The Difference in Using Natural or Artificial Datasets when Comparing Static Analysis Tools – A Controlled Experiment

John Alanko Öberg
Department of Computer Security
Blekinge Tekniska Högskola (BTH)
Karlskrona, Sweden
e-mail: joob17@student.bth.se

***Abstract*—Researchers conducting experiments which compare SATs often do so by running them on artificial test suites which are specifically created to test SATs. The purpose of a SAT is to aid in finding bugs and vulnerabilities in natural code, so the question arises whether the results of previous studies are meaningful for developers who intend to use the SATs to scan their code. The aim of this study is to find out whether SATs perform differently on artificial or natural datasets with regards to detected flaws (true positive rate). To test this, a controlled experiment was conducted where two SATs were ran on both natural and artificial datasets (matched pair design). The results showed that one tool performed significantly better on the artificial dataset while the other tool performed equally on both. The biggest limitation to the study was the available budget, meaning that the scope of the experiment had to be severely limited. In conclusion, the results of the study threatens the validity of existing experiments which have used artificial code for testing SATs.**

I. Introduction

A common method for detecting bugs and security vulnerabilities in software is by utilizing Static Analysis Tools (henceforth SATs). SATs are a fast and cheap way to find some but not all problems with written code. There are several commercial and open-source tools available for use, each specializing in different types of vulnerabilities, bugs, platforms or languages.

Numerous studies have been conducted to determine which of these tools perform the best with synthetic datasets as testbeds [1], [6], [2]. Several different metrics have been used, such as precision, recall, false positive rate, true positive rate, ease of use, execution time, vulnerability type etc. Some reports [7], [8] have suggested benchmarks for this purpose.

To evaluate the effectiveness of different SATs, test data is needed. There exist artificial test suites specifically made for the purpose of evaluating the performance of SATs, such as the JULIET test suite [9]. Such test suites are appropriate for evaluating different SATs in the regard that the seeded faults are well documented (in terms of what type of error and where in the code it is located), and as such are commonly used by researchers. In other regards such as code complexity and realism however, artificial test suites often falter.

It is a well determined fact that for the results of a study to possess external validity - that is be applicable in a real-world setting - the sample must be representative of the population. But are artificial test sets representative of natural[1] or "real" code?

In the documentation for the JULIET test suite, a remark is made regarding the limitations of the test cases. It states the test cases are simpler than natural code and that tool results may be inflated due to the tools reporting flaws that they wouldn't find in natural, non-trivial code.

Daniel Marjamäki, a developer for the SAT CPPCheck has the following to say about the JULIET test suite: "In my opinion, the Juliet Test Suite is not well-made. The code is weird and unrealistic." [12].

Some researchers [5], [3] have shown awareness of the importance of selecting representative datasets in their tool evaluation studies, by highlighting drawbacks of synthetic or artificial datasets and disdaining from using them. However, there is a lack of studies exemplifying the difference in results from analysing different SATs on natural and artificial datasets respectively.

In this study I intend to find out if there is a significant difference in performance of tools when tested against both artificial and natural/realistic datasets. The study will be performed by analysing the results of two SATs (Flawfinder and CPPCheck) with respect to their buffer-related vulnerability detection rate on two sets of data: the JULIET test suite and the Linux kernel source code. The scope of the study is 5 ECTS which translates to 135 person hours. Following is the Goal/Question/Metric (GQM) template filled out according to the objectives of this study:

Analyze performance results from using two SATs on natural and artificial datasets
For the purpose of comparing the results
With respect to buffer-related vulnerability detection rate
From the point of view of SAT researchers and users
In the context of a university course in research methodology.

[1] The definition of natural code in this case is code which has been created for a purpose other than to test SATs.

This study will demonstrate the effects that selecting artificial or natural datasets can have on the performance outcome of different SATs. The results can influence the validity of previous and future experiments which makes use of artificial code for their test sets. If the SATs have a significantly higher detection rate on artificial code as compared to natural code, the validity of similar experiments could be put in jeopardy. In the best of worlds, this study can prompt the construction of an extensive, realistic dataset with well documented vulnerabilities for SATs to be tested on.

II. Related Work

Wagner and Sametinger (2014) [1] use the JULIET test suite to compare five open-source SATs. They discuss the topic of natural and artificial datasets and mention that artificial test cases are typically simpler than natural code in terms of complexity, and that tools might fail to find vulnerabilities in natural code. The study concludes that the results were disappointing except for one tool which performed well and that it is advisable to use more than one tool for finding bugs and vulnerabilities because of the different strengths of the tools. The authors also suggest that the "*Juliet Test Suite provide a powerful means of revealing the security coverage of static scanners, which in turn enables us to pick the right set of scanners for improved software coding*", but give no motivation as to why the results from the artificial dataset would generalize to natural datasets.

Li, Beba and Karlsen (2019) [2] evaluate coverage, performance and usability of five different IDE plugins on artificial test code. They conclude that several categories of vulnerabilities were undetected, which contradicted the coverage information in the tool documentations. Furthermore, most plugins had a high false positive rate and were not user-friendly. In the discussion they state the following: "*The results indicate how the plugins perform on generated code which is a good indication of what it detects objectively, but it does not say anything about their performance in a real-life setting. Detecting vulnerabilities in natural code is a different matter and the distribution of occurrences by vulnerabilities are very different*".

Delaitre et al. (2015) [3] evaluate SATs with respect to their bug finding capabilities. The authors acknowledge the need for test cases with three characteristics: statistical significance (size and diversity of code base), ground truth (the knowledge of where defects are located) and relevance (representativeness of real source code). "*In summary, the perfect test cases are a set of large production software, developed according to typical industry practices and whose defects are all identified*". They state that test cases possessing all three characteristics are out of reach, and instead use combinations of test cases which have two of the characteristics. When analysing the results, the authors found that the precision metric mostly scored higher on a synthetic dataset than on two production software, for four different tools. The results also show that the tools differ vastly in terms of which defects they detect, and the authors conclude that no tools prevail in all regards.

III. Experiment Planning and Execution

A. Considerations for Other Research Methods

The chosen research approach is a controlled experiment. This method was chosen because it allows such a specific research question to be answered by offering high control regarding the selection of dependent and independent variables as well as maintaining all other variables constant. Furthermore, an experiment is the way to go when studying the effect that changing one independent variable has on the dependent variable(s), which is done in this study. Finally, a small experiment fits perfectly in the scope of the study.

A case study would be unnecessary because having the study be performed in a real-life setting would bring few benefits. Usually a case study is performed when the research questions of the study would be difficult to answer or have less validity if attempted to be answered by an experiment or another method. In this study however, it is not the case.

A compelling alternative method is a systematic literature review. It would allow for gathering data from available studies and comparing the results. Since there are ample studies conducted on evaluating the performance of SATs with respect to vulnerability type, SAT type, measurement metrics, and some of which are applied on artificial datasets and some on natural datasets, a conclusion could be drawn to answer the research question of this study properly. In that sense, either an experiment or a systematic literature review would be appropriate for this kind of study.

B. Goals and Hypotheses

Following is the research question of this study: *What is the difference in performance of SATs when testing them on natural datasets as opposed to artificial ones?*

This question can formalized into the following hypothesis and its null-equivalent:

Main hypothesis: *Usage of natural or artificial testbeds have a significant impact on the vulnerability detection rate of SATs*

Null hypothesis: *Usage of natural or artificial testbeds have no significant impact on the vulnerability detection rate of SATs*

In order to evaluate the hypotheses, the experimental context must first be discussed.

C. Experimental Units and Material

The experiment will use two samples:

- The JULIET test suite representing a population of artificial test data
- Linux kernel source code representing a population of realistic test data

The type of test suite (artificial or natural) constitute the independent variable of this experiment. The JULIET test suite was selected because it is a well-established test bed

specifically made for testing SATs. It has been used extensively by researchers in the past and thus the results gathered in this study can be compared to those of other studies, increasing the validity. The Linux kernel source code was selected for its realistic characteristics, importance and prevalence in the real world. Both samples are written in C code.

The tools will be attempting to detect memory buffer errors. Such errors can lead to a wide range of vulnerabilities including code execution (injection), denial of service and sensitive data exposure, two of which are present on OWASP's top 10 list of vulnerabilities for 2017 [4]. For the Linux kernel specifically, buffer overflow has remained the third biggest threat since 1999 [11].

1) CWEs and Initial Dataset Filtering

Common Weakness Enumerations (CWEs) are descriptions of weaknesses that software and hardware can possess. Known vulnerabilities are often classified by CWE id. There exist several hundred types of different CWEs which are all assigned a category such as class, base or variant. For example, CWE 119 is a class which contains bases CWE 120 and 125 to name a few. Classes have generic descriptions while bases have more specific. The category with the lowest level of abstraction in the CWE system is called variant.

The JULIET test set includes CWEs 121-124, 126, 127 to name a few. These CWEs all relate to the CWE class 119: "Improper restriction of operations within the bounds of a memory buffer". In order to allow the SATs to scan for similar vulnerabilities with the lowest level of abstraction possible in both datasets, the choice of which CWE(s) to scan must be made. This choice must take into consideration whether a CWE is relevant for the experiment, if the SATs have the capability to detect it, and in which quantity it exists for either dataset. Bearing this in mind, test cases marked with CWE 126 and 127: "Buffer over/under-read" respectively, were chosen for the JULIET test set. These CWEs are variants of CWE 125: "Out-of-bounds read".

Because of the nature of ongoing software development, reported security vulnerabilities typically are not present in subsequent versions of the software. This means that finding a natural dataset with a multitude of known vulnerabilities is a difficult task. The strategy that will be used to mitigate this is to instead analyse multiple commits spanning several versions of the program in which a vulnerability was introduced to it.

To obtain the subsamples for the Linux kernel, the CVE Details website was used [11]. This website hosts a collection of known security vulnerabilities (CVEs) for different software, sorted by type and date to name a few. In total, the Linux kernel has 2712 such CVEs documented. Every CVE entry has exactly one associated CWE. For the Linux kernel, no further distinction than CWE 119 or 125 is usually present for any CVE entry in which a CWE 119 related vulnerability was reported. Thus, vulnerabilities marked with CWE 125 were chosen for the natural dataset.

2) Further Dataset Filtering and Acquirement

To gather the appropriate test cases for the artificial dataset, test cases which were marked CWE 126 and 127 and written in C code were considered. Given these filters, it remained to be decided which flaw types and flow variants to use. Flaw types are phrases which describe particular variants of a flaw and can be for example: *char_alloca_loop*, *malloc_char_memmove*, *wchar_t_declare_cpy*[2]. A test case with flow variant 02 or higher contains either added control flow or data flow complexity. A test case with flow variant 01 contains no such added complexity.

The best solution for determining which flaw type(s) and flow variant(s) to use would be to identify which are present in the Linux kernel dataset and use a similar distribution for the artificial dataset. To identify relevant flaw types and flow variants for even a few samples would require large amounts of work however. It was determined to be outside of the scope of this experiment. Instead, all flaw types with flow variant 01 for both CWE 126 and 127 will be used. This totals to 18 entries for CWE 126 and 29 entries for CWE 127. These entries were placed inside folders to be scanned.

In obtaining relevant test cases for the natural dataset, all entries on the CVE Details website marked with CWE 125 were considered, regardless of vulnerability category (e.g. DoS, Overflow, Code execution). If there was a link in the references section containing the commit which fixed the security issue, the parent file(s) of that commit were downloaded for scanning. This ensured that the vulnerable file(s) were being scanned. If no commit hash could be found in the details of the CVE entry, its parent file(s) were not added to the subsample selection. In total, 10 out of 48 available commits with CWE 125 were omitted.

The parent files(s) of the commit were subsequently put inside a folder named by that CVEs ID, which in turn was placed in a folder named by its vulnerability type. For some CVE entries, vulnerability was of more than one type, in which case it was listed one time within every respective category on the website. Every such entry was downloaded only once and placed in the first category it was found in, and subsequently put on a blacklist to ensure it wouldn't be scanned multiple times.

With the relevant subsamples acquired for both datasets, scanning could begin.

3) The SATs

For comparing the results from the two datasets, Flawfinder and CPPCheck was used. The tools were chosen based on their popularity, simplicity to install/use, applicability to C and code and the fact that they are open-source.

For measuring the results of the tools, detection rate or true positive rate was used as metric. This is the dependent

[2] If the reader finds this description unintuitive, they can take solace in the fact that it is just as unintuitive for the author.

variable. While vulnerability detection rate is not the only metric traditionally measured when evaluating SATs, it serves as a suitable candidate for finding out if there is a significant difference in the type of testbed used. Furthermore, the use of additional metrics was outside the scope of this experiment.

Section 8 of the JULIET test suite manual covers how SATs should be used on its dataset. It states that if a tool finds the appropriate flaw anywhere inside a function marked by "bad", the tool has found a true positive. This was the method of tool evaluation for the artificial dataset. For the natural dataset, the tools were considered to have found a true positive if they found the appropriate flaw anywhere in the function(s) in which the fix(es) to the vulnerability were present. If multiple functions were vulnerable and a tool detects a vulnerability in only one, it was counted as a hit.

Flawfinder has a set amount of CWEs it can detect, including but not limited to CWE 119, 120, 126, 134. The appropriate vulnerability was considered found if Flawfinder detected CWE 119 or 126 in any of the datasets (only CWE 119 for the CWE 127 subsamples of the JULIET dataset).

CPPCheck uses a different approach for reporting errors, not relating any errors to CWEs. To determine if CPPCheck found the correct vulnerability, each hit in a vulnerable function was individually analysed.

D. Tasks

Flawfinder was ran on the acquired subsamples from the artificial dataset using the following command in the command line: *flawfinder -e "CWE-119|CWE-126" <path> > "Flawfinder_126.txt"*, where *<path>* was the path containing the CWE 126 datasets. The output was restricted to only include identified CWE 119 and 126 vulnerabilities, and it was redirected from the command line to a text file. Similarly for the CWE 127 subsamples, the following command was ran: *flawfinder -e "CWE-119" <path> > "Flawfinder_127.txt"*.

CPPCheck was ran on the artificial subsamples with the following command: *cppcheck --output-file=CPPCheck_126.txt --quiet --inconclusive <path>*. The command was ran one time for each CWE folder, with the appropriate path and output text file. The *--quiet* flag supresses output that are not errors or warnings while the *--inconclusive* flag includes flaws which the program is less certain of.

The two SATs were run in the same way for the natural dataset.

E. Expertiment Design

The experiment utilized a within-subjects design, specifically a matched pairs design with the same experimental units. Such a design applies all treatments on all participants, in contrast to the between-subjects design in which each participant only is given one treatment. In this study, the participants are objects in the form of the two SATs, and the treatments are the two datasets. Usually in such a design, a randomized ordering of applying the treatments is necessary for the study to be considered a true experiment and not a quasi-experiment. Because of the objectivity present in the participants however, this was not deemed necessary. For all intents and purposes, the ordering chosen might as well have been determined randomly.

An interesting alternative method which was out of the scope for this study (due to limited number of SATs/vulnerabilities allowed) was the matched pairs design with similar units. It differs from the abovementioned matched pairs design in the sense that it considers the participants to be slightly different from each other in one or some aspects. For example, such a design could assign the participants into groups of two based upon the marketed strength of each tool with respect to specific vulnerabilities (or based on other factors which could affect the result). A participant in each group would then randomly receive one of the treatments while the other participant in the group receives the other treatment. By using this design, the conclusion of the experiment could take into consideration the differences in results for each group and thus extend the external validity of the study.

Another common design type is the randomized block design. In this design, the participants are divided into blocks based on a variable believed to influence the results, called the blocking variable. The design is randomized because the participants are randomly assigned to treatment-groups within their respective block, each group receiving one treatment (in accordance to the between-subjects design). This design type was not preferred for this type of experiment because there are inherent differences in the participants (SATs) and thus comparing each participant to itself by way of having them receive all treatments is necessary.

IV. Results

The results of the experiment can be seen in Figure 1 in the Appendix. Flawfinder performed best on the artificial test set, correctly identifying all vulnerabilities, while only identifying one vulnerability in the natural test set. CPPCheck performed equally on either test set, finding no appropriate vulnerabilities. The sample datasets used in the experiment and the output of the scanners can be found here [13].

V. Discussion

From the results it is clear Flawfinder performed better on the artificial dataset. This can be attributed partly to the evaluation technique used (true positive is recognized if correct vulnerability is found *anywhere* in the function) and partly to the simple nature of the tool. Flawfinder markets itself as a "simple text pattern matching" tool [10]. It scans the software for use of potentially dangerous functions and syntax. As such, safe usage of such syntax is also reported by the tool, potentially resulting in a lot of false positives. While not the focus of this study, looking at the output from Flawfinder it is obvious that this was the case. In fact, Flawfinder reported every instance of a *char* -or *wchar_t* array declaration as a potential CWE 119 vulnerability. While

an array can be used to cause CWE 119 related errors, it is also an essential part of the C programming language. The declaration of such arrays were present in all the samples of the artificial dataset, in both the "good" and "bad" functions. This is the reason for Flawfinder's 100% true positive (and false positive) hit rate. In a similar manner, Flawfinder also reported every usage of functions strlen and wcslen as potential CWE 126 related vulnerabilities. For the natural dataset, the only true positive found was a CWE 119 threat on a line of code which created a char array.

The results from CPPCheck were disappointing, even with the *–inconclusive* flag enabled. The tool did not find anything to report for the CWE 126 samples of the artificial dataset, and only memory leaks for the CWE 127 samples. For the natural dataset, the majority of reports were regarding uninitialized variables, while the remaining reports pertained to errors not related to CWE 119.

A. Threats to Validity

1) Internal Validity

The internal validity is threatened by the fact that only one dataset was used to represent an entire population of datasets (artificial or natural). For example, the difference in results between the natural and artificial datasets might be explained by the fact that one dataset had harder to detect vulnerabilities than the other, which was not necessarily caused by it being natural or artificial. Such a difference in results could just as well be obtained by for example comparing two artificial datasets. To mitigate this, more artificial and natural datasets would have to be evaluated.

When acquiring data for the artificial dataset, only test cases with flow variant 01 (the simplest type of the flaw) were used, because a conclusion regarding which specific flow variant(s) to use could not be reached. This is a validity threat because only the flaws with the least amount of complexity were scanned for the artificial dataset, while for the natural dataset, all entries available marked with CWE 125 were scanned. This could be a reason for why the results are skewed in favour of tools finding vulnerabilities in the artificial dataset. As mentioned in section III, a better approach would have been to map the flow variants to the natural dataset and scan a similar distribution for both.

Something which might threaten the validity regarding the natural dataset is that the scanners only scanned the parent files to the commit which resolved the security vulnerability, and not the entire parent system. This being a validity threat is a reasonable assumption, since a tool might need the vulnerable files in the context of the rest of the system in order to determine if there is a vulnerability, just as a human would in some cases. Because of the simplistic nature of the tools however, this was deemed unnecessary. Furthermore, the vast amounts of network and disk usage required for this task would be unreasonable and outside the scope of this small experiment, as one would have to download several versions of the entire system.

The tool evaluation technique of only accepting a true positive from the SATs if they could find the vulnerability within the same function(s) in which the fix was present, is flawed. Consider for example that a SAT finds a vulnerability inside a function which is used by many other functions, i.e. the source of the vulnerability. If that specific source function is not present in a fix commit, the result will be that a hit was not registered within the correct location, when in reality it could have been that the fix commit only put out fires in other functions and left the source of the fire untouched. In this case, the SAT would be right in its prediction, without receiving a true positive hit. A mitigation technique would be to assess the flow of the code to determine which functions are used by other functions. For an entire operating system, this would inconvenient to say the least. Furthermore, a function might be very long, and a tool finding the correct vulnerability somewhere inside that function might not be very useful since it might not have anything to do with the vulnerability.

2) External Validity

There are several threats to external validity due to the small scope of the experiment. It would be difficult to motivate that the results will generalize beyond the two specific tools used, the type of vulnerability analysed and the metric used to compare them. For external validity to increase, the number of SATs, vulnerability types and measurement metrics must increase. For example, both SATs used were open-source, maybe commercial tools would show different results due to presumably being of higher quality. And perhaps the chosen vulnerability type is equally difficult to detect for any type of dataset, while another vulnerability type is not. Furthermore, the results might not be similar to those obtained by using testbeds with different programming languages, as the SATs vulnerability detection capabilities would most likely differ for different languages.

VI. Conclusions and Future Work

Given the tool results, the null hypothesis can be rejected: SATs perform better on artificial datasets than they do on natural datasets. The experiment has noticeable threats to its validity however. A future study could eliminate many of said threats if given a larger budget and by evaluating a larger scope such as by using additional SATs, dataset samples and measurement metrics. Still, there is a need for a large, realistic test suite with documented flaws in order to accurately evaluate SATs.

References

[1] A. Wagner and J. Sametinger, "Using the Juliet Test Suite to compare static security scanners," *2014 11th International Conference on Security and Cryptography (SECRYPT)*, 2014, pp. 1-9.

[2] Jingyue Li, Sindre Beba, and Magnus Melseth Karlsen. 2019. Evaluation of Open-Source IDE Plugins for Detecting Security Vulnerabilities. In *Proceedings of the Evaluation and Assessment on Software Engineering* (*EASE '19*). Association for Computing Machinery, New York, NY, USA, 200–209.

[3] A. Delaitre, B. Stivalet, E. Fong and V. Okun, "Evaluating Bug Finders -- Test and Measurement of Static Code Analyzers," *2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS)*, 2015, pp. 14-20

[4] https://owasp.org/www-project-top-ten/2017/Top_10.html

[5] B. Liu *et al*., "A Large-Scale Empirical Study on Vulnerability Distribution within Projects and the Lessons Learned," *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 1547-1559.

[6] A. Arusoaie, S. Ciobâca, V. Craciun, D. Gavrilut and D. Lucanu, "A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code," *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2017, pp. 161-168

[7] Higuera, Juan R. Bermejo, et al. "Benchmarking Approach to Compare Web Applications Static Analysis Tools Detecting OWASP Top Ten Security Vulnerabilities." *Computers, Materials and Continua* 64 (2020): 3.

[8] P. Nunes, I. Medeiros, J. C. Fonseca, N. Neves, M. Correia and M. Vieira, "Benchmarking Static Analysis Tools for Web Security," in *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1159-1175, Sept. 2018

[9] https://samate.nist.gov/SRD/testsuite.php

[10] https://dwheeler.com/flawfinder/#how_work

[11] https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33

[12] https://stackoverflow.com/questions/57444094/can-cppcheck-static-code-analyzer-actually-detect-not-very-common-warnings-like

[13] https://github.com/Jonnen98cool/PA1433-A3-Data

APPENDIX

	Linux kernel vulnerabilities by category, years 1999-2022 (www.cvedetails.com)								JULIET test suite		
	Bypass something	Code execution	DoS	Gain information	Gain privileges	Memory corruption	Overflow	Total	CWE-126	CWE-127	Total
Total vulnerabilities any CWE	133	262	1323	400	260	141	394	2913	N/A	N/A	N/A
Total vulnerabilities CWE-125	1	3	31	9	1	0	3	48	N/A	N/A	N/A
Vulnerabilities for which a commit was not found	0	1	4	4	0	0	1	10	N/A	N/A	N/A
Vulnerabilities scanned	**0***	**2**	**23****	**4***	**0***	**0**	**2**	**31**	**18**	**29**	**47**
Flawfinder TP:s	N/A	0	1	0	N/A	N/A	0	1	18	29	47
Flawfinder hit rate (%)	N/A	0%	4.3%	0%	N/A	N/A	0%	3.2%	100%	100%	100%
CPPCheck TP:s	N/A	0	0	0	N/A	N/A	0	0	0	0	0
CPPCheck hit rate (%)	N/A	0%	0%	0%	N/A	N/A	0%	0%	0%	0%	0%
	*One vulnerability in this category is of multiple types and is instead included in the DoS category										
	**There exists four multiple-type vulnerabilities in this category which have already been included in other categories										

Figure 1: Results from running the two SATs on the test sets.