GitSlice: Slicing the Repository That Feeds Us

Dylan Wilson <dwilson402@qub.ac.uk>
Queen's University Belfast

***Abstract*—Professional software development teams now almost universally use Git (a distributed version control system) to manage changes to their code. Git repositories contain metadata relating to the development of the project and show how a project has changed over time. Analysis of this metadata to gain insight into the software development process is performed in the field of mining software repositories (MSR). Static code analysis (SCA) tools perform analysis on source code to detect issues during development (they are static because they analyse the code without running it) such as untested or vulnerable code. SCA only analyses one version of the code, however. The ability to run code analysis tools across many versions of a project could therefore provide developers with useful information of how their code changes over time. To achieve this, a tool, GitSlice will be built to iterate through a given set of Git commits to run a code analysis tool on each version of the software and provide the developer with information about how metrics of their code change over time. Iterating through commits in a linear fashion is slow, so to reduce the time taken to iterate through the commits we will parallelize this process.**

***Index Terms*—Version control systems, Mining software repositories, High-performance computing.**

I. Introduction

Today, virtually all professional and open-source development teams use version control systems (VCSs) to manage changes in their projects. StackOverflow reports over 96% of professional developers use Git [1]. Git is a distributed VCS developed originally by Linus Torvalds and other contributors to the Linux kernel. Version control systems enable developers to manage changes to code in their software projects. A set of changes in Git is known as a commit [2]. Each commit points to one or more previous commits to form a tree (except for the first commit known as the root commit). Multiple commits can have the same parent, which allows for a branching model and commits which have multiple parents merge two or more branches together. Figure 1 shows the Git branching model visually.

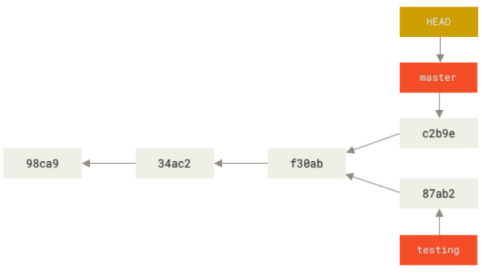

Fig. 1. Git branching model, two branches exist one called master and one called testing [2].

Commits contain more information than just changes to file contents - they also include information about who authored the commit and who last applied it (for example, when a commit is imported from a patch file or when the commit is part of a rebase)—known as the author and committer respectively. The field of mining software repositories (MSR) attempts to analyze the contents of software repositories to obtain useful information about a software project [3].

Code analysis tools are used to understand the quality of software code such as test coverage and security vulnerabilities. Tools which do not run the code but instead analyse the code by reading it are called static code analysis. One common type of static code analysis is static application security testing (SAST). A SAST tool such a Snyk Code or SonarQube Security Analysis will analyze the code to detect common security vulnerabilities such as the use of known weak hashing or encryption algorithms, or known vulnerable dependency versions [4]. One common limitation of these tools, however, is that they only perform analysis on one version of the software. It is possible then that an older version of the project used a dependency which has since had a vulnerability discovered—if the vulnerability is not present in a later version of the dependency (a version which the current version of the software project uses) then the SAST tool will not identify this vulnerability. Similarly, as these tools continually improve and can detect more types of vulnerability it is possible that an older version of the project contains a vulnerability which was not initially discovered but running the tool again would result in it detecting the vulnerability. This older version of the software could still be deployed, however and so identifying vulnerabilities in older versions of software is still important.

Another example of static code analysis is code coverage of software testing. This type of tool will run tests written for the software and calculate the percentage of the program tested and which lines are untested. While it could be useful for software development teams to see the trend in code coverage over time, code coverage tools typically only perform code coverage on the current version of the code.

A tool which performs analysis on multiple versions of the project could show a time-series graph to show how metrics change over time. One example of this a graph which shows how the percentage of code covered by tests changes over time.

Iterating through each version of the project sequentially and running analysis is slow (the Linux repository for example has over 1.3m commits as of Dec 2022 and if each commit only took 1 second to be checked out and built it would still take 15 days to go through every commit [5]). Each version of the program does not depend on other versions previously having been built so parallelization can be used to significantly speed up this process. Each process could use an isolated

environment to avoid overwriting output from other processes.

II. Literature Review

Mining software repositories: The mining software repositories (MSR) field analyses software repositories to find actionable information about software projects [6]. In MSR studies, researchers extract data from a repository to produce evidence to answer a research question [3]. While software repositories are typically used as historical records of the software development process, MSR attempts to use the information contained within them to guide future decisions.

Many tools exist to analyse software evolution but these primarily focus on metadata. For example, one such tool is Boa [7] which uses a domain-specific language (DSL) to describe how to mine a selection of software repositories from SourceForge and GitHub. This enables researchers to write DSL for questions like "How many files are changed on average per commit in projects which use Java?". The tool will then iterate through each commit in every Java project on SourceForge and GitHub to produce the answer. Boa, however, does not allow for the running of a specific tool on each iteration but rather simply extracting metadata from the commits of the repository.

Similarly to Boa, [8] introduces CLOSER which is a DSL for metadata across multiple types of VCS'. This allows for the conversion of a software repository from one type of VCS to another. CLOSER primarily focuses on the metadata of a repository (as opposed to each version of code stored within in it) but has provided a starting point for research into this area.

Another is example is described in [9] which introduces LibVCS4j, a Java library which enables the mining of software repositories. LibVCS4j supports multiple types of repository and abstracts the specific VCS away at an API level to enable the mining of data from multiple types of repositories. Again, however, this library primarily focuses on enabling the mining of metadata from the repository as opposed to performing analysis on the code itself.

[10] describes CVSgrab is a tool for visualizing metadata of a CVS repository (an older VCS than Git). This allows for visualization of different metadata such as the types of files and authors of lines of code. The tool produces a graph which shows how this metadata changes over time. CVSgrab however, is built for CVS and doesn't allow the running of any tool on each version of the software but merely one type of visualization of the data contained within the repository.

[11] describes MJgit, a tool which performs code analysis on changes to code in a project by understanding how these changes affect the code. This allows for better textual search of the source code by ignoring cases where methods were simply moved but their functionality did not change (this would be registered in Git as line deletions then line additions, but wouldn't be included in the search results for MJgit) allowing for more accurate searching of the repository.

Source code analytics: [12] compared the features of the three most commonly presented tools for static code analysis in research: Cppcheck, FindBugs and SonarQube. The finding was of all the features compared, SonarQube provided the greatest feature set. SonarQube also supports more languages than the other two where Cppcheck only supports C/C++ and FindBugs only supports Java. Therefore, SonarQube could be used to perform static code analysis on many types of project, and therefore a good potential candidate for a tool to validate any approach.

[13] investigates the use of static code analysis in the CI/CD pipelines of 20 open-source Java projects. A majority of projects used tool called CheckStyle which checks adherence to a set of rules on code style. CheckStyle is another tool which could be used to validate the approach by understanding how rule violations vary over time. Further, the use of GitSlice in the CI/CD pipeline would enable developers to continually run their static code analysis tools on previous versions of the project. For example, running GitSlice upon push to a mainline or release branch would ensure older versions are continually checked.

[14] analysed the benefits and limitations of Checkmarx, a common static application security testing (SAST). This is another good example of a tool which could be used to validate the approach by creating known vulnerable code between specific commits and testing to see if the tool is able to use Checkmarx to find the vulnerability.

Running at scale: [15] describes torcpy, a load-balancing Python library which manages the execution of multiple asynchronous tasks and supports OpenMPI which is an open-source MPI implementation that is widely used on computing clusters including Perlmutter, the 8th most powerful supercomputer according to the Nov 2022 TOP500 rankings [16]. This enables multiple versions of the program to run across nodes in a cluster, balancing workload to ensure optimal use of resources.

Some existing work has been done in relation to accessing Git repositories in high-performance computing environments such as RepoFS [17], a tool for accessing multiple versions of the repository. RepoFS exposes the commits of the repository as a virtual filesystem, ordered in a tree structure by metadata from the commit by both commit ID and by date of the commit. Use of RepoFS would allow a highly parallelised analysis to be performed on a shared filesystem against a single copy of a repository rather than making multiple copies each at a specific point.

Running multiple versions of the same program can lead to issues, for example SonarQube will fail to run if another instance is already scanning the same project [18]. Containers, such as Docker, enable the isolation of a process from the rest of the operating system and ensures that the process will run predictably across different platforms [19]. Running the code analysis tool inside a container also means it will be isolated (including its filesystem) from the other processes on the system allowing for multiple versions of the same tool to run simultaneously without interacting with each other. An analysis of Docker in high-performance environments shows that this also provides performance similar to that of running the software natively (i.e. directly on the operating system without any visualisation) and much greater than that of running it inside virtual machines [20].

APPENDIX
CURRENT WORK

To date an application written in Python has been partially developed [21]. The application uses the GitPython package to interact with the Git repository. The program reads from a configuration file (written in YAML) at the location passed as an argument. This file contains options such as the repository to be analysed, the Docker image containing the analysis tools to be used, and the command to run to start the container. It will then copy a local Git repository (later, it will be able to clone a remote repository) and copy it into a working directory. The application gets each commit in the repository and will checkout that version in an incremental fashion. For each commit, the Docker image specified by the configuration file will be pulled and a container based on this image run. The project (as it existed at the commit to be analysed) will be mounted on the container and the command provided in the configuration file will be run to begin the analysis. When the container exits, it's output (both stdin and stderr) is output to the screen. This represents a basic, working version of what we aim to develop.

There are some limitations with this current version – the most obvious of which is that every commit in the repository is checked-out and run, later the configuration file will be read to identify which set of commits should be checked-out for example, every 5th commit from the year 2019. Further, the output of the container is simply printed to the screen and not useful work is done to interpret this. As previously mentioned, the ability to clone the repository directly from a remote will also be added. Later versions of the application could save the output of each run to a directory to allow further analysis to be performed on the output, some additional features to interpret the output and display graphs could also be added (for example, by grepping for a particular value in the output). The application currently runs each commit in a sequential fashion so changes will be made to parallelize the code so that multiple versions of the repository can be checked out and run in parallel. The packages mentioned in the previous section will be used to achieve this. To use the RepoFS tool on Kelvin 2 it is necessary to build the libgit2 headers from source (we cannot install packages here) – this will be handled by the program upon starting up on each node. It is possible to use run containers on Kelvin 2 using the Singularity module, further work will be done into how to extract a container as an .iso file and run this in singularity programmatically.

REFERENCES

[1] StackOverflow, "StackOverflow Annual Developer Survey 2022," Jun 2022. [Online]. Available: https://survey.stackoverflow.co/2022/#section-version-control-version-control-systems

[2] S. Chacon and B. Straub, *Pro Git (Second Edition)*. Apress, 2014.

[3] A. E. Hassan, "The road ahead for mining software repositories," in *2008 Frontiers of Software Maintenance*, 2008, pp. 48–57.

[4] Dave Wichers, "Source Code Analysis Tools," Dec 2022. [Online]. Available: https://owasp.org/www-community/Source_Code_Analysis_Tools

[5] L. Torvalds, "Github - torvalds/linux: Linux kernel source tree," Dec 2022. [Online]. Available: https://github.com/torvalds/linux

[6] International Conference on Mining Software Repositories, Aug 2022. [Online]. Available: http://www.msrconf.org/

[7] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 422–431.

[8] J. Garrity and D. Cutting, "Closer," 2021 Sep. [Online]. Available: https://closer.eeecs.qub.ac.uk/

[9] M. Steinbeck, "Mining version control systems and issue trackers with libvcs4j," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 647–651.

[10] M. Sasaki, S. Matsumoto, and S. Kusumoto, "Integrating source code search into git client for effective retrieving of change history," in *2018 IEEE Workshop on Mining and Analyzing Interaction Histories (MAINT)*, 2018, pp. 1–5.

[11] ——, "Integrating source code search into git client for effective retrieving of change history," in *2018 IEEE Workshop on Mining and Analyzing Interaction Histories (MAINT)*, 2018, pp. 1–5.

[12] D. Nikolić, D. Stefanović, D. Dakić, S. Sladojević, and S. Ristić, "Analysis of the tools for static code analysis," in *2021 20th International Symposium INFOTEH-JAHORINA (INFOTEH)*, 2021, pp. 1–6.

[13] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta, "How open source projects use static code analysis tools in continuous integration pipelines," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 334–344.

[14] J. Li, "Vulnerabilities mapping based on owasp-sans: A survey for static application security testing (sast)," Jul 2020. [Online]. Available: https://doi.org/10.48550/arXiv.2004.03216

[15] P. Hadjidoukas, A. Bartezzaghi, F. Scheidegger, R. Istrate, C. Bekas, and A. Malossi, "torcpy: Supporting task parallelism in python," *SoftwareX*, vol. 12, p. 100517, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711020300091

[16] TOP500, "Perlmutter - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10," Nov 2022. [Online]. Available: https://www.top500.org/system/179972/

[17] V. Salis and D. Spinellis, "Repofs: File system view of git repositories," *SoftwareX*, vol. 9, pp. 288–292, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711018300712

[18] S. JIRA, "Sonar doesn't support parallel runs/analysis on the same project," Sep 2011. [Online]. Available: https://sonarsource.atlassian.net/browse/SONAR-2761

[19] R. Gandhi and P. Szmrecsanyi, "The Benefits of Containerization and What It Means for You," Feb 2019. [Online]. Available: https://www.ibm.com/cloud/blog/the-benefits-of-containerization-and-what-it-means-for-you

[20] J. Higgins, V. Holmes, and C. Venters, "Orchestrating docker containers in the hpc environment," in *High Performance Computing*, J. M. Kunkel and T. Ludwig, Eds. Cham: Springer International Publishing, 2015, pp. 506–513.

[21] D. Wilson, "GitSlice - GitLab Repository," Dec 2022. [Online]. Available: https://gitlab.eeecs.qub.ac.uk/40234266/csc4006-project