Benchmarking Open Source Static Analysis Security Testing Tools for C

Christoph Gentsch, Rohan Krishnamurthy, and Thomas S. Heinze

German Aerospace Center (DLR),
Institute of Data Science,
Jena, Germany
{christoph.gentsch,rohan.krishnamurthy,thomas.heinze}@dlr.de

Abstract. As the number of available static analysis security testing (SAST) tools grows, the more difficult it becomes for developers to decide which tool(s) to use. We report on our evaluation of 11 open source SAST tools for the C programming language on the Juliet Test Suite for C/C++ and of six tools on the Wireshark production software. In line with the previous work, we find that there is no single superior tool, though sound tools performed the best on the Juliet test cases.

1 Introduction

In recent years, the market for *static analysis security testing (SAST) tools* has expanded, since software security increasingly becomes a concern and draws more and more attention. As a result, there exists a variety of tools, whether commercial or open source, which claim more or less to do a "security analysis". Some of these tools are basically simple syntax checkers, which apply context-insensitive pattern matching to check program source code for compliance to a certain standard or best practices. As these tools certainly can help in improving code style and security to some extent, it is though questionable, how good they are at finding, e.g., deep nested memory errors or concurrency problems. Then there are more sophisticated, semantic analyzers, which focus on finding bugs and vulnerabilities instead of compliance checking, but make no claims on the complete absence of vulnerabilities or run-time errors. Last, there are so-called "sound" semantic analyzers, which mostly rely on abstract interpretation of source code and promise to make "sound" propositions. What this means in practice can however vary – e.g., for the developers of *Frama-C* it means to "aim at being correct, that is, never to remain silent for a location in the source code where an error can happen at run-time"[1]. The developers of *Infer* state: "soundness" does not translate to 'no bugs are missed'. The role of soundness instead is to serve as an aid to pinpoint what an analysis is doing and to understand where its limitations are; in addition to providing guarantees for executions under which the model's assumptions are met"[2].

[1] cf. Frama-C website (https://www.frama-c.com)

As the variety of tools and techniques can be confusing for a developer, it is apparent that an evaluation of their abilities is desirable. The objective of our paper is to conduct such an evaluation and, in particular, to measure to which extent open source SAST tools are capable of finding vulnerabilities in C code. The focus on C has several reasons: C is still one of the most important programming languages around. Approximately one third of the software packages of Debian Linux are written in C. In the field of (in-)security, C also stands out: A small fraction of C programs is responsible for 50% of all vulnerabilities in Debian Linux (cf. Sect. 3). There exist several tools for analyzing C code and datasets with C code to analyze. The *SARD Juliet Test Suite* [4] provides a state-of-the-art benchmarking dataset and is also the basis for our evaluation. The limitation to open source tools comes from our requirement to have for anyone reproducible results and avoid problems with *DeWitt clauses*, as used in many license agreements of commercial tools [11]. Additionally, as open source developers usually use open source tools themselves, this is also a test of the ability of the open source community to facilitate secure software development.

The question we want to answer is: *If I, as a software developer, use this tool, what can I expect of the security of my code?* In this light, a SAST tool can also be seen as a kind of "insurance" for the developer: "*If I run this tool, my code is mostly secure*". Apparently, this proposition only holds if the used tool is able to identify most types of defects which threaten security. Therefore, we need a benchmark which reflects the most common vulnerabilities, weighted for their observed frequency, or "prevalence", and test every tool on it. This takes away the burden from the developer, to study the capabilities of each tool, and decide, if he needs a tool capable of detecting vulnerability X or vulnerability Y, or both. This approach does though not bias the calculated precision: A tool which only detects format string vulnerabilities – but with no false positives – would gain, on the one hand, 100% precision, but on the other hand, also a low recall over the whole test suite, since it only detects format string vulnerabilities. However, this perfectly reflects the tool's "narrowness" or "specialization". Our goal is to ultimately have a single measure for each tested tool, which tells a developer, how "secure" he can be when using the tool. Thus, the contributions of the paper can be summarized as follows:

- We evaluated 11 open source static analysis security testing tools for C on the synthetic *Juliet Test Suite* and report on the tools' accuracy.
- We analyzed the prevalence of vulnerability patterns in the *Juliet Test Suite* and in *Debian Linux*, as a representative for production software.
- We conducted a trial of the six most promising SAST tools on *Wireshark v1.8* and report on the tools' recall in finding real vulnerabilities.

The rest of the paper is structured as follows: In Sect. 2 we review related work and discuss the challenges when evaluating SAST tools for C. An investigation on the prevalence of C-related vulnerabilities for the Debian Linux distribution is presented in Sect. 3. Based upon this, we introduce the evaluation methodolgy and dataset in Sect. 4 and present the results in Sect. 5. Threats of validity are discussed in Sect. 6 and, eventually, Sect. 7 concludes the paper.

2 Previous Work on Benchmarking SAST Tools for C

Several authors have analyzed the effectiveness of SAST tools in finding vulnerabilities in C code. Chatzieleftheriou and Katzaros [3] tested four open source tools and two commercial tools using their own test suite of C programs, covering 30 vulnerability patterns selected from the *Common Weakness Enumeration (CWE)*[2] catalogue and the *CERT C Secure Coding Standard* [12]. The test suite included “bad” code, representing real vulnerabilities, and “good” code, representing spurious findings if reported by a SAST tool, for each vulnerability pattern and thus allowed to assess the tools effectiveness in terms of precision and recall. They found the two commercial tools to rank best, achieving a F-Score of 0.85, closely followed by the open source tools *Frama-C* (0.8) and *UNO* (0.7). *CppCheck* and *Splint* ranked at the lower end (<0.6). The same test suite was used later in [9] to compare the tools *Splint*, *Cppcheck*, *Frama-C*, *Infer* and *Clang*. *Clang* and *Frama-C* lead again with a F-Score>0.9, followed by *Infer* (0.88) and *Cppcheck* (0.78). The good scores may indicate that the test suite is now too outdated to give a good measure for comparison of SAST tools.

The *SARD Juliet Test Suite* for C/C++ was first published in 2011 and last updated in 2017 [4, 10]. The test suite covers more than 100 different CWEs in 64k test cases. Similar to Chatzieleftheriou and Katzaros [3], each test case has a “bad” and a “good” function, together with the labels for the tested CWE. As half of the test cases thus consists of “bad” functions, this implies that the measured precision is only a precision for 50% prevalence – which means, there is a 50% chance for a SAST tool to just *guess* the right location. To address this, the authors suggested another metric in [10], the discrimination rate, wherein the findings counted as true positive only if a tool reported the “bad” function, but not the “good” function. Note that the type of error (CWE) also has to match with the CWE associated to the test case. As this is a feasible procedure in theory, the practical application is not as straightforward, especially when testing open source tools. Those tools often do not report CWEs, or just report some abstract CWE class, whereas the *Juliet Test Suite* asks for specific CWE variants. One solution to this is proposed by Goseva-Popstojanova and Perhinschi [5], who also used the *Juliet Test Suite* for benchmarking three anonymous commercial tools. They carried out a fuzzy matching of vulnerabilities by matching CWEs that are closely related in the CWE hierarchy. This approach though does not help in all cases. Consider the following example from the *Juliet Test Suite*[3]:

Listing 1.1: Example Code from the Juliet Test Suite for C/C++

```c
data = -1;
if (data < 100)
{
        char * dataBuffer = (char *)malloc(data);
...
```

[2] `https://cwe.mitre.org/`
[3] File `CWE195_Signed_to_Unsigned_Conversion_Error__negative_malloc_18.c`

Here, a SAST tool could report *CWE-686* (function call with incorrect argument type) at line 5, since *malloc* expects unsigned integers. Otherwise, a *CWE-131* (incorrect calculation of buffer size) would also be correct. The *Juliet Test Suite* however expects *CWE-195* (signed To unsigned conversion error). The complexity of the CWE hierarchy is causing these issues. With more than 800 weaknesses, it is very fine-grained and in addition offers more than 30 different views, which change the relation between CWEs. In this jungle, chosing the correct CWE is hard for test suite designers and tool developers.

For a pragmatic approach, one could, instead of matching CWEs, sophistically design the test cases in a way that only one type of vulnerability can be identified at a specific location. This approach was taken by Shiraishi et al. [13] and lead to the *ITC Benchmark.* This test suite has the advantage of being more easy to comprehend, as there is no need to check the location of a vulnerability and no CWE mapping has to be done. However, the tests are rather easy such that static analysis tools can achieve up to perfect results on this benchmark in case of some defect patterns [13]. As a result, comparing tools above this cutoff is infeasible. Other issues regarding the *ITC Benchmark* include the inclusion of unintended defects in the test cases, wrong and/or missing vulnerability markers and the selection of types of defects in the test cases, which is not representative [6]. Besides, the *ITC Benchmark* has been designed with respect to safety tools such that tests for security-related defects and flaws concerning input validation, path traversal, or code injection are completely missing. The benchmark is though used in [1] to compare open source SAST for C. According to the results, *Clang* and *Frama-C* are leading the field, closely followed by *CppCheck.*

While the *Juliet Test Suite* was primarily designed for evaluating the effectiveness of commercial SAST tools and benchmarking results are therefore presented anonymously [4], Lu et al. [8] report on a more recent comparison of open source SAST tools and one commercial tool. They used the *Juliet Test Suite*, but left the method for evaluating the tools findings and measuring precision and recall unspecified. Despite the included commercial tool, the best F-Scores were reached by *CppCheck* (0.34), *Frama-C* (0.3), and *Clang* (0.3).

3 Prevalence of C-Related Vulnerability Patterns

To investigate which C-related flaws cause the most vulnerabilities found in production software, and therefore should be included in a test suite for benchmarking SAST tools, we analyzed vulnerabilities and vulnerability patterns reported for the Debian Linux distribution[4]. Debian comes with thousands of software packages, ranging from desktop applications like OpenOffice or Firefox to web servers like Apache or shells like bash, which makes it a representative example for production software. As a first step, we downloaded all software packages from the stable Debian release when conducting our research, i.e., *Debian 9 "Stretch"*. We then gathered reported vulnerabilities for the packages, as listed by means of

[4] `https://www.debian.org`

Common Vulnerabilities and Exposures (CVE) in the packages' changelog files. CVEs are vulnerabilities, which were reported to the public CVE database[5].

Table 3.1: Statistics on Debian vulnerabilities

Packages total:	24,438
Packages containing CVEs:	1,327
Packages with C as main language:	8,132
Packages with C as main language containing CVEs:	838
CVEs since 1988 (not only Debian):	103,193
CVEs in the Debian packages:	10,472
CVEs in all C packages:	5,639
⌀ CVEs per package:	0.4
⌀ CVEs per C-package:	0.7

As can be seen in Table 3.1, we found 10k CVEs reporting vulnerabilities for all Debian software packages. Note that this accounts for approx. 10% of all CVEs ever reported since 1988. Considering only the Debian packages with C as the main language, we found that 10% of those packages contain 50% of all CVEs reported for the Debian packages. The average count of CVEs in a C package is consequently also higher than the overall average of CVEs per package.

Table 3.2: Top-10 vulnerabilities in Debian packages with C as main language

Common Weakness ID	Percentage
CWE-119 (Improper Restriction of [..] the Bounds of a Memory Buffer)	32.1%
CWE-20 (Improper Input Validation)	10.5%
CWE-125 (Out-of-bounds Read)	10.0%
CWE-399 (Resource Management Errors)	8.1%
CWE-190 (Integer Overflow or Wraparound)	6.1%
CWE-476 (NULL Pointer Dereference)	5.0%
CWE-787 (Out-of-bounds Write)	4.3%
CWE-284 (Improper Access Control)	3.6%
CWE-264 (Permissions, Privileges, and Access Controls)	3.8%
CWE-416 (Use After Free)	3.6%

Combining the gathered CVE data with data from the *National Vulnerability Database (NVD)*[6] allowed us to map individual CVEs to vulnerability patterns, i.e., CWEs, and weight them with their severity scores. This way, we were able to analyze which types of vulnerabilities were found in those Debian packages,

[5] `https://cve.mitre.org`

[6] `https://nvd.nist.org`

whose main language is C. To gain an impression of the current siutation, we only considered CVEs as reported in the years 2017-2018. Table 3.2 presents the Top-10 CWEs for C packages, according to the summed up NVD severity scores. Apparently, the "buffer overflow" is still the leading flaw in C programs, followed by several other memory-related vulnerabilities. To provide a more comprehensive overview, we also mapped the C-related CWEs to the clusters of the *Software Fault Patterns (SFP)* view of the CWE hierarchy[7].

Table 3.3: Top SFP clusters found in Debian packages with C as main language

Cluster	Percentage
Memory Access	49.3%
Resource Management	12.8%
Tainted Input	10.7%
Risky Values	6.8%

As shown in Table 3.3, the top fault cluster is memory access with a fraction of approx. 50% of all found vulnerabilities Members of this cluster are vulnerabilities like, e.g., faulty buffer access, faulty pointer use, improper NULL termination. The second-severe fault cluster resource mnagement with a fraction of 12% includes vulnerabilities like use after free or missing release of file handles. Tainted input with approx. 10% is related to missing sanitization of data from user input. Finally, risky values consists of vulnerabilities related to integer overflow, incorrect type conversions or divides by zero. Note that this result is on par with the findings by other research [7]. Altogether, it seems like most vulnerabilities come from implementation errors which have been in the focus of static analysis tools for many years. In the following, we report on our evaluation of the capabilities of SAST tools finding these errors.

4 SAST Tool Evaluation Method

4.1 Tested Open Source SAST Tools

The open source SAST tools for C which we included in our evaluation are shown in Table 4.1. We included static analysis tools, but decided to omit pure linters, e.g., *Lint*, *cpplint*, and model checkers, e.g., *Blast*, *CPAchecker*, due to their focus. Also, we did not include tools which are not maintained anymore, like *Splint*, *RATS*, or *ITS4*. Besides, three sound SAST tools have been tested: *Infer*, *IKOS*, and *Frama-C*. The latter was already evaluated on the *Juliet Test Suite* in [4], but in a separate category for sound tools, which makes it difficult to compare with the other tools. For all of the tools, we chose the latest stable version in Ubuntu 18.04 LTS when conducting our research.

[7] `https://cwe.mitre.org/data/definitions/888.html`

Table 4.1: In our study evaluated static analysis tools

Tool	Version	Reference
AdLint	3.2	`http://adlint.sourceforge.net`
Clang-Tidy	4.0	`http://clang.llvm.org/extra/clang-tidy/index.html`
Clang Scan-Build	4.0	`http://clang-analyzer.llvm.org/scan-build.html`
CppCheck	1.72	`http://cppcheck.sourceforge.net`
Flawfinder	1.31	`http://dwheeler.com/flawfinder`
Frama-C Eva	17	`http://frama-c.com`
IKOS	2.0	`http://github.com/NASA-SW-VnV/ikos`
Infer	0.15	`http://fbinfer.com`
OCLint	0.13.1	`http://oclint.org`
Pscan	1.2-9	`http://deployingradius.com/pscan`
Sparse	0.5.0	`http://sparse.wiki.kernel.org/index.php/`

4.2 Evaluation Datasets

As discussed in the report of the *Static Analysis Tool Exposition (SATE) V* [4], the ideal benchmarking dataset should be representative of real, production code, have large amounts of test data to yield statistical significance, and have a known ground truth. Three types of datasets can thus be used:

Synthetic Test Suites A synthetic test suite includes generated code with limited complexity and precisely placed vulnerabilities and defects in it. For the C/C++ track of *SATE V*, this is covered by the *Juliet Test Suite 1.3 for C/C++*. The *Juliet Test Suite* was developed specifically for assessing the capabilities of SAST tools and contains 64,099 test cases in 100k files. The suite covers more than 100 different weaknesses, including all major software fault patterns and satisfies the requirements of having ground truth and statistical significance. Therefore, precision and recall metrics are applicable. In Table 4.2, the distribution of software fault pattern clusters is shown. As can be seen, the top clusters are comparable to the ones we found in our preliminary investigation of common C fault patterns for Debian packages in Sect. 3.

Table 4.2: Top SFP clusters in the Juliet Test Suite 1.3 for C/C++

SFP Cluster	Percentage
Memory Access	31.4%
Resource Management	6.4%
Tainted Input	12.1%
Risky Values	32.6%

Production Software Production software offers realism and statistical significance, due to the large number of warnings issued by tools. Production software however suffers from the lack of ground truth, since we can not be sure about all the defects it may contain. Findings thus need to be reviewed manually for correctness and for evaluating the precision of the tools; recall cannot be calculated at all. All this makes this kind of data inappropriate for automated evaluation.

Software with CVEs Real production software with publicly reported vulnerabilities from the *Common Vulnerabilities and Exposures (CVE)* database, which forms a prime source of known defects in production software. On the one-hand side, they are unfortunately still too few to achieve statistical significance for measuring tools' precision. On the other-hand side, the high realism regarding code and the vulnerabilities is an advantage.

In our research, we have primarily used the *Juliet Test Suite 1.3 for C/C++* and additionally conducted a trial on the *Wireshark v1.8* software with 83 known CVEs. For the *Juliet Test Suite*, we though used a subset since we are focusing on C programs on Linux, therefore skipping C++ and Win32 test cases, and we used a single-file test setup, thus omitting multi-module-test cases.

4.3 Evaluation Procedure for Synthetic Test Cases

As pointed out in Sect. 2, benchmarking SAST tools using the *Juliet Test Suite* is not straightforward, since vulnerabilities as reported by the tools need be mapped to the CWEs associated to test cases. This can introduce a bias. Consider for example a tool output like "memory error", mapping this result to CWE-119, or to CWE-120, or to CWE-125 makes a difference for evaluating the recall of the tool and so the overall measured tool performance. This gets even more problematic with respect to the many possible views in the CWE hierarchy. Due to this, we decided for a different approach. Under the assumption, that the probability of a tool reporting a defect at the exact location but for another defect type is very low, we matched reported defects and test case CWEs solely based upon the defect location, thus ignoring the reported defect type.

In more detail, we run the tools on the dataset file by file and log their reports. The tool reports are colleced in a database for further processing. A tool's findings are compared with the *Juliet Test Suite* manifest files, specifying the exact defect location for a test case by file and line number. If there is match between a tool's finding and a test case with respect to the reported location, we count that as a true positive. If a tool reports another location, we count that as false positive. All non-reported locations are treated as either true or false negatives, according to whether the test case represents "good" or "bad" code. Note that we in this way assume, that the totalty of condition negative equates to the lines of code of all test cases. This is in contrast to the original procedure, where the totality of condition negative equates to the sum of test cases. In our case, the probabilities for gaining a true positive or a true negative are though supposed to be more realistic, compared to the 50:50

chance for the default *Juliet Test Suite* procedure [4]. Although, this comes at the cost of penalizing tools which report the right error on a different location, and otherwise rewards tools, which report all locations with a wrong defect type. Using the counts of true/false positives/negatives, the tools' precision, recall and *Matthews correlation coefficient (MCC)* are calculated. We chose MCC in favor of the F-Score, because the former also rewards a higher number of true negatives in contrast to the latter. This penalizes very noisy tools which report every location. The MCC is in general suggested when dealing with imbalanced datasets, as is often the case for vulnerabilities. Measures like accuracy and F-Score do not reflect this imbalance properly.

4.4 Trial on Production Software

For our trial on production software with real CVEs, we only used the most promising tools as evaluated in the first experiment. In general, this part was more difficult to carry out, for several reasons:

- As first tests showed, it was not feasible to analyze the whole *Wireshark* software at once using the sound analyzers like *Frama-C* and *IKOS*. Both tools have the option to analyze single modules, using an alternative entry point for the file to be analyzed. This however causes issues in finding a suitable entry point among all functions in the module. We approached this by calculating the call graphs of all modules and choosing the roots as entry points. We in this way though still miss inter-module defects.
- Except for *CPPcheck* and *Flawfinder*, all tools required include files and preprocessor definitions of the sources under test. For *Wireshark*, we created a JSON compilation database, where each source file has an entry for its definitions and dependencies. All tools, except *AdLint* supported for this compilation database and we therefore also excluded AdLint from the trial.
- Similar to the *Juliet Test Suite*, we used a file-by-file test setup, collecting the tools reports in a database. We also imported the CVE data into the database. The CVE data did not only contain the defect, but also other information, which required us to manually review the matched findings to ensure that only true positives are counted. Due to the number of findings reported by the tools, we only measured the tools' recall, but not the precision.

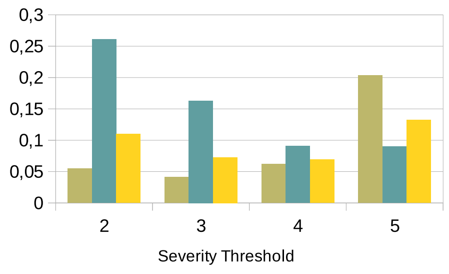
5 Results and Discussion

In this section, we present the results of our evaluation. A replication package is available online[8]. In the evaluation, we were mainly interested in the five following research questions, which will be discussed in the following:

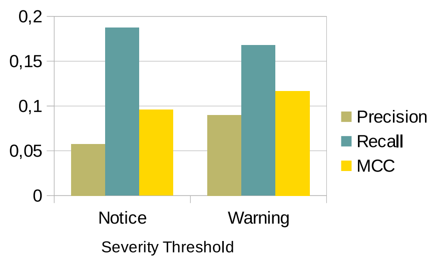
- *What is the effect of the tools' severity threshold parameter on the tools' precision and recall?*

[8] `https://github.com/RohanKrishnamurthy/Automated-bechmarking-of-static-analysis-for-C`

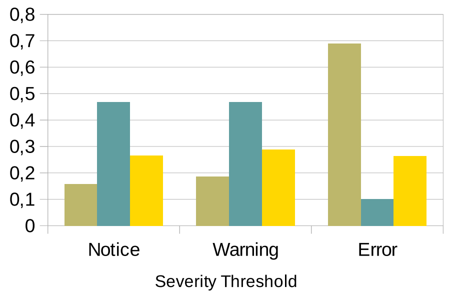
Fig. 5.1: Vulnerability detection metrics with respect to chosen severity threshold



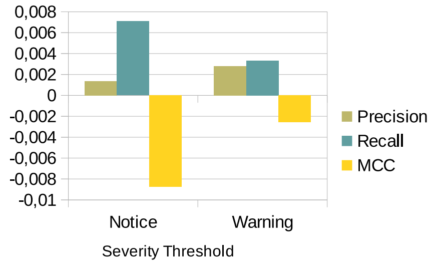
(a) Flawfinder



(b) Clang-Tidy



(c) IKOS



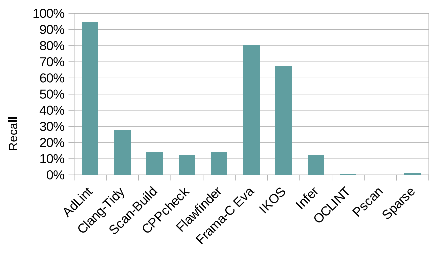
(d) OCLINT

- *Are there differences in the tools' sensitivity with respect to specific vulnerability patterns?*
- *What are the overall tools' accuracies on the Juliet Test Suite?*
- *Are one tool's findings subsumed by another tool's findings, i.e., do the tools' findings overlap?*
- *What is the tools' precision on the Wireshark production software?*

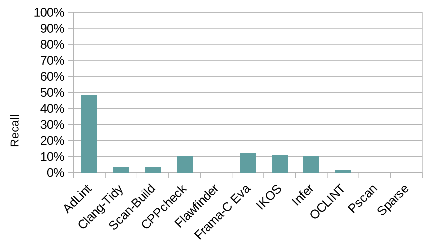
5.1 Effect of Severity Thresholds

The SAST tools *Flawfinder*, *Clang-Tidy*, *IKOS*, and *OCLINT* support a severity threshold parameter, i.e., a minimum severity level upon which found vulnerabilities are reported by the respective tool. In Fig. 5.1, the influence on the tools' precision, recall, and MCC are shown for the *Juliet Test Suite*. As expected, with increasing severity thresholds, most tools also showed an increase in their precision, while the their recall decreased. Overall the accuracy, as measured by *MCC*, did though not change substantially. Due to this, we have chosen the *Warning*-level as minimum severity in all further experiments. For *Flawfinder* we chose the highest level (5) to achieve optimal accuracy.

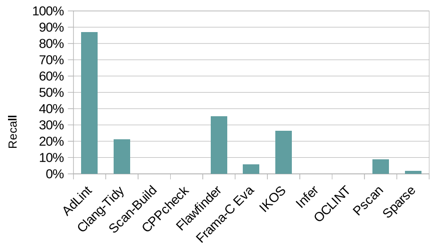
Fig. 5.2: Recall on the Juliet Test Suite for top SFP clusters



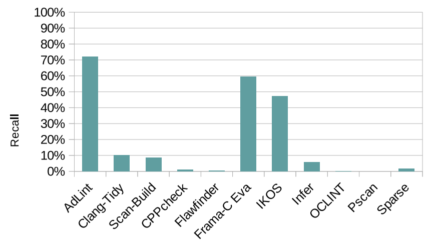
(a) Memory Access



(b) Resource Management



(c) Tainted Input



(d) Risky Values

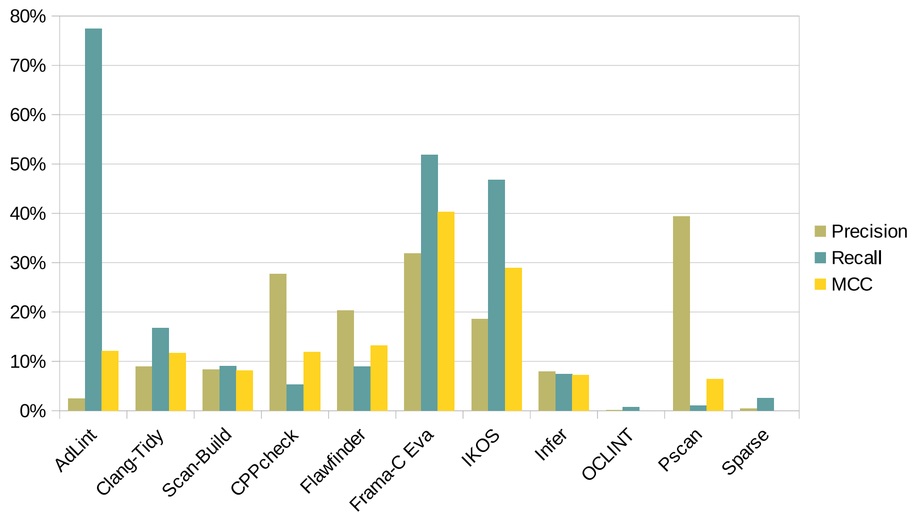
5.2 Tools' Recall on SFP Clusters

Next, we were interested in whether some tools provided a better sensitivity for vulnerabilities of certain types of vulnerabilities. In Fig. 5.2, we show the recall of the 11 SAST tools on the top four SFP clusters, as introduced in Sect. 3. Apparently, the sound tools *Frama-C* and *IKOS* excel particularly for the clusters memory access and risky values, while having only an average recall for vulnerabilities related to resource management and tainted input. In general, with the exception of *AdLint*, none of the tools covers all the clusters with the same sensitivity. Furthermore, some tools only provide a good sensitivity for vulnerabilities of a certain type, such as *Pscan* for the cluster tainted input. For an explanation of the good recall of *AdLint*, see the following section.

5.3 Overall SAST Tool Accuracy

In Fig. 5.3, we present the measured precision, recall, and MCCs for all 11 tested SAST tools on the *Juliet Test Suite*. Apparently, there are differences in the tools' accuracy. With respect to recall, some of the tools found nearly nothing (*OCLINT*), while others found more than 50% of all tested vulnerabilities (*AdLint*, *Frama-C*). Regarding precision, there are tools where almost every second finding is a true positive (*Pscan*), and others, where a developer would have to go through hundred of findings to have only one true positive (*OCLINT*). In particular two tools show a gap between precision and recall. *PScan* seems to be

Fig. 5.3: Precision, recall and MCC metrics on the Juliet Test Suite



highly specialized on certain types of vulnerabilities and therefore has only a low recall but a high precision in that, what it finds. The tool *AdLint* shows a good recall, but this comes at the cost of a bad precision. In fact, *AdLint* produced warnings for about 1/8 of the test suite, resulting in 800k warnings in total. Note that a tool reporting just every line of code as vulnerability gains 100% recall. Finally, we can observe, that the different characteristics of precision and recall lead to nearly aligned MCCs for some tools. Thus, the accuracy of *AdLint*, *Clang-Tidy*, *CPPcheck*, and *Flawfinder* are almost on par, while the sound tools *Frama-C* and *IKOS* are outstanding. This result is in line with the findings in related work [1, 3, 8, 9], in which *Frama-C* ranked similarly. We have though no comparison for *IKOS*, as it was not included in those evaluations. Other tools with reported good accuracy were *Clang* and *CppCheck*. We can confirm a high precision of *CppCheck* and a fair recall of *Clang* in our experiments.

5.4 SAST Tool Overlap

We have also analyzed, if vulnerabilities reported by one tool are reported by another tool as well. Basically, we were asking if there exists one superior tool or if, instead, it makes sense to use more than one SAST tool. In Table 5.1, we denote the measured overlap of the vulnerabilities reported for the *Juliet Test Suite* for the four tools with the most findings, i.e., *Clang-Tidy*, *Flawfinder*, *Frama-C*, and *IKOS* and several other tools. For example, *Clang-Tidy* found 75.6% of the vulnerabilities that were also reported by *Clang Scan-Build* and *Flawfinder* found all of the vulnerabilities found by *Pscan*. This last example may indicate, that in some cases, using one SAST would make the use of another SAST tool redundant. However, considering *Flawfinder* and *Pscan*, note that the

Table 5.1: Overlap of reported vulnerabilities for subset of tested SAST tools

	Clang-Tidy	CppCheck	Infer	Pscan	Scan-Build	Frama-C
Clang-Tidy		47.5%	40.2%	0.0%	75.6%	20.5%
Flawfinder	9.5%	6.4%	1.7%	100.0%	1.5%	4.3%
Frama-C	20.5%	38.0%	13.2%	0.0%	16.5%	
IKOS	17.5%	36.6%	12.7%	50.0%	8.9%	57.9%

latter tool provides better precision than the fromer one and therefore still may the option of choice. In general, we can observe that there exists no one single superior open source SAST tool for the C programming language.

5.5 Trial on Production Software

The results for the trial of the six tools *Clang-Tidy*, *Cppcheck*, *Flawfinder*, *Frama-C*, *Infer*, and *IKOS* on the *Wireshark v1.8* production software are shown in Table 5.2. As mentioned in Sect. 4, we decided to only measure the tools' recall but not precision, due to the rather large number of reported warnings which would need to be checked manually for being true or false positives otherwise. It is apparent, that all of the six tools have much more difficulties in finding real vulnerabilities in production software, compared to finding vulnerabilities in the synthetic *Juliet Test Suite*. Alltogether, only 17 out of the 83 CVEs included in the *Wireshark v1.8* dataset [4] were found. There was no overlap, each tool reported different vulnerabilities. *CPPcheck* and *Infer* surprisingly did not find any of the CVEs. *IKOS* produced so many warnings because of unknown side effects of included code, that we removed it from the trial. Also, the other sound tool *Frama-C* did not perform as good as on the synthetic data.

Table 5.2: Recall results for the SARD-94 test cases (*Wireshark v1.8*)

Tool	Findings	TP	Recall
Clang-Tidy	1,733	1	1.2%
CppCheck	74	0	0.0%
Flawfinder	2,256	9	10.8%
Frama-C	10,273	7	8.4%
Infer	3,022	0	0.0%
IKOS	48,130	-	-

6 Threats to Validity

In this section, we discuss the threats to the validity for our research.

Tool Selection First to mention, a bias is introduced through the selection of SAST tools. Since we did not consider commercial tools and may have missed some novel academic proof-of-concept implementations, we can not provide a complete picture of the capabilities of SAST tools. Nonetheless, we believe that we have covered an exhaustive selection of mature and maintained open source tools. Another threat relates to the chosen version for the tools. Since we used the latest stable version available for Ubuntu 18.04 LTS, some versions are already outdated. This is especially true for the Clang-tools, which we tested for their version 4. This is a general problem, as many tools are contineously updated and improved. Establishing a ongoing SAST tools benchmark facility with automated evaluation and reporting could help to cope with this problem.

Test Cases Since our focus has been to benchmark open source SAST tools, which often did not support the Windows Platform, we omitted the Windows test cases from the *Juliet Test Suite*. Also, since not all tools fully suppor C++ code, we omitted all C++ test cases. Note that this is no serious flaw of our evaluation, but should be considered before one draws any conclusions on the the tools for Windows-targeted code and/or C++. Also, as already mentioned, the realism of synthetic test code as used in the *Juliet Test Suite* is not high, and does probably not reflect the accuracy of the tools on production code. There is no easy solution for this, due to the tradeoff when benchmarking SAST tools between realism and statistical significance and/or unknown ground truth.

Tool Coverage We have chosen to test all 11 open source SAST tools on all test cases – regardless of the respective vulnerability pattern. Our results may therefore differ from other benchmarks. For instance, in the *SAMATE evaluation* [4], they used a special measure called applicable recall, where the supposed tools' coverage on vulnerability patterns is taken into account, which therefore enhances the recall significantly. We have decided against such a measure, in order to make the results more comparable and provide a comprehensible result.

Procedure As discussed in Sect. 4, our method of counting true positives using the vulnerabilities' locations is based on the assumption that it is unlikely, that a SAST tool would report the correct location with a finds a wrong vulnerability. A very noisy tool that reports almost every thus achieves low precision but high recall, since we do not check if the tool reports the expected vulnerability types. Averaging precision and recall by the F-Score, such a tool would gain a reasonable ranking. To compensate for this, we use the MCC instead of the F-Score. Our approach also penalizes tools, which report the expected vulnerabilities at a different code location., and tools, which find other defects than the expected ones. This is in particular problematic for sound tools like *Frama-C* and *IKOS*, as they halt analysis after finding a defect, and may thus explain their unexpected low precision of under 50% in the evaluation as well as their low recall for vulnerability patterns related to memory access and risky values. However, these are general problems and known issues of the *Juliet Test Suite*.

7 Conclusion

In this paper, we have reported on our evaluation of 11 open source SAST tools for the C programming language using synthetic test cases of the *Juliet Test Suite 1.3 for C/C++* and the *Wireshark v1.8* production software with real CVEs. Starting with a preliminary investigation on fault patterns and vulnerabilities in the Debian Linux distribution, we found that half of all vulnerabilities are coding errors related to memory management, followed by errors related to resource management, tainted input, and risky values. We therefore decided to use the *Juliet Test Suite* as a dataset for our evaluation, as it covers these weaknesses with a similar weighting. We confirmed findings of previous work on benchmaring SAST tools for C, using other test suites and evaluation procedures. programming language. For example, we can confirm that *Frama-C* is one of the best open source SAST available, even though it is not specifically focused on security. Another sound tool *IKOS*, which has not yet been evaluated on the *Juliet Test Suite* before, performed similar and ranks second in our benchmark. Howeverm we have also found that no superior tool exists and the tools accuracy in detecting real vulnerabilities in production software can still be improved.

Future benchmarks would benefit from a new or overhauled test suite, which addresses the known problems we have also identified, like the absence of unintended defects and vulnerabilities or better test annotations for matching reported with expected vulnerabilities. It would be of general interest to develop a benchmark suite as open standard, where tool developers and security experts can join to promote realistic SAST tool benchmarking. To cope with the rapid outdating of benchmarks and tools, establishing a regular and an automated evaluation of SAST tools would be a valuable contribution.

References

1. Arusoaie, A., Ciobâcă, Ş., Craciun, V., Gavrilut, D., Lucanu, D.: A comparison of open-source static analysis tools for vulnerability detection in C/C++ code. In: SYNASC. pp. 161–168. IEEE Computer Society (2017)
2. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: NFM 2015. LNCS, vol. 9058, pp. 3–11. Springer (2015)
3. Chatzieleftheriou, G., Katsaros, P.: Test-driving static analysis tools in search of C code vulnerabilities. In: COMPSAC Workshops 2011. pp. 96–103. IEEE Computer Society (2011)

4. Delaitre, A., Stivalet, B., Black, P.E., Okun, V., Ribeiro, A., Cohen, T.S.: Sate v report: Ten years of static analysis tool expositions. Tech. Rep. NIST-SP-500-326, NIST (2018), https://doi.org/10.6028/NIST.SP.500-326
5. Goseva-Popstojanova, K., Perhinschi, A.: On the capability of static code analysis to detect security vulnerabilities. Inf. Softw. Technol. **68**, 18–33 (2015)
6. Herter, J., Kästner, D., Mallon, C., Wilhelm, R.: Benchmarking static code analyzers. Reliab. Eng. Syst. Saf. **188**, 336–346 (2019)
7. Kuhn, R., Raunak, M.S., Kacker, R.: Can reducing faults prevent vulnerabilities? IEEE Computer **51**(7), 82–85 (2018)
8. Lu, B., Dong, W., Yin, L., Zhang, L.: Evaluating and integrating diverse bug finders for effective program analysis. In: SATE. Lecture Notes in Computer Science, vol. 11293, pp. 51–67. Springer (2018)
9. Moerman, J., Smetsers, S., Schoolderman, M.: Evaluating the performance of open source static analysis tools. Bachelor thesis, Radboud University, The Netherlands **24** (2018)
10. NAS-CAS: On analyzing static analysis tools. Technical report, National Security Agency Center for Assured Software (2017), https://media.blackhat.com/bh-us-11/Willis/BH_US_11_WillisBritton_Analyzing_Static_Analysis_Tools_WP.pdf
11. Prause, C., Gerlich, R., Gerlich, R.: Evaluating automated software verification tools. In: ICST 2018. pp. 343–353. IEEE Computer Society (2018)
12. Seacord, R.C.: The CERT® C Coding Standard, Second Edition: 98 Rules for Developing Safe, Reliable, and Secure Systems. Addison-Wesley Professional, 2nd edn. (2014)
13. Shiraishi, S., Mohan, V., Marimuthu, H.: Test suites for benchmarks of static analysis tools. In: ISSRE Workshops 2015. pp. 12–15. IEEE Computer Society (2015)