Gobra entries, for entries that were successfully retrieved from the cache, and unsuccessfully verified Viper entries. These tags are used to color nodes during visualization.

When visualizing data, the Neo4j browser does only display 300 nodes per default. This limit can be increased in the Neo4j browser settings. However, we found that when visualizing graphs with around 1000 nodes, visualization can get very slow. Additionally, when viewing an interconnected cluster of nodes, the visualization

can get quite hard to read, as can be seen in figure 4.1, where we look at the call graph of a whole package.

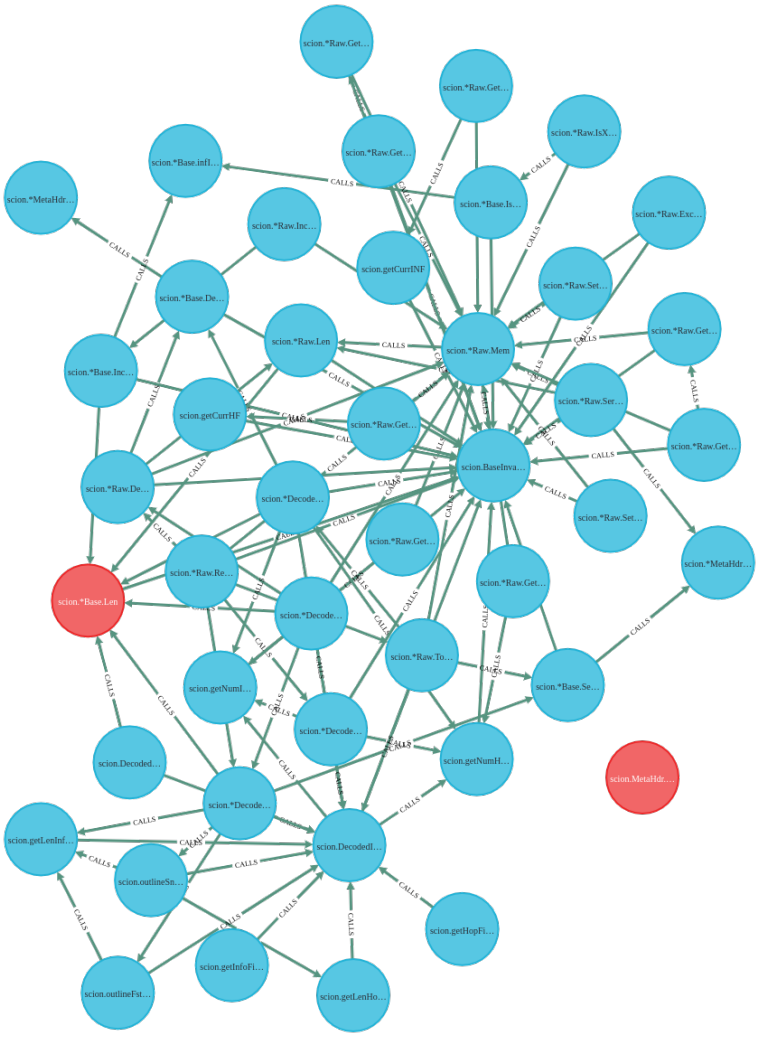

Figure 4.1: Call graph for the scion package of the VerifiedSCION project. It contains many call relations which makes the graph hard to read.

Therefore, we found that when analyzing nodes with the visualization, one should focus on a smaller group of nodes. To provide a start for future users of such a visualization, we provided small snippets of code needed to perform basic analyses such as a dependency analysis of trusted members or to detect nodes that had a long verification time.

Chapter 5

Evaluation

In this chapter we evaluate how our changes impact Gobra when verifying large projects. First, we measure the overhead introduced by our changes on runs of Gobra under different configurations and cache states. With this, we evaluate the tradeoffs between the verification time increases and the features we implemented. Second, we evaluate the effectiveness of our caching strategy. To this end, we verify a project on each commit of an excerpt of a commit history from a large project. As the code base for both parts, we use the VerifiedSCION project as it serves as a motivation to this thesis and is the largest codebase verified with Gobra to date. Additionally, it provides us with a diverse range of packages, some of which take a few seconds to verify and others that take minutes or even hours to verify. We use the master branch of the Gobra fork made for this thesis with commit hash `b0b6dc68` for all tests. It contains the most recent Gobra version at the time of this evaluation, including the caching support and support for collecting statistics. As for the Viper backends, we use ViperServer with the commit hash `46fb237`, Silicon with the commit hash `edb5d079` and Silver with the commit hash `7228e714`. To work around the limitation that caching currently does not work reliably with unordered top-level Viper structures as described in section 4.2.3, we enforced an ordering for them when calculating the dependency hash for methods. The packages `slayer` and `router` take the longest to verify. The chopper feature is enabled for both of them, as this speeds up verification and prevents some errors that are thrown nondeterministically during their verification. Additionally, we had to verify the `conn` package with a timeout of 30 minutes, since either its verification terminates after roughly some minutes or not at all. The whole evaluation was performed on an Arch Linux operating system with an Intel Core i7-5930K processor 3.5 GHz and 32 GB of RAM.

5.1 Overhead

With this evaluation we will show the impact on performance our changes cause in Gobra. To achieve this, we make three experiments on the VerifiedSCION project with the commit hash 804de10c. In the first one, we verify the project with neither ViperServer nor the `StatsCollector` to get a reference value. In the next three, we verify the project again but once with the `StatsCollector` enabled, once with ViperServer using an empty cache and once with ViperServer using a full cache. Each experiment runs on the same Gobra version, as the `StatsCollector` as well as ViperServer are modular and can easily be removed or turned off. Note that we only consider the actual verification time, since only there, the overhead of the `StatsCollector` and ViperServer will occur. Therefore, the times displayed do not include the time it takes to parse, type check and transform the Gobra program to a Viper program. We considered only 25 of the 26 packages in VerifiedSCION. Each of these packages is verified 20 times. Due to time constraints, we do not verify the `router` package, given that it may take up to five hours to verify.

In this experiment we work with relative data, to see if there is an increase proportional to the verification time. However, to put this data into some context, one can see the absolute verification time of each package for the reference experiment in figure 5.1.

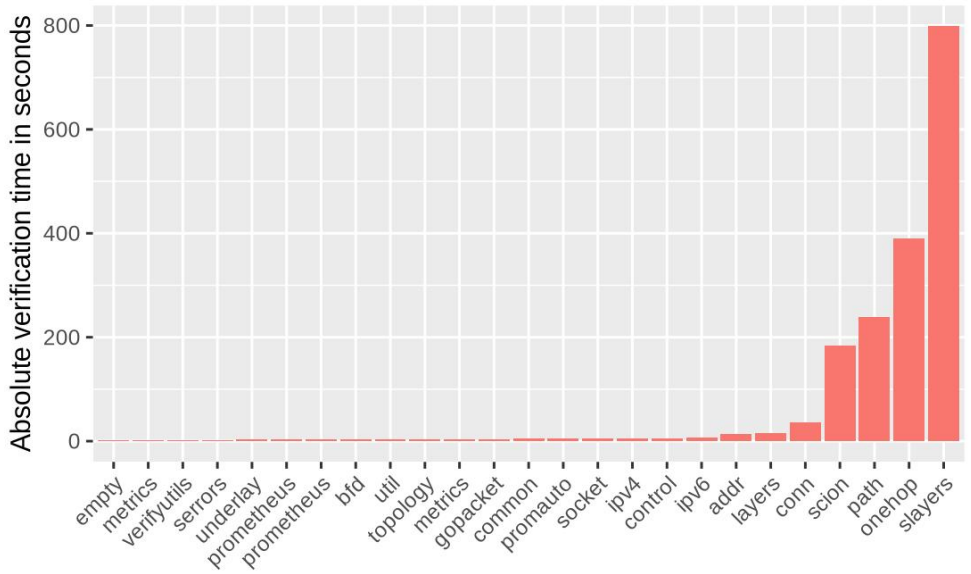

Figure 5.1: Time in seconds for the reference experiment to successfully verify the packages (n=20)

The relative time increases compared to the version with neither `StatsCollector` nor ViperServer enabled can be seen in figures 5.2 and 5.3.

For the `StatsCollector`, we observed a mean increase in the verification time of 2.22% and a median increase of 0.24%. For the `onehop` package, the verification time increased by 43.4% as can be seen in figure 5.3. We observed that this package's verification time lies mostly around three to four minutes but in some rare cases, goes up to 20 minutes or even 40 minutes for the `StatsCollector`. Since this event only occurred two times, it is impossible to say if the `StatsCollector` caused this huge overhead or if it was just a rare event caused by differing runtimes of the underlying SMT solver. However, such a huge overhead caused by the `StatsCollector` would imply that it had to processes a large amount of messages or the process of getting data out of a message took longer in these cases. Both seem very unlikely, as neither the data structure nor the messaging behavior has any non-deterministic part that would explain these increases of time in such rare events. To be completely sure however, the test would have to be run again, preferably with a larger sample size. In some other cases, the verification time can be observed to decrease. Since the `StatsCollector` did not make any optimizations, this is not expected and may result from the runtime differences in the underlying SMT solver.

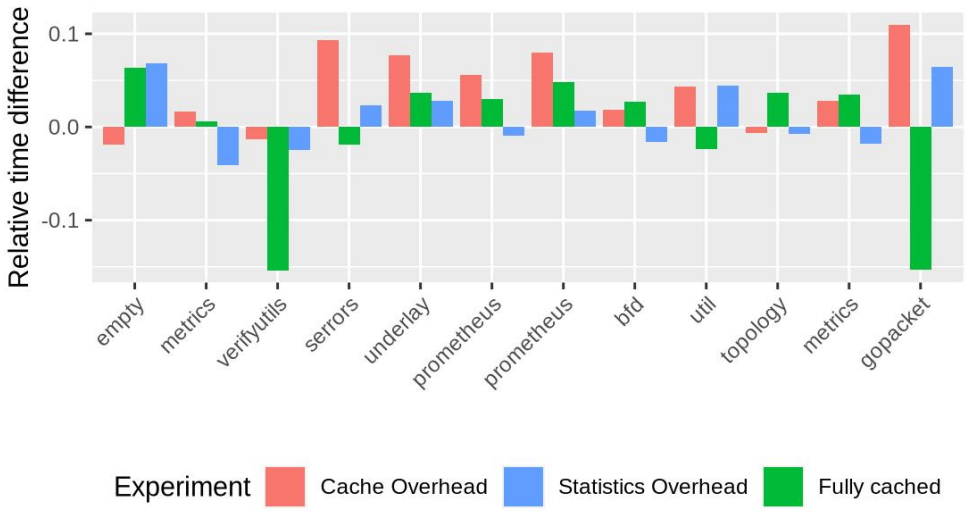

Figure 5.2: Relative time measurements for the first twelve packages (n=20)

For `ViperServer`, we observed a mean increase in the verification time of 4.24% and a median increase of 3.34%. As for the `StatsCollector`, the verification time sometimes decreased for the caching overhead test, which is also not explainable by our changes, since ViperServer serves as a middleware and uses the exact same Viper backend as Gobra. However, the mean and median verification time increase is higher which implies that there is some sort of overhead when using ViperServer with an empty cache. In the fully cached version, the runtime decreases by 19.5% on average. The highest decrease can be found in the `onehop`

package, where it decreased by 96.6%. This confirms, that caching can have a huge impact on verification time. Moreover, if we compare the times in figure 5.3 with the verification times in 5.1, we can see that caching worked especially well for all package with a verification time of more than a minute. For all of them the verification time decreased more than 60% when every cacheable method was cached. However, there also are packages that did not profit from caching at all. In some cases, they don't contain any or only very few Viper methods that can be cached. This explains, why for example for the `promauto` package, the fully cached time is almost equal to the cache overhead time.

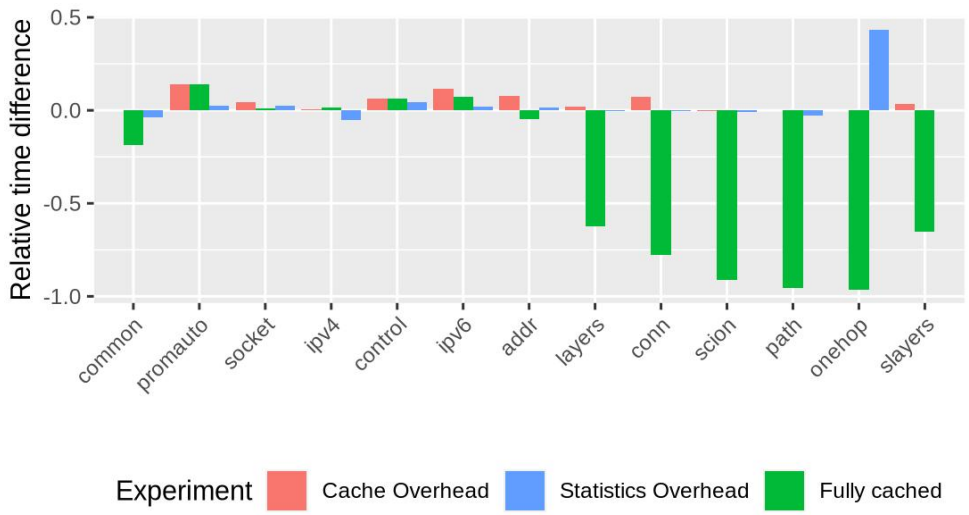

Figure 5.3: Relative time measurements for the other 13 packages compared to the version (n=20)

In general, we can conclude that the `StatsCollector` causes a minimal overhead in verification time. This is reasonable, since it also provides the means to analyze verification runs in detail. While ViperServer in general causes a larger verification time overhead, it is still justified to be used when verifying a project multiple times, because of the huge verification time decrease that comes from successfully cached packages. As stated in the beginning, we did not include the `router` package in the general overhead test. Nonetheless, we tested it with and without caching to get an idea about the possible time savings that could be achieved. Note that we only verified it three times with and without caching, therefore this data should be viewed with caution. The verification time without caching were 5.28 hours, 4 hours and 3.48 hours. It can be seen that there is a big variance in that data. However, when running with a full cache, the verification time decreased to around 10 minutes on all three runs. Therefore, in particular on this package, caching could have a huge potential.

5.2 Caching Verification Results on Evolving Codebases

We want to evaluate how well the caching mechanism works in practice and how much time is saved by using it on an actively developed project, subject to code changes. In this evaluation we simulate a CI workflow by verifying VerifiedSCION on a history of 13 commits once with caching and once without caching enabled. We started with the commit hash `8ce8ed09` and took the next 13 commits that made changes to the code up to commit 804e10c. Therefore, we ignored all commits in between that only made changes to the CI file or the README, since they would bring the same result as the experiment with a full cache. The verification with caching enabled starts with a clean cache and shares the resulting cache between verification runs. Statistics are enabled for both parts, so we can get information about the workflow and methods that timeout. Because of time constraints we had to limit the sample size to three runs, since verification for five out of the 13 commits can take up to six hours. Therefore, a single run with all commits may already take more than 48 hours. The mean verification time in seconds per commit may be seen in figure 5.4

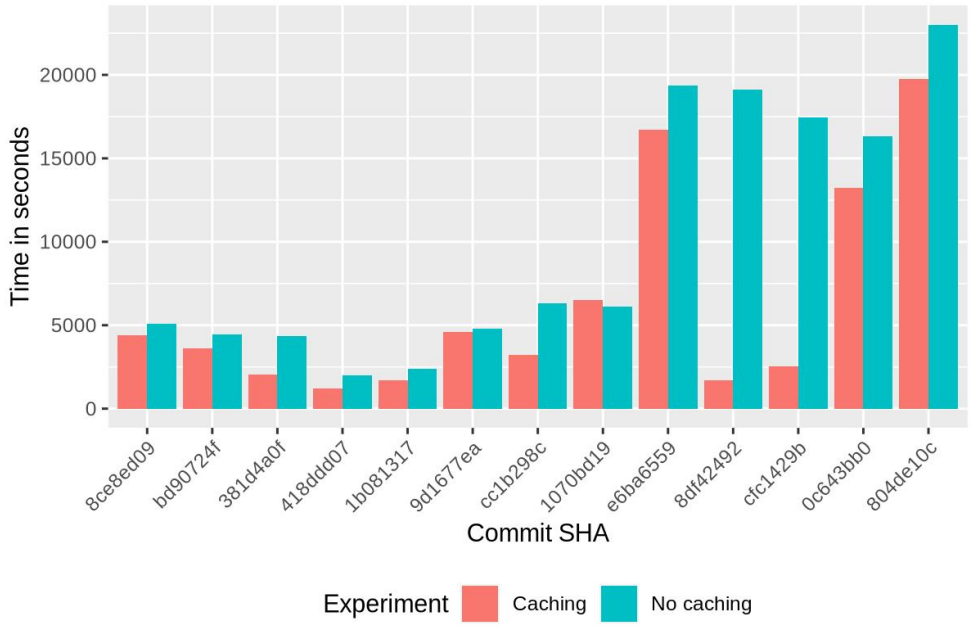

Figure 5.4: Mean verification time on the commit history (n=3)

There are a few anomalies in this data that should be clarified. Firstly, for the first commit the verification times should be equal, or the caching experiment should have a higher time, since we started caching with an empty cache. Again this probably comes from some randomness in the verification time paired with a very small sample size. Secondly, one may notice the big verification time increase

that comes with the commit `e6ba6559`. There, changes to the `router` package were introduced that increased its verification time to multiple hours. For commits `8df42492` and `cfc1429b`, the `router` package was successfully cached, which resulted in a drastic decrease for the verification time in the cache experiment. Lastly, the commits `0c643bb0` and `804de10c` both are squashed commits from pull requests that extended the `router` package. When squashing a pull request into a single commit, all commits in the pull request are combined into a single commit. These results in a large amount of changes for these commits. Thus, caching for the `router` package in these commits failed, which leads to the high verification times even with caching enabled. With this data, we can conclude that caching will influence the verification time in the CI of this project very positive, since once the `router` package is successfully cached, the verification time will decrease drastically. Additionally, we can conclude that our approach is especially effective for small changes to a codebase.

Chapter 6

Conclusion

The main goal of this thesis was to provide a straightforward way to integrate Gobra in the CI process of Go projects. This was in a first step achieved by creating a GitHub Action that runs with a Docker image containing Gobra. However, by just adding this action, the main issues, identified in chapter 1 are not addressed. To tackle performance issues, we added caching support to Gobra by making use of existing caching solutions in ViperServer and extending them with the ability to be serialized and deserialized. This way we allow caching in a CI environment, since there, the cache has to be shared in form of an artifact between subsequent runs, since processes are shutdown after each run. We then addressed Gobra's issue of not being able to support certain Go language features, by adding the trusted flag. This flag lets a user trust a function's specification and instructs Gobra to not parse the function's body. Therefore, Gobra won't encounter the unsupported feature. While not solving the underlying issue of non-supported features, it still enables developers to integrate Gobra into their workflow without having to modify the source code to remove the occurrences of these features. Lastly, we provided a modular extension to Gobra that lets it collect statistics of verification runs. These are logged in the form of a short report and exported to a file for more detailed analyses. This way, developers receive feedback to how far a project is verified and get the ability to analyse problems like time-outs that might occur during verification.

All these changes cause small overhead in verification time which we believe to be offset by the gains from caching. The caching strategy has proven to be especially effective for small codebase changes.

Future Work

This thesis presents multiple opportunities for follow-up work. Firstly, caching still has some issues, described in section 4.2.3. These should be addressed to improve its performance. Additionally, it could be explored, what kind of syntactic

differences could be allowed for methods to still consider them as unchanged for the purpose of caching.

Secondly, with the `StatsCollector`, we laid the foundation for the collection of arbitrary statistics about verification runs. Additionally, the exported statistics could be used in the future for automatic analyses as part of a CI process.

Lastly, as developers get more experienced in using the Gobra CI action, GitHub composite actions could be created. For example to provide different ways of storing the cache.

Bibliography

[1] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular specification and verification of go programs," in *Computer Aided Verification (CAV)*, A. Silva and K. R. M. Leino, Eds. Springer International Publishing, 2021, pp. 367–379.

[2] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62.

[3] N. Chong, B. Cook, J. Eidelman, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle, "Code-level model checking in the software development workflow at amazon web services," *Software: Practice and Experience*, vol. 51, no. 4, pp. 772–797, 2021.

[4] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[5] 2022. [Online]. Available: https://neo4j.com/product/neo4j-graph-database/

[6] 2022. [Online]. Available: https://docs.docker.com/develop/develop-images/multistage-build/

[7] 2022. [Online]. Available: https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-container-registry

[8] 2022. [Online]. Available: https://docs.github.com/en/actions/learn-github-actions/usage-limits-billing-and-administration#usage-limits

[9] 2022. [Online]. Available: https://github.community/t/feature-request-build-args-support-in-docker-container-actions/16846

[10] 2022. [Online]. Available: https://github.com/marketplace/actions/download-workflow-artifact

[11] 2022. [Online]. Available: https://neo4j.com/developer/neo4j-apoc/

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Program Verification in Continuous Integration Workflows

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Gasser	Johannes

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

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

Place, date	**Signature(s)**
Zürich, 17.03.2022	J. Gasser

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.