AutoMetric: Towards Measuring Open-Source Software Quality Metrics Automatically

Taejun Lee
tjlee@korea.ac.kr
Korea University

Heewon Park
paki9977@korea.ac.kr
Korea University

Heejo Lee
heejo@korea.ac.kr
Korea University

ABSTRACT

In modern software development, open-source software (OSS) plays a crucial role. Although some methods exist to verify the safety of OSS, the current automation technologies fall short. To address this problem, we propose AutoMetric, an automatic technique for measuring security metrics for OSS in repository level. Using AutoMetric which only collects repository addresses of the projects, it is possible to inspect many projects simultaneously regardless of its size and scope. AutoMetric contains five metrics: Mean Time to Update (MU), Mean Time to Commit (MC), Number of Contributors (NC), Inactive Period (IP), and Branch Protection (BP). These metrics can be calculated quickly even if the source code changes. By comparing metrics in AutoMetric with 2,675 reported vulnerabilities in GitHub Advisory Database (GAD), the result shows that the more frequent updates and commits and the shorter the inactivity period, the more vulnerabilities were found.

KEYWORDS

Open source · Software test automation · Software metrics

CCS CONCEPTS

• **Software and its engineering** → **Software verification and validation**;

1 INTRODUCTION

In recent years, there has been a significant rise in the number of supply chain attacks, with malicious software supply chain attacks showing an average annual growth rate of 742% [1] since 2019. A supply chain attack is a type of cyber attack that exploits the structure of the software supply chain, effectively attacking the software's dependency structure. These attacks can cause significant damage by spreading through the entire supply chain, and are difficult to detect due to being behind the normal chain of trust.

Nowadays, a great deal of open-source software (OSS) are necessarily used as component to develop programs. According to a report by the Linux Foundation, it has been estimated that OSS constitutes 70-90% of any given piece of modern software solutions [2]. The complexity of the software dependency structure makes it easier to spread vulnerabilities to other packages and attack software supply chain. In such a development environment that utilizes OSS as components, selecting packages and managing vulnerabilities in components are time-consuming and costly. For example, if a vulnerability occurs in one of the components of the software, the vulnerability can be patched by updating the component to the latest version. To do this, it is necessary to find and patch a vulnerability in the components. However, the number of components has increased that it has become difficult to manage all components manually.

In this situation, we need efficient software security measurement. The criteria that represents how secure and safe the software is called a software security metric (SSM). Every security-related measurements can be SSM, and each SSMs has different properties such as a difficulty of measurement and impact.

The software supply chain encompasses four key phases: open-source communities, repositories, package managers, and end users. As a result of the collaborative nature of OSS, all users are simultaneously involved in both production and consumption. End users may also contribute to the development process by participating in another open-source community. It would be better to measure all phases to get more accurate result. However, measuring metrics takes time and we have to choose which level is the most efficient. Our objective is to develop a method for measuring multiple software supply chain metrics at once, which is fast and easy to use.

It is important that codes in open-source community could be gateways where cyber attacks occur. Nevertheless, because additional vulnerability analysis technology is required, it is unsuitable to be our SSM. It takes a long time to analyze the entire source code, and if the code changes, it needs to be operated every time it alters. In case of packages, these are language dependent and users are scattered according to the language. In addition, because AutoMetric aims at language-independent analysis, source code analysis which is language-dependent could not be included.

Meanwhile, SSMs in the form of repository are able to measure automatically since repositories provide API. We can check the history of changes in the source code and how the developers manage the project from repositories. In addition, currently, the number of GitHub users is more than 94 million, [3] and the number of GitLab users is over 30 million. [4] Therefore, we decided to focus on repositories.

Limitation of existing techniques. There are three prominent existing technologies that can be used for software security metrics: 1) Software development best practice(SDBP) and 2) Scorecard.

1) Software development best practice (SDBP): SDBP refers to developing software in a proven manner, with various principles

AST'23, May 15-16, 2023, Melbourne, Australia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

presented in text format, depending on the companies and institutions. Both security-related companies and government agencies have conducted studies on SDBP. For instance, the National Institute of Standards and Technology (NIST) in the United States has presented software guidelines such as the Cyber security Framework [5] and SSDF [6] to be followed during the software development process. SDBP outlines the principles that need to be followed for developing safe software, and it can contribute to our research in this respect. However, the disadvantage is that it is challenging to comprehend the guidelines quickly since most of them are written in natural language. Efficient metrics should be represented numerically.

2) Scorecard: Scorecard [7] is an automatic mechanism developed by the Open Source Security Foundation, consisting of 18 software security metrics (SSM) and is composed of 10 points. It distinguishes whether OSS dependencies are safe or not. However, there are limitations in measurement. For instance, 'Contributors,' one of the SSMs, is based on the number of organizations involved. The number of organizations is determined by the organization name and email, but it is challenging to differentiate them if they are spelled differently or if personal emails are used instead of organizational emails.

Our approach. In this paper, we present AutoMetric, a technology that automatically performs measurements on selected metrics of the repository phase. The AutoMetric contains five SSMs: Mean Time to Update (MU), Mean Time to Commit (MC), Number of Contributors (NC), Inactive Period (IP) and Branch Protection (BP).

Our objective is to determine whether security can be measured effectively through the five selected SSMs. To achieve this, we utilized the GitHub Advisory Database [8] (GAD), which provides access to vulnerability information from the National Vulnerability Database [9] in real time as well as vulnerabilities in ecosystems such as pip, npm, and Maven. GAD also provides repository addresses for the listed projects. Thus, by applying AutoMetric to the vulnerabilities stored in GAD, we expect to obtain similar results to those obtained by applying it to overall vulnerabilities. We compared the number of vulnerabilities and the values measured for each metric using GAD and AutoMetric. The GAD contained 2,675 repositories selected based on the CVEs that were reviewed by GitHub and uploaded to the GAD repository in json file.

Evaluation. All five selected metrics of AutoMetric were automated using the repository's API. SSM automatically generates a json file of selected repository within seconds. We compared each result to the number of CVE in GAD. We also do the case study which is an interesting case found in the automatic measure process using AutoMetric and it was written in Section 4.3. For example, we analyze why the maximum and minimum values in each metric are measured in the repository.

Contribution. This paper makes the following contributions:

We have demonstrated the feasibility of automatically measuring security metrics for large-scale software using AutoMetric. With only the Git repository address, AutoMetric can measure five SSMs and perform automatic measurement on multiple repositories simultaneously.

AutoMetric can contribute to the standardization of SSMs by providing a consistent and comparable approach for measuring security. Currently, SSMs are difficult to compare and often lack consistency. Developing and applying standardized software security metrics can provide consistent and comparable security results.

We compared AutoMetric with the already known vulnerabilities identified in GAD. This allowed us to cover the source code level where vulnerabilities were found and we found a significant correlation. MU, MC, and IP showed a clear increase in the number of vulnerabilities as the metrics increased. Especially, the three timing-related metrics showed a strong correlation with the known vulnerabilities in GAD. While some projects may not have sufficient information to compute MU, we found that these complementary metrics, such as MC and IP, can still be useful in measuring the security of OSS.

2 BACKGROUND

In this section, the structure of the software supply chain is introduced, and software security measurement metrics are described.

2.1 Structure of the Software Supply Chain

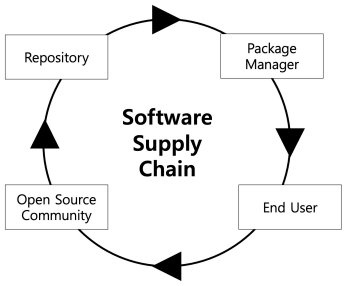

Figure 1: Structure of the Software Supply Chain

Open-source communities. The software supply chain of the OSS consists of four phases: open-source community, repositories, package managers, and end users. [10] Software developers are in the process of creating source code for software. The developer uploads the source code of the project to the repository.

Repositories. Repositories are a form of data structure where source codes are stored such as GitHub and GitLab. The repository is driven based on a version control system (VCS) such as Git, and all software changes are recorded. Repositories have fork functions that allow you to replicate existing projects to develop modified software, branch functions that allow you to develop software in multiple branches, and contribute to specific projects.

Package managers. Package Manager is a service that allows end users to download software distributed in the form of packages. Javascript's package manager npm, Python's package manager PyPI, Java's package manager Maven, Ruby's package manager RubyGems are examples of package managers. Apple's AppStore and Google's PlayStore can also be seen as package managers.

End users. The end user is the last phase in the software supply chain and refers to a user using the developed software. However, If the end user downloads the package from the package manager

and develops it, the end user may be both the developer and the end user. The software supply chain is composed of such multiple chains.

The more software libraries are used, the more complex the structure of the software supply chain becomes. This can cause the increase of software supply chain attacks. To prevent this, software libraries should be reduced to a minimum, and we should choose a secure library.

2.2 Motivation.

Among the numerous metrics, it is necessary to derive an optimized set that can perform efficient security verification. To this end, it is necessary to collect SSM and create an initial set by considering the difficulty of measuring metrics and the possibility of automatic measurement. After deriving the initial set of metrics, an automatic measurement technology is developed and applied to a project.

Challenge 1: Measure SSMs automatically. There are numerous SDBPs, and many of them involve metrics that cannot be quantitatively measured. For instance, it is difficult to quantify best practices such as "Train all users of software, based on their roles and responses." [11] Given the multitude of development best practices, it can be considered a primary challenge to gather measurable metrics.

As many SSMs cannot be automatically measured, the process of developing automatic measurement technology after evaluating the possibility of automation is one of the significant challenges. For instance, Mean Time to Update (MU), which indicates the average time taken for a software to update when a new version of its component is released, is a crucial metric to measure. However, measuring MU requires two essential pieces of information, i.e., the release date of each version of the software component and the release date of the new version of the software component. Unfortunately, these pieces of information are stored in various ways by different projects, and accurate information collection may not be feasible. In cases where systematically managed SBOM is available, automatic measurement is possible; otherwise, the metric can only be measured manually.

Challenge 2: Verify the effectiveness of the security metrics. Even if SSMs are created from SDBP, the effectiveness of the metrics is not clear. The measurement of the metrics is meaningless if the metrics are not related to security. Therefore, verify the effectiveness of the security metrics through evaluation is one of the main challenges.

2.3 A Motivating Example

Table 1: the Number of packages in popular Linux distributions counted in 2022

Linux Distribution	Number of pre-compiled packages
Debian	153,621
Fedora	66,075
Ubuntu	100,530

In the case of open-source operating system (OS) such as Linux, many OSS are included as a software package. Checking the security of an open OS is to check the security of a package included in the open-source OS. Table 1 shows the approximate number of pre-compiled packages in each Linux distributions. Ubuntu, one of the most famous distributions of Linux, contains more than 100k packages in the Ubuntu package manager. [12] With a lot of packages, it is impossible to manually check the security of packages before the next version is released.

3 METHODOLOGY/DESIGN

3.1 Overview

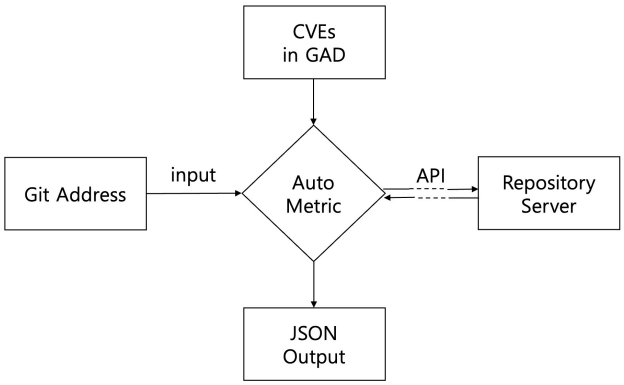

Figure 2: Overview of the AutoMetric

AutoMetric automatically measures measurable security metrics through the GitHub API and GitLab API. AutoMetric is divided into three phases: 1) Address analysis, 2) Metric measurement, and 3) Result output. We chose Python, an interpreted language that can be easily used in multiple operating systems, as the development language to support multiple operating systems. In order to measure metrics of repository phase, values required for measurement must be returned using the repository API. Figure 2 shows the overview of the AutoMetric.

3.2 Address Analysis

AutoMetric can perform metric measurements on projects stored in GitHub and GitLab-based repositories. In the case of GitHub, the address is organized in the form of [github.com/GROUP/PROJECT.git.] We can parse the address by '/' to extract the project name and group name, and then use the GitHub API. In the case of GitLab, unlike GitHub, it supports the subgroup function. Therefore, the address is organized in the form of [gitlab.com/GROUP/SUBGROUP/PROJECT.git]. We can parse the address based on' /' to extract the project name, group, and subgroup names, through which you can use the GitLab API. In the case of GitLab, it is possible to create a GitLab-based third-party repository separate from the existing GitLab server. For example, Debian Salsa, which stores packages for the Debian operating system, is a third-party repository based on GitLab. The domain of Salsa is [salsa.debian.org]. The domain is different from the original GitLab, but we can still use the GitLab API.

3.3 Measuring Metrics

For measuring the metrics of the repository phase, we need the information of the repository of the project. In this paper, automatic

Table 2: Details of each automatic measurement

	Metric	Computational method	Description
M1	MU	(last_release - first_release) / count	the average update time of a repository release.
M2	MC	(last_commit - first_commit) / count	the average update time of a repository commit.
M3	IP	now - last_commit	the inactive period, from last commit date up until now
M4	NC	total (contributors)	the number of contributors participating in the repository.
M5	BP	IsProtected	whether branch protection are applied or not.

measurements are made for GitHub and GitLab, so GitHub API and GitLab API are utilized. To use the APIs, we used python library for each APIs. We used PyGithub [13] for GitHub API, and the python-gitlab [14] for GitLab API.

In Git-based repositories, we can find branches. Branch is the concept for collaborating with others and managing multiple direction of the development. Among multiple branches, only one branch is called the main branch. We are going to measure the metrics within the main branch.

Our goal is to create metrics that are more efficient than the existing ones. The explanation for the five SSMs are as follows.

MU: mean time to update. Mean time to update (MU) is a metric that measures the average time of a repository release update. There is an API that returns all the releases of the repository at the releases API of GitHub and GitLab. In the returned release information, the date of the release is contained through the value 'created_at', and the version name is returned through the value 'tag_name'. If the period between the first release and the last release is calculated and averaged by the total number of releases, the MU can be measured.

MC: mean time to commit. Mean time to commit (MC) is a metric of how often commits have been made in the repository. Commit is the smallest unit of code update in the repository. Therefore, the frequent commits indicates that the repository is very actively updated.

The GitHub Commits API allows you to retrieve the list of commits in the repository, and provides the results in multiple pages. Commits API has parameters 'perPage' and 'page', which specify how many commits are displayed on each page. The 'page' is a parameter that specifies how many pages are displayed. If the 'perPage' value is set to 1, only one commit per page is fetched, so the total number of pages is equal to the total number of commits. The GitLab Commits API can retrieve the list of commits in the repository. At this time, the parameter 'all' must be specified as yes to receive all commits. If you count the number of returned commit lists, you can find the total number of commits. The MC can be obtained by dividing the total number of commits by the time it has passed from the first commit.

IP: inactive period. Inactive period (IP) is a metric measures the past time from the last commit. The higher the value, the longer the repository is not updated. In the case of a project that has been completed and does not require further development, security of the repository may not be degraded even if the IP is high, so we have to consider carefully if it is required or not. GitHub and GitLab's Branches API allows us to return information about a particular branch, and you can check the return value for the date of the last commit made by that branch. Calculating the time elapsed from that date can measure the IP.

NC: the number of contributors. The number of contributors (NC) is a metric of the number of contributors participating in the repository. GitHub and GitLab's repositories API provides APIs that return a list of contributor. Number of contributors can be measured by counting the number of tuples in the contributor list.

BP: branch protection. Branch protection (BP) is a metric of whether branch protection settings provided by GitHub or GitLab are applied. There are various branch protection rules, and the collaborator controls behavior, such as deleting branches or push the changes by force. We can check the 'protected' value of a specific branch through the Branch API of GitHub and GitLab. The value 'protected' is in Boolean format, and if it is true, the protection setting is applied. However, if it is False, the protection setting is not applied.

3.4 Result Output

The result of measuring the metrics is output as a file in JSON format. The project name and measured score for each metric are recorded together. We used Python's json library to read and write json documents through Python.

4 EVALUATION

This section evaluates AutoMetric. Subsection 4.1 introduces experimental setup, including the experimental environment and target dataset. Subsection 4.2 shows the result of the AutoMetric, and investigates the correlation between the security and the security metrics. Subsection 4.3 is the case study of interesting cases found during evaluation.

4.1 Experimental Setup.

Experiment environment. AutoMetric was developed based on Python 3.10.8. The Python packages used is as follows: PyGithub 1.56, python-gitlab 3.10.0, requests 2.28.1. We used i5-12400 CPU, 32GB Ram, and native Windows 11 desktop to run AutoMetric.

Target dataset. We used the GitHub Advisory Database (GAD) [8] as a target dataset. The GAD is vulnerability database inclusive of CVEs and GitHub originated security advisories from the world of open source software. The GAD contains over 10k GitHub reviewed advisories, and over 170k unreviewed advisories. Our target is only the reviewed advisories.

Each Advisory in the GAD is security vulnerability. It includes the description, severity, affected package and references. We downloaded the full database and parsed it to figure out the number of vulnerabilities in each repository of the package. As a result, 2,675 packages were collected.

Define the security of the software. It is imperative to establish a clear definition for software security. Vulnerable software is defined as software that contains vulnerabilities, and therefore, the more vulnerabilities that exist within the software, the less secure it is considered to be. In essence, we can measure the security of software by quantifying the number of vulnerabilities present.

4.2 Results

The results of performing automatic measurements on 2,675 repositories are as follows. All values were rounded to the third decimal place. Total means the number of software in which the metric is detected. Maximum refers to the maximum value among the detected values of the metric, and the minimum refers to the minimum value among the detected values of the corresponding metric. Mean refers the average of all detected values. Exceptionally, maximum and minimum of the branch protection means number of true and false respectively.

Details of the number of vulnerabilities. In the case of the number of vulnerabilities, a total of 2,675 packages were detected. The most number of vulnerabilities is 410, which were found in tensorflow/tensorflow. [15] However, the second most vulnerable project after tensorflow is microweber/microweber, [16] with 62 CVEs. The third is 54 and the fourth is 47 and the fifth is 45, so the difference is not severe from the second. Since the number of vulnerabilities in the tensorflow and the difference between other projects has not a big difference, the tensorflow was removed during analysis.

The least number of vulnerabilities is 1, and it was found in a total of 1,861 repositories. Given that 2,619 repositories had fewer than 10 vulnerabilities, most of GAD's repositories had relatively fewer vulnerabilities. Among the repositories in which vulnerabilities were detected, an average of 2.19 vulnerabilities were detected.

Details of the MU. In the case of MU, it was detected in 1,765 packages. Since the MU measurement is based on the release of GitHub and GitLab, it means there is no release in 910 undetected packages. The maximum value of the MU was 3,481.0, which was detected in the Snorby/snorby. [17] This means that updates to the repository are being made once every 3,481 days. The minimum value of the MU was 0, which was detected in the rack/rack repository. This means that the repository is updated every days. In order to derive meaningful insights, it is necessary to focus on projects that have been actively developed, rather than those that have not been updated for more than 5 years since their last update. Therefore, we excluded the top 10% of MU values from our analysis, as they are likely to be associated with less active projects.

Correlation between CVEs in GAD and MU. Figure 3 shows the correlation between number of vulnerabilities and MU. Looking at the graph, repositories with high MU values have fewer vulnerabilities, and repositories with low MU values have high vulnerabilities.

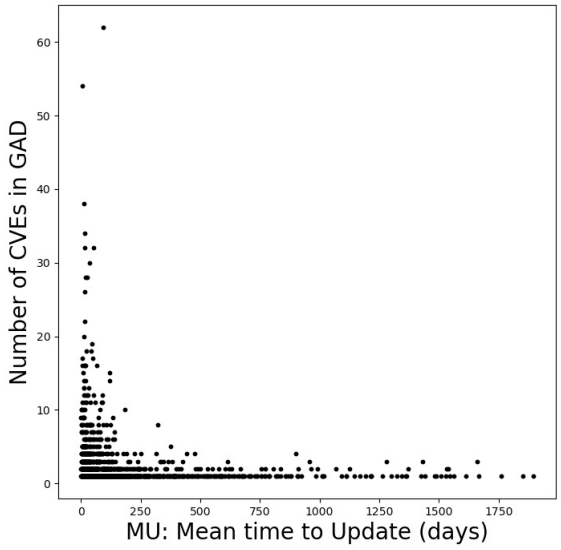

Figure 3: MU-CVE scatter plot analysis

Through this, it can be inferred that the lower the MU, the higher the number of vulnerabilities. In general, this is completely contrary to the general perception that the more frequently updates are performed, the more secure the software is. Rather, it is possible to interpret that performing updates too often reduces the security of the software, and further analysis is needed.

Details of the MC. In the case of MC, it was detected in 2,674 packages among the total 2,675 repositories. The repository whose MC value is 'n/a' is hiuminhnv/Zenario-CMS-last-version. When we checked it out, the repository itself was empty. The project was considered to have disappeared and was excluded from the analysis. The maximum value of the MC was 3,058, which was detected in nrako/psnode. [18] The repository means that commits are made once every 3,058 days. The values whose MC values are too large, unlike other values, are as follows. The result of MC was 2,259 days for andrewimm/xopen, [19] 1,290 days for jenkinski/delete-log-plugin, 1,275 days for xjamundx/gitblame, [20] 1,253 days for krl/bunch, and 1,319 days for andrewjstone/dynamo-schema. [21] The largest value used for statistics is jenkinski/random-string-parameter-plugin. This shows an 942 MC days. The above six repositories were removed because they showed a too large difference of more than 300 days compared to other values. So the total number of repositories used for MC statistics is 2,668.

Strong correlation between CVEs in GAD and MC. Figure 4 shows the correlation between number of vulnerabilities and MC. The MC shows almost the same graph as the MU. Like the MU, repositories with high MC values have fewer vulnerabilities, and repositories with low MC values have high vulnerabilities. It is

Table 3: Statistics of automatic measurement

Result	M1: MU (days)	M2: MC (days)	M3: IP (days)	M4: NC (persons)	M5: BP (cases)
Maximum	3481	3058	5070	473	—
Minimum	1	0	0	1	—
True	—	—	—	—	1187
False	—	—	—	—	1487
Mean	175.08	27.37	311.53	83.49	—
Total cases	1765	2674	2675	2675	2674

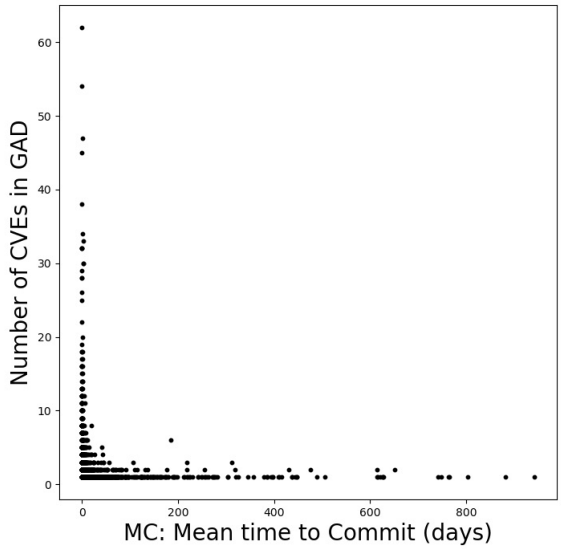

Figure 4: MC-CVE scatter plot analysis

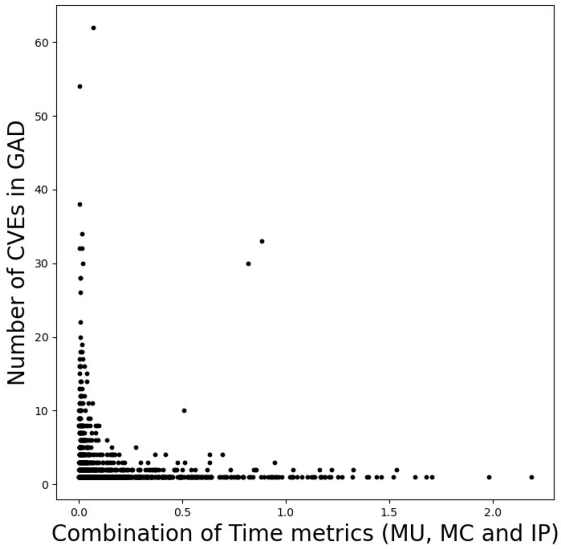

Figure 5: Time metrics-CVE scatter plot analysis

possible to interpret that performing updates too often reduces the security of the software, and further analysis is needed.

Details of IP. IP was successfully detected in all packages. The oldest package from the last commit, spejman/festivaltts4r, [22] was detected 5,070 days. The second longest repository is dynamo-schema project under andrewjstone, which has 3,958 IP days. Since the second one, there has been no significant difference, so only spejman/festivaltts4r with a large difference of more than 1,000 days was excluded from the analysis graph. The least recent package from the last commit has a detected value of 0, which means that less than a day has passed since the last commit. The average IP was 311.53 days.

Correlation between CVEs in GAD and IP. A strong correlation was found by comparing the number of vulnerabilities and IP. Repositories with high IP have fewer vulnerabilities, and repositories with low IP have high vulnerabilities. In general, this is completely contrary to the general perception that the more active the repository is managed, the more secure the software is. Rather, if the IP is too long, it may be interpreted that fewer vulnerabilities are found because it becomes a project that people do not use.

Correlation between CVEs in GAD and Time metrics (MU, MC and IP). The relationship between time metrics and GAD is more pronounced in the table. By normalizing, it changes the difference in the numerical value range of the dataset to a common scale without distorting it. It has a preprocessing process to remove values with 'n/a' in MU and MC, and those with too large values unlike other repositories.

The method of normalizing each metric is as follows:

$$0 \leq m_1(p_j) = \frac{MU(p_j)}{\max\limits_{i=1\cdots n} MU(p_i)} \leq 1$$

$$0 \leq m_2(p_j) = \frac{MC(p_j)}{\max\limits_{i=1\cdots n} MC(p_i)} \leq 1$$

$$0 \leq m_3(p_j) = \frac{IP(p_j)}{\max\limits_{i=1\cdots n} IP(p_i)} \leq 1$$

We try to find out the correlation between ϕ, which added all the normalized time metric values and vulnerabilities.

$$\Phi = sum(m1, m2, m3)$$

By adding all the normalized values, it was possible to confirm a common relationship in time metrics. The more frequent updates and commits and the shorter IP is, the more vulnerabilities were found. However, CVEs in GAD was the unclear association with NC and no association with BP.

Details of NC. NC have been detected in every repositories. Repository with the largest NC, esphome/esphome, [23] had 473 people. There were 8 repositories that had only one contributor. There were average of 83.49 contributors in the test set.

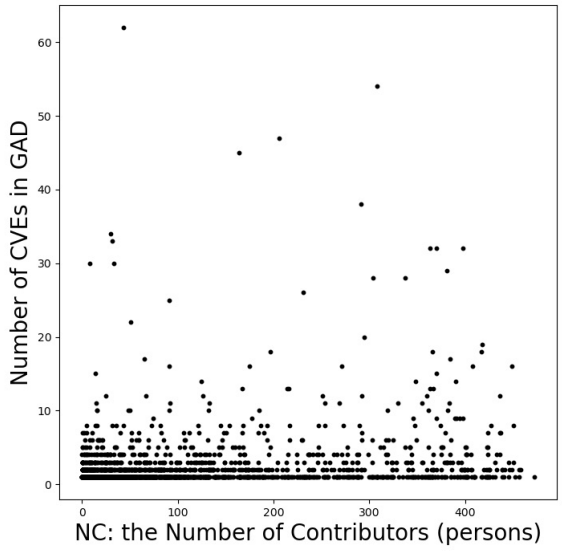

Figure 6: NC-CVE scatter plot analysis

Weak correlation between CVEs in GAD and NC. Figure 6 shows the correlation between the number of vulnerabilities and NC. The graph shows that the higher NC, the lower the overall number of vulnerabilities in the repository. This means that the more contributors, the better the security of the repository, but the relationship does not appear clear. According to Linus' law, it was assumed that many people would be safer to participate in the development, but the figure showed that it did not always work.

Details of BP. In the case of branch protection, 2,674 packages were detected. The repository whose BP value is 'n/a' is hiuminhnv/Zenario-CMS-last-version. The project was deemed to have deleted and was therefore excluded from the analysis. Out of the total number of packages, 1,187 had branch protection, while 1,487 had no branch protection.

Weak Correlation between CVEs in GAD and BP. Since the branch protection is true or false metric, we only need to compare the measurements for each result. The average number of vulnerabilities in the repository where BP is true was 2.81, and the average number of vulnerabilities in the repository where BP is false was 1.70. Since BP is an option to protect branches from attackers, it can be interpreted that it is not related to the number of vulnerabilities from developers' mistakes.

4.3 Case Study

Highest MU. The repository with the highest MU measured was the Snorby/snorby, [17] which recorded 3481.0. The repository is a ruby on rails web application for network security monitoring. Snorby was released just once in 2013, so MU was measured high. There was just one vulnerability in snorby.

Lowest MU. The repository with the lowest MU measured was rack/rack [24] with 0. Rack provides a minimal, modular, and adaptable interface for developing web applications in Ruby. MU was calculated as 0 since the first release occurred a few hours before the metric measurement.

Highest MC. The repository with the highest MC measured is nrako/psnode, [18] which recorded 3058. Psnode is a Node.js KISS module to list and kill process on OSX and Windows. The repository had only one commit on July 22, 2014. This is a case in which the MC value was measured high because there was no commit since then.

Lowest MC. The lowest measured repository for MC is liferay/liferay-portal. [25] Liferay-portal is produced by the worldwide Liferay engineering team, and involves many hours of development, testing, writing documentation, and working with the wider Liferay community of customers, partners, and open-source developers. Linux has had more than 669k commits since its first commit was made on April 16, 2006. The MC of linux is 0.01, which means that about 100 commits were made a day.

Highest IP. The repository with the highest IP is spejman/festivaltts4r. [22] FestivalTTS4r is an interface to Festival TTS Speech Synthesis System. There were 26 commits in this repository, and the last commitment was made on January 17, 2009.

Highest NC. The repository with the highest NC is esphome/esphome. [23] ESPHome is a system to control your ESP8266/ESP32 by simple yet powerful configuration files and control them remotely through Home Automation systems. There are 473 contributors in esphome.

4.4 Summary

In conclusion, this study proposes AutoMetric as an automated method for measuring the safety of OSS, and identifies a correlation among the five metrics, particularly in time metrics. The study recommends the use of MC, which measures the minimum unit of code changes, as an effective metric for measuring software security.

AutoMetric is specifically designed to measure the overall security of software groups with multiple sub-components, such as operating systems. Therefore, it is expected that significant differences will exist when comparing the highest and lowest values

covered by the case study. It is not the purpose of this study to compare values at the extremes one-to-one from a micro-perspective. However, there is a need to develop a mechanism for handling outliers covered in the case study.

5 DISCUSSION

Adding other metrics. AutoMetric currently consists of five metrics, and there is a possibility that other metrics will be added. In order to add other SSMs, the possibility of automation should be evaluated first. If it is determined that the possibility of automation exists, automatic measurement technology may be developed by matching the corresponding metric with the element of the repository.

Limitations of information available from repositories. Each project has unique purposes for utilizing their repository, depending on the policies of the repository manager. For example, some projects immediately commit code changes to the repository upon modification, while others collect changes and commit them at specific intervals. These differences can result in varying AutoMetric results. Additionally, the package manager can collect more accurate information about versions. GitHub stores information based on commits, not releases, making it difficult to get an accurate version history. Further research is necessary to address this issue.

Too many contributors. If the repository has too many contributors, the GitHub API cannot return the list of contributors, and it is impossible to measure the total number. For example, Linux repository is a repository for developing Linux kernels with more than 10,000 contributors. If you send a request to the repository via the Contributors API, an error occurs.

None-code repositories. While most repositories are used for project development, there are also repositories used for education and information dissemination. The-Art-Of-Programming-By-July, for example, is a repository that provides an e-book for algorithm learning and is ranked at the top of C Star Ranking with 20.6k stars. When measuring the metric for the repository, IP was found to be 508 days since the last commit. For a typical repository, this would suggest a wrong state. However, since the repository is not directly related to security, the meaning of the measured value is not significant. If these repositories are included in the statistics, the statistics can be misleading.

6 RELATED WORK

Software supply chain security. Ruian Duan et al.[10] modeled the package management workflow and found root causes of supply chain attacks. In addition, they proposed the vetting pipeline MALOSS to measure supply chain attacks on package managers for interpreted languages. It is similar in that it is a study to reduce supply chain attacks.

Other security metrics. Nikolaos et al.[26] found the average vulnerability life and minimum value of software such as Linux and Firefox through a large-scale empirical study on the life of vulnerabilities within FOSS. This is similar to evaluating software through a metric of vulnerability life.

Possibility of utilizing SBOM. Software Bill of Materials (SBOM) is a method for accurately recording components contained in software. SPDX [27], the most widely used SBOM format, provides a tool for automatically generating SBOM through metadata, and a technology for extracting SBOM from source code is being developed. If an SBOM for the target software can be extracted, the Git addresses of the sub-components that make up the software can be identified through the SBOM, enabling the software to be inspected all at once regardless of its size. As SBOM become more standardized, the usefulness of AutoMetric is expected to increase.

Detecting vulnerable code clones. There exist several code clone methods that may not be considered direct measurement approaches, but they can be useful in enhancing software security. Vuddy [28] is an effective and scalable technique for identifying vulnerable code clones that can accurately detect security weaknesses in large software systems. CENTRIS [29] is another method that can efficiently improve the detection of nested elements. By utilizing V0Finder [30], time metrics can be easily identified. These approaches have the potential to contribute to improving software security, and their effectiveness should be further evaluated and validated through empirical studies.

7 CONCLUSION AND FUTURE WORK

In the current trend of software development, where multiple OSS components are required, selecting the appropriate software components is a critical concern. To address this, we propose AutoMetric, a program that enables automatic measurement of security metrics. By providing repository addresses as input solely, AutoMetric can automatically measure multiple security metrics. Moreover, we analyzed the security metrics in AutoMetric by comparing them with the number of vulnerabilities. AutoMetric can aid developers in selecting more secure software components.

Future work. There is potential for further development and expansion of AutoMetric in various ways. Firstly, although strong correlations were found among time metrics, other SSMs did not show significant correlations, indicating a need to explore and identify better metrics. Additionally, AutoMetric currently requires human pre-processing, and automation of this process would increase ease of use.

ACKNOWLEDGMENT

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government Ministry of Science and ICT (MSIT) (No.2022-0-00277, Development of SBOM Technologies for Securing Software Supply Chains, No.2022-0-01198, Convergence Security Core Talent Training Business, and IITP2023-2020-0-01819, ICT Creative Consilience program).

AVAILABILITY

The source code of AutoMetric is publicly available at GitHub: https://github.com/HckEX/AutoMetric

REFERENCES

[1] Sonatype. https://www.sonatype.com/state-of-the-software-supply-chain, 2022.

[2] Linux Foundation. A summary of census ii: Open source software application libraries the world depends on. https://www.linuxfoundation.org/blog/blog/a-summary-of-census-ii-open-source-software-application-libraries-the-world.-depends-on, 2022.

[3] GitHub. https://github.com/about, 2023.

[4] GitLab. https://about.gitlab.com/company, 2023.

[5] Nist Cybersecurity Framework. https://www.nist.gov/cyberframework, 2022.

[6] Secure Software Development Framework. https://csrc.nist.gov/Projects/ssdf, 2022.

[7] Scorecard. https://github.com/ossf/scorecard, 2023.

[8] Github Advisory Database. https://github.com/advisories, 2023.

[9] NVD. https://nvd.nist.gov, 2023.

[10] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. *arXiv preprint arXiv:2002.01139*, 2020.

[11] National Institute of Standards and Technology. https://www.nist.gov/itl/executive-order-improving-nations-cybersecurity/security-measures-eo-critical-software-use-2, 2021.

[12] Ubuntu Package Management. https://ubuntu.com/server/docs/package-management, 2022.

[13] PyGithub. https://github.com/PyGithub/PyGithub, 2023.

[14] python gitlab. https://python-gitlab.readthedocs.io/en/stable/index.html, 2023.

[15] tensorflow. https://github.com/tensorflow/tensorflow, 2022.

[16] microweber. https://github.com/microweber/microweber, 2022.

[17] snorby. https://github.com/Snorby/snorby, 2022.

[18] psnode. https://github.com/nrako/psnode, 2022.

[19] andrewimm. https://github.com/andrewimm/xopen, 2023.

[20] xjamundx. https://github.com/xjamundx/gitblame, 2022.

[21] andrewjstone. https://github.com/andrewjstone/dynamo-schema, 2022.

[22] FestivalTTS4r. https://github.com/spejman/festivaltts4r, 2022.

[23] ESPHome. https://github.com/esphome/esphome, 2021.

[24] rack. https://github.com/rack/rack, 2022.

[25] Liferay. https://github.com/liferay/liferay-portal, 2022.

[26] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. What are weak links in the npm supply chain? In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 331–340, 2022.

[27] spdx. https://spdx.dev, 2023.

[28] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614. IEEE, 2017.

[29] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. Centris: A precise and scalable approach for identifying modified open-source software reuse. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 860–872. IEEE, 2021.

[30] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich. V0finder: Discovering the correct origin of publicly reported software vulnerabilities. In *USENIX Security Symposium*, pages 3041–3058, 2021.