Technical Report
DLR-IB-DW-JE-2020-16

Evaluation of Open Source Static Analysis Security Testing (SAST) Tools for C

Christoph Gentsch

DLR German Aerospace Center
Institute of Data Science
IT-Security
Jena

DLR **Deutsches Zentrum für Luft- und Raumfahrt**
German Aerospace Center

DLR German Aerospace Center
Institute of Data Science
IT-Security
Dr. Robert Axmann
Mälzerstraße 3
07743 Jena
Germany
Tel: +49 3641 30960-0
Fax: +49 3641 30960-0
Web: https://dlr.de/dw/en

Christoph Gentsch
Tel: +49 3641 30960-131
Fax: +49 3641 30960-131
Mail: christoph.gentsch@dlr.de

Document Identification:

Report number	DLR-IB-DW-JE-2020-16
Title	Evaluation of Open Source Static Analysis Security Testing (SAST) Tools for C
Subject	Technical Report
Author(s)	Christoph Gentsch
Filename	report.tex
Last saved on	30th January 2020

Document History:

Version 0.1	Initial draft	05.02.2019
Version 0.2	Revised and extended	25.07.2019
Version 1.0	Release	09.01.2020

Contents

1 Introduction

Static program analysis in general has the goal to make propositions about the behavior of programs by the analysis of their source code without executing it. On the contrary dynamic program analysis follow other approaches e.g. by executing the code and observing its behavior directly. The former approach tries to make propositions about *any* possible path in the source code, and thus does not need any test code, to run - and the latter approach is dependent on this tests and thus always suffers an incompleteness, since no test code can exhaustively test all paths with every possible input. However, static analysis also has its limitations in making certain propositions about programs in general is regarded as an undecidable problem as has been already proven by Turing (1937). Nonetheless, it is still possible to give valid answers by using approximations, but the consequence is, that a static analysis will always be incomplete or unsound, i.e. miss critical runtime-errors or report false positives. The application of static analysis is manifold e.g. in compiler optimization, test generation, or safety and security testing. This work is about static analysis for security testing, furthermore abbreviated as "SAST".

The goal of SAST-tools is to help developers coding software in a more secure fashion, by pointing early at suspicious constructs, insecure API usage or dangerous run-time errors. In the last years, the market for SAST-tools has expanded, since software security draws more and more attention. There exists a variety of tools whether commercial or open source claim more or less to do a "security analysis". Some of these tools are basically syntax checkers, which apply context insensitive pattern matching to check the 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 questionable, how good they are at finding e.g. deep nested memory errors, or concurrency problems. Then there exists more sophisticated, semantic analyzers, which focus more on bug finding than on compliance checking, but can make no claims on the complete absence of run-time errors. Last, there are so called "sound" semantic analyzers, which mostly rely on abstract interpretation of the source code and promise to make "sound" propositions. What this means in practice can 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 w.r.t. the mathematical model 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." (Calcagno et al., 2015).

As the variety of tools and techniques can be confusing for a developer, it is apparent that an objective evaluation of their abilities is desirable. The goal of this evaluation is to measure, to which extent open source SAST-tools can find vulnerabilities in C code. The focus on C has several reasons: at first, C is still one of the most important programming languages around. Approximately one third of all Linux packages are mainly written in C; also when it comes to embedded or IoT devices, C is the language of choice (Eclipse Foundation, 2017). In the field of (in-)security, C also stands out: we can show, that a small fraction of C programs is responsible for 50% of all vulnerabilities in our Linux dataset (Table 3.1). There exists several tools for analyzing code and datasets to analyze, e.g. the Juliet Test Suite for C/C++ from SAMATE - the Software Assurance Metrics and Tool Evaluation (SAMATE) team of the U.S. National Institute of Standards and Technology (NIST)[2] - which we have used in our evaluation. The restriction to evaluate open source tools exclusively comes from the requirement, to have for anyone reproducible results, as is common in the scientific field. Additionally, as scientific and/or open source developers usually use open source development tools themselves, this is also a test of the ability of the scientific and/or open source community to produce secure software.

The question we want to answer is: *if I, as a developer, would use this tool - and only this - what can I expect of the security of my code after it has been analyzed (and corrected)*? 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 safe..". Apparently, this assumption can only hold if the tool covers *all* kinds of defects which would threaten the security of the application. Therefore, we need test cases, which reflect the most common vulnerabilities of the past, weighted for their observed frequency, or "prevalence", and test every tool with them. This takes away the burden from the developer, to study the capabilities of each tool, and decide, if he needs the detection of vulnerability X or vulnerability Y or both. This approach doesn't bias the calculated precision: e.g. a tool which only detects format string vulnerabilities - but with no false positives - would nonetheless gain 100% precision. But on the other hand, it would gain 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" and therefore the inability to make up an insurance alone. Our goal is to ultimately have a single measure for each tool, which tells a developer, how "safe" he is, when using this

[1] from the Frama-C website (www.frama-c.com)
[2] samate.nist.gov

tool. This is why we chose to test each tool with all test cases, regardless of the self-stated defect coverage of the tools.

2 Previous Benchmarks and Test Suites

In the past years, several works have been published on the evaluation of static analysis tools for the C language. The undermining fact from the publications from previous years is that some tools under test have been developed further and some may have been terminated and are not available anymore. One such publication is from Chatzieleftheriou and Katsaros (2011) where they tested CppCheck, Splint, UNO, Frama-C and two other commercial tools with their own test suite. The two commercial tools ranked best regarding their F-Score of 0.85, closely followed by Frama-C (0.8) and UNO (0.7). CppCheck and Splint ranked at the lower end (<0.6). NSA-CAS (2011) published the first version of their Juliet Test Suite for C/C++, which covers 116 different CWEs in over 45k test cases, where each one has a "good" and "bad" function, together with the labels for the CWEs of each test case. The decision to have those "good" and "bad" functions should have solved the problem of analysis tools that will probably not mark exactly the "intended" line of code in the test case as faulty, but instead they could point somewhere within a "bad" function to have a "hit". For example in listing 2.1 a tool could mark line 2 or line 3 as erroneous.

```
1 ...
2 for (i = 0; i <= sizeof(dst)/sizeof(int); i++) {
3   dst[i] = src[i];
4 }
5 ...
```

Listing 2.1: **Example Code for a Off-By-One Error**

As half of the test cases consists of those "bad" functions, this implies that the measured precision is only a precision for 50% prevalence - which means, there is a 50% chance to just *guess* the right location and would then only gain 50% Precision. It is questionable if this measured Precision reflects the experience of a developer using that tool on production software where the ratio of good code vs vulnerabilities is completely different. To address this, NSA-CAS (2011) suggested another metric, the Discrimination Rate, wherein the findings counted as True Positive which the tool reported in the "bad" function - but not in the "good" function. This penalizes just guessing tools which then would only gain

0% Discrimination Rate. To prevent that a tool also gets a high Recall by just guessing, the evaluator shall compare the type of error (CWE) the tool states for a certain code location to the CWE under test within the test case. Only if both matches, the CWE of the test case and the CWE the tool reported as a finding, this shall be counted as a True Positive. As this is a feasible procedure in theory, the practical application is not as straight forward, especially when testing open source tools. Those often lack the option to report CWEs, or just report a hand full of very general CWE-classes like "CWE-119" (Improper Restriction of Operations within the Bounds of a Memory Buffer) whereas the Juliet Test Suite asks for very specific CWE-variants like "CWE-126" (Buffer Over-read). The result is, that a lot of true positive findings from the tools would not count since the CWEs don't match exactly, which would lead to a very bad Recall for those tools on the Juliet Test Suite. One solution to this is proposed by Goseva-Popstojanovaa and Perhinschi (2015) who also used the Juliet Test Suite for C/C++ with 3 anonymous commercial tools. They had carried out a "fuzzy" match by using the CWE hierarchy to match CWEs that are closely related to each other in the hierarchy, but had a different ID. As this would solve the example with "CWE-119" vs "CWE-126" since they are on the same path in the CWE-hierarchy, it would not for others. Consider the following example from the Juliet Test Suite 1.3 for C/C++:

```
1 ...
2 data = -1;
3 /* Assume we want to allocate a relatively small buffer */
4 if (data < 100)
5 {
6   /* POTENTIAL FLAW: malloc() takes a size_t (unsigned int) as input and
       therefore if it is negative, the conversion will cause malloc() to
       allocate a very large amount of data or fail */
7   char * dataBuffer = (char *)malloc(data);
8 ...
```

Listing 2.2: **Example Code from the Juliet Test Suite for C/C++**

Here a tool could rightly report a "CWE-686" (Function Call With Incorrect Argument Type) at line 7 since "-1" is definitely incorrect for *malloc* which expects unsigned integers. Otherwise, a "CWE-131" (Incorrect Calculation of Buffer Size) on line 7 would also be feasible. But what the Juliet Test Suite expects in this case is "CWE-195" (Signed To Unsigned Conversion Error)[1]. It is obvious, that the solution from Goseva-Popstojanovaa and Perhinschi (2015) can not help here, since those CWEs are too far away from each other in the CWE-hierarchy. One of the reason for those difficulties lies in the complexity of the CWE-hierarchy itself. With more than 800 weaknesses is it very fine-grained and offers more than 30 different "Views" (meaning hierarchies) which completely change the relation between the CWEs to better reflect the different perspectives one can have on

[1] the example code is taken from the Juliet Test Case file "CWE195_Signed_to_Unsigned_Conversion_Error_-_negative_malloc_18.c"

those weaknesses. In this jungle it is hard for test suite makers and tool developers to choose the right CWEs to label their test cases or tool findings. A pragmatic approach to solve this, is to just not match any CVEs, but instead design the test cases in a way, that any reader (and any tool) can only find the flaw at one certain location in the test case, which makes it less important to also check the type of the error on that location. This is what Shiraishi et al. (2015) from Toyota published as the so called "ITC Benchmark". It has the advantage over the Juliet Suite as being more easy to understand and also very easy to carry out, since one need not check in which function an error has been found and no CWE mapping and matching has to be done. But this "one flaw - one line" comes at the cost of relatively easy test cases - which can lead to a uncertain measurement. Therefore, some tested tools in Shiraishi et al. (2015) gain a perfect result (100%) in some of the categories, suggesting that they are equally accurate - a false conclusion since this 100% act like a "cutoff" where probably higher results are just not measurable anymore. Other problems of the ITC Benchmark include several unintended defects, several wrong and/or missing error markers and the selection of test cases itself and their weighting as not being representative for commonly accepted defect types as Herter et al. (2017) found in their review of the suite. In general the ITC Benchmark is mainly designed to check for safety issues with the consequence, that tests regarding input validation, path traversal, code injection or OS command injection are completely missing. These are all issues which can only be found by a tool with "taint analysis" capabilities, which unfortunately will not be tested by this suite. All this makes the ITC Benchmark non optimal for the evaluation of SAST-tools.

Nonetheless, Arusoaie et al. (2017) used the ITC Benchmark to compare most of the open source tools we were also interested in. In their results, Clang and Frama-C are leading the field, closely followed by CppCheck. Lu et al. (2018) compare CppCheck, CBMC, Frama-C, Clang and a commercial tool. They used the Juliet Test Suite, but with an unspecified algorithm for how they actually evaluated the findings to calculate precision and recall. Despite the commercial tool, the best F-Scores reached CppCheck (0.34), Frama-C (0.3) and Clang (0.3). Moerman (2018) in his bachelor thesis also tested several tools for C, notably: Splint, Cppcheck, Frama-C, Infer and Clang. He was using the test suite from Chatzieleftheriou and Katsaros (2011) and an own reworked version of this. Clang and Frama-C lead again with an F-Score>0.9, followed by Infer (0.88) and Cppcheck (0.78) on the Chatzieleftheriou and Katsaros (2011)-Test Suite. The high scores, again, may indicate, that the test suite could be too easy for the tools to give a good measure. His reworked test suite gave similar results, while lacking of some important test cases for format string vulnerabilities or concurrency errors.

There exists also papers on testing single tools like Stikkelorum (2016), who is mainly testing Infer, while using some particular test cases from the Juliet Test Suite for Java and C/C++ which Infer claims to find. A comparison is done to FindBugs and Coverity for those

CWEs (690, 401, 476, 761), where Infer leads the field with nearly perfect results (>0.9 F-Score) for CWE-476 on Java and C.

3 Preliminary investigation of C-related fault patterns

As a starting point, we wanted to know which C-related flaws caused the most vulnerabilities found in production software within the last two years. Since Unix/Linux is widely used on internet servers (69% market share (W3Techs, 2019)) but also desktop computers, mobile devices (as base of Android OS) and in the emerging field of IoT (internet of things, market share 81% (Eclipse Foundation, 2017)), we decided to use a complete Unix*oid operating system as the "production software" for our survey.

Traditionally, a Linux based OS not only consist of the kernel, but is also distributed with thousands of software packages of all sort, ranging from desktop applications like OpenOffice or Firefox to e-mail and web server software like the Apache HTTP-server, which then are all part of the so called "distribution". This makes a Linux-distribution a worthy study object, since it contains a mature cross section of productive open source software. Additionally, most major Linux distributions have their own security tracker, where they collect information on security related bugs and incidents for all packages, and also deliver security patches for them. One major and mature Linux distribution is Debian GNU/Linux[1], which we chose.

As a first step we downloaded all the packages from the current stable release (Debian 9). Then we gathered all the occurrences of CVEs[2] in the "changelog" files within the packages. Then we combined the CVE data for each package with the data from the National Vulnerability Database, which maps the CVEs to CWEs[3] and scores them for their severity. After this we collected several metrics of the source code of each package and saved this all together in a database. An overview about the dataset is given in table 3.1. It can be seen, that a) our dataset contains approx. 10% of all software vulnerabilities ever reported to CVE since 1988, and b) 10% of all C packages contain 50% of all CVEs in the

[1] www.debian.org

[2] Common Vulnerabilities and Exposures (CVE) are software vulnerabilities which were officially reported to the public CVE database (cve.mitre.org).

[3] The "Common Weakness Enumeration" is a software security ontology, which identifies and categorizes software weakness *types* (nvd.nist.gov).

Table 3.1: **Some statistics on our Debian vulnerabilities dataset**

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 in C packages since 2017:	1,380
⊘ CVEs per package:	0.4
⊘ CVEs per C-package:	0.7

Table 3.2: **"Top" ten commercial vendors with the most CVEs in the NVD dataset**

Vendor	percentage of CVEs
Microsoft	5.8%
Oracle	5.0%
IBM	4.2%
Apple	4.0%
Cisco	3.7%
Adobe	2.9%
HP	1.6%
Sun	1.2%
Huawei	0.5%
SAP	0.5%
$\sum$	29.3 %

dataset and c) the average count of CVEs in C-packages is higher than the overall average of CVEs per package in Debian. A comparison to other languages is given in figure 3.1.

For validation, we tried to find out the other 90% of the vulnerabilities (if not in Debian) and aggregated the CVEs of the top ten commercial software vendors. Their software is not in the Debian dataset, because most of their products are not free and open source software. In table 3.2 we can see that almost one third of all the vulnerabilities are from this 10 commercial vendors alone. Since we were primarily interested in C language based software we continued to examine only the open source projects within the Debian distribution, where we could easily verify the programming language used, due to the open source nature of those projects.

Figure 3.1: **Average CVEs per package by main programming language**

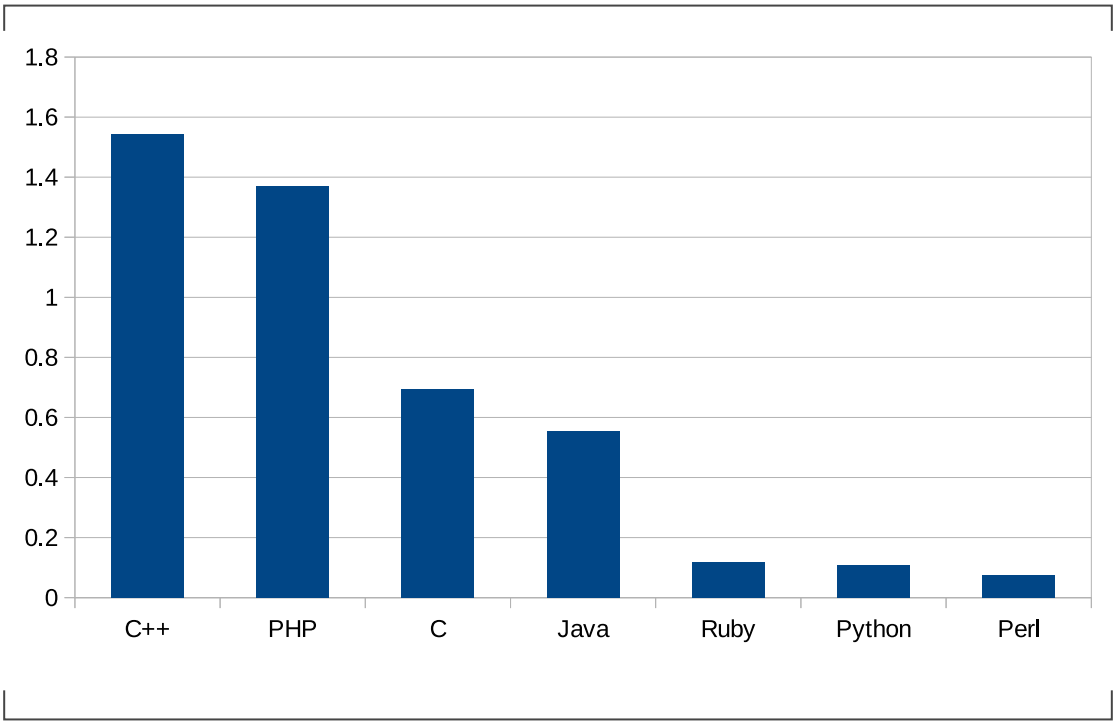


Table 3.3: **Top 10 vulnerabilities found in Debian packages with C as main language**

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%

Next we wanted to know which type of vulnerabilities were found in those packages, whose main language (meaning: with a majority of source files) is C. To be sure, that we don't make assumptions about issues which happened in the past - but don't really occur anymore in the present - we only considered CVEs which were reported within the last 2 years (2017-2018). Table 3.3 shows the Top 10 list of CWEs for C packages, according to the summed up NVD severity scores for all CVEs. Apparently, the "buffer overflow" is still the leading flaw in C-programs, followed by several other memory management-related issues. Nonetheless, to have a better overview, we also mapped the C-related CWEs to the "Software Fault Patterns (SFP) Clusters", also by MITRE [4]. As shown in Table 3.4, the top fault cluster is "Memory Access" with a fraction of approx. 50% of all vulnerabilities found in the C-packages. Members of this cluster are "Faulty Buffer Access", "Faulty Pointer Use", "Faulty String Expansion", "Improper NULL Termination", and "Incorrect Buffer Length Computation". The second important fault cluster "Resource Management" with a fraction of 12% is also often related to memory management like "use after free()", but also to things like missing release of file handles or uncontrolled recursion. The cluster "Tainted Input" with approx. 10% is mostly about missing neutralization of special elements from user input. Finally, "Risky Values" consists of issues related to integer overflows or wraparounds, incorrect type conversions or divides by zero. Altogether it seems like most vulnerabilities come from very C-specific implementation errors which should be detectable. In the next step we tested static analysis tools, which promise to do exactly this. This result is on par with the findings from Kuhn et al. (2018).

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

Table 3.4: **Top Software Fault Pattern clusters found in C-written Debian packages**

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

4 Tools Evaluation Method

4.1 Tool Selection

The tools were selected from a curated list of static analysis tools (Wikipedia, 2019) and the Debian Security Audit Project guide (Debian, 2019). Some of the tools mentioned there have not been tested because they are out of scope of this survey, like model checkers (Blast, CPAchecker), or because they are pure linters (Lint, cpplint) or are no longer maintained (Splint, RATS, ITS4). Besides, three sound tools have been tested: Infer, IKOS and Frama-C. The latter has also been evaluated in SATE V, but in a separate category for sound tools, which makes it difficult to compare the results with the other tools. For all of the tools, we chose the latest stable version that was available in the standard Ubuntu Software Repository for Ubuntu 18.04 LTS in December 2018. Infer and IKOS were not available in Ubuntu and thus had to be downloaded directly from the vendor. All scripts written to perform the evaluation can be found in our GitLab-Repository[1].

[1] https://gitlab.dlr.de/dw-its-sst/sastevaluation
[2] http://adlint.sourceforge.net
[3] http://clang.llvm.org/extra/clang-tidy/index.html
[4] http://clang-analyzer.llvm.org/scan-build.html
[5] http://cppcheck.sourceforge.net
[6] http://dwheeler.com/flawfinder
[7] http://frama-c.com
[8] http://github.com/NASA-SW-VnV/ikos
[9] http://fbinfer.com
[10] http://oclint.org
[11] http://deployingradius.com/pscan
[12] http://sparse.wiki.kernel.org/index.php/Main_Page

Table 4.1: **In our study evaluated static analysis tools**

Tool	Version	Arguments
AdLint[2]	3.2	
Clang-Tidy[3]	4.0	-checks=-*,clang-analyzer-*,cert-*
Clang Scan-Build[4]	4.0	
CppCheck[5]	1.72	
Flawfinder[6]	1.31	
Frama-C Eva[7]	17 (Chlorine)	
IKOS[8]	2.0	-d=var-pack-dbm
Infer[9]	0.15	
OCLint[10]	0.13.1	
Pscan[11]	1.2-9	
Sparse[12]	0.5.0	-Wsparse-all

4.2 Datasets

In the report of the Static Analysis Tool Exposition (SATE) V from SAMATE, they point out, that the ideal dataset should: 1) be representative of real, existing code, 2) have large amounts of test data to yield statistical significance, and 3) have a known ground truth. To ensure this, they propose three types of datasets for effectively evaluating static analysis tools:

Synthetic Test Suites This is generated code with limited complexity and precisely placed flaws in it. For the C track, the dataset is the *Juliet Test Suite 1.3 for C/C++* originally created by the U.S. National Security Agency's (NSA) Center for Assured Software (CAS) and developed specifically for assessing the capabilities of static analysis tools. It contains 64,099 test cases and more than 100,000 files. An example test case is given in listing 4.1. The suite covers 118 different weaknesses including all major software fault pattern clusters and satisfies the requirements of having ground truth and statistical significance. Therefore, precision and recall metrics are applicable, but the realism is only limited. In Table 4.2 we examined the distribution of fault clusters within the Juliet Test Cases. The top clusters are identical to the ones in our preliminary investigation of common C fault patterns, although the ratios are not exactly the same.

```c
void CWE126_Buffer_Overread__char_alloca_loop_01_bad()
{
    char * data;
    char * dataBadBuffer = (char *)alloca(50*sizeof(char));
    memset(dataBadBuffer, 'A', 50-1); /* fill with 'A's */
    dataBadBuffer[50-1] = '\0'; /* null terminate */

    /* FLAW: Set data pointer to a small buffer */
    data = dataBadBuffer;
    {
        size_t i, destLen;
        char dest[100];
        memset(dest, 'C', 100-1);
        dest[100-1] = '\0'; /* null terminate */
        destLen = strlen(dest);
        /* POTENTIAL FLAW: using length of the dest where data
         * could be smaller than dest causing buffer overread */
        for (i = 0; i < destLen; i++)
        {
            dest[i] = data[i];
        }
        dest[100-1] = '\0';
        printLine(dest);
    }
}
```

Listing 4.1: **A test case from the Juliet Test Suite for C/C++**

Table 4.2: **Top Software Fault Pattern (SFP) Clusters in the Juliet Test Suite 1.3 for C**

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

Production Software The idea here is to take real production software to have realism and statistical significance, due to the large number of warnings issued by tools. The drawback is, that it lacks ground truth, since we do not know all the bugs that it contains. Therefore, the findings need to be reviewed manually for correctness to measure the precision of the tools; the recall cannot be calculated at all. All this makes this kind of data inappropriate for an automated evaluation.

Software with CVEs This is also real production software, but with publicly reported vulnerabilities from the Common Vulnerabilities and Exposures (CVE) database, which forms a prime source of known defects in production software. Unfortunately, they are still too few to achieve statistical significance to measure precision. The advantage is the high realism regarding the code and the vulnerabilities.

We decided to use the Juliet Test Suite 1.3 for C/C++ in first instance, and then additionally have a try on real software with CVEs. Because of the manual effort, we left the "Production Software" out. Furthermore, we only used a subset of the Juliet dataset for following reasons: a) since we were focusing on C on the Linux platform, we left out the C++ and Win32 test cases b) since the dataset was analyzed file by file in our test setup, we also left out the more complex multi-module-test cases. As test set for the production software with CVEs, we chose Wireshark 1.8 from SAMATE (SARD-94), which contains 83 CVEs.

4.3 Evaluation Procedure on Synthetic Test Cases

As we pointed out in the Related Work (chapter 2), the evaluation procedure which is intended by the Juliet Test Suite is somehow problematic, since it requires one to map the error types of the tool output to CWEs and after this, match them correctly with the CWEs of the test cases. The first task can introduce a bias, since one would map a tool output like "memory error" to CWE-119, or to CWE-120, or to CWE-125 could make a big difference

for evaluating the Recall of the tool and so the overall measured tool performance depends heavily on the quality of such a mapping. The difficulty of the second task of actually matching the reported CWE with the test case has already been discussed in the Related Work (chapter 2). For all this reasons we decided to not to follow the procedure stated by NSA-CAS (2011), but instead match directly on the error location without considering the type of error reported for it, with the assumption, that the probability, that a tool would hit exactly the right location, but reports a false defect type on it, is very low.

The actual evaluation of the tools is carried out as follows: At first, we run the tools on the dataset file by file while logging all output to a log file (one per tool). Then we import all tool log files into a common database, with fields for *tool_id*, *severity*, *file_name* and *line_number*. After this all findings of the tools are matched to the previously imported Juliet "Manifest"-file, where the position of each flaw in the dataset is exactly specified by file and line number, together with the particular CWE. When the tool marks a code location as faulty which is also in the database, then we count this as a true positive, regardless of the kind of error reported. When a tool marks a code location different from the one in the database, we count this as a false positive, again, not taking into account which kind of error was reported. All non-reported locations are treated as true or false negatives. This assumes, that the totality of "Condition Negative" equates to the lines of code of all test cases. This is in contrast to the Juliet intended 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 supposed to be more realistic, compared to the 50:50 chance for the default Juliet procedure. Although, this comes at the cost of penalizing tools which report the right error on a different location, and otherwise rewards tools, which report on every location with a wrong defect type. From this counts of true/false positives/negatives the tools' Precision, Recall and MCC[1] are calculated, according to the formulas given below. We chose the MCC in favor of the F-Score, because the former also rewards high true negatives in contrast to the latter. This penalizes very noisy tools which reports on every location. The MCC is in general suggested when dealing with an imbalanced dataset, where e.g. 99% are negative and 1% positive, as is often the case in the field of vulnerability detection. Accuracy and F-Score do not reflect this imbalance properly and thus are no feasible measures for such classification problems. See a discussion on that in (Chicco, 2017). Other alternatives to the F-Score would be the "Informedness" or "Youden's index" (Youden, 1950), which basically equates to the recall subtracted by the false positive rate, which also rewards high true negatives. Also a similar measure is the "Discrimination Rate" from NSA-CAS (2011) used in the SAMATE. All this metrics is calculated for several severity thresholds to find the best possible MCC. This only applies for: Flawfinder, Clang-Tidy, Sparse, IKOS and OCLINT

[1] Matthews correlation coefficient (Matthews, 1975)

- as most tools give no severity level in their log messages[2]. To better understand the differences of *what* the tools find, we also group all data into the Software Fault Pattern (SFP) Clusters and calculate the Recall of the tools for each cluster. The formulas we use for our metrics are as follows:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

with TP = true positives, TN = true negatives, FP = false positives and FN = false negatives.

4.4 Trial on Production Software

For the experimental audit of the production software with CVEs we only used the most promising tools, as evaluated in the first experiment. Our goal here was to get an idea how the tools perform when used on real software with real vulnerabilities.

In general, this part was more difficult to carry out, for several reasons:

a) as first tests showed, it was not feasible to analyze a whole production software at once with sound analyzers such as Frama-C or IKOS - we broke off a test run with IKOS after 6 days, without results. To address this problem both tools have the option to analyze single modules, by setting an alternative entry point for the file to be analyzed. This leads to the problem how to find a suitable entry point among all functions in the module. We solved this by calculating the call graph (CG) for each module, and then chose all top nodes in the tree, which were not called by others. Of course this added some computational overhead to the whole procedure - for the calculation of the CG, as well as for analyzing

[2] CPPcheck has two severity levels, but only "errors" appear on the Juliet Test Suite

each module multiple times (once for each entry point). The other drawback is, that this way it was very difficult for the tools to find bugs on the inter-module level.

b) except for CPPcheck and Flawfinder, all tools needed the include-files and pre-processor definitions of the sources under test. The same holds for the Juliet dataset, but there the files to be included were very limited, and for each test case the same. For the Wireshark code, each part of the sources had different includes and directives - so we had to get this information somewhere. One solution is to build up a JSON compilation database, where each source file has an entry for its definitions and dependencies. This can be done with the *CMake* build system or tools like *Bear*. Suitably, all tools except AdLint had support for this compile database - this is why we also excluded AdLint from this evaluation.

c) as in the first evaluation, we run the tools file by file on the Wireshark C source files, while collecting all tool messages in a log file which was later imported into the database. We also imported the "Manifest"-file of the test set and matched both tables. Since the "Manifest" of the production software test sets did not only contain *the* flaw itself, but often also something like a stack trace - which leads to the flaw - we had to manually review the matched findings to ensure that only true positives where counted for the *recall*. Since we didn't want to review all the findings of the tools, we could not measure *precision*.

4.5 Tool Usability

Finally, we want to share our experience regarding the usability of the tools. The criteria for this came through our use of the tools itself. Especially the production software dataset - where we had a more real world use of the tools - revealed some handicaps which were not so annoying with the synthetic data before. This was in first instance the need for include-files, as described formerly. Since the production software had significantly more dependencies as the synthetic code, those tools which didn't needed any *includes* at all, were more pleasantly to use. For those tools, which needed includes, we found the support for a compilation database as input very helpful. As formerly described, this database can be conveniently created using a build tool. Finally, we found, that a ranking of the severity of the tool messages is also a useful feature, which helps the developer to focus on the most likely true positives, when going trough the findings.

5 Results and Discussion

In this section, we present the results of our study, as well as some interesting observations and how they impact the approaches we have used. Also, we want to discuss which threats to the validity of our results exist, and thus how our study could be improved.

5.1 Effect of Severity Thresholds

In Figure 5.1 we see the influence on *precision*, *recall* and *MCC* of the chosen minimum severity level of the log messages considered. As expected, for rising severity thresholds, most tools showed an increase in *precision* while the *recall* decreased. Overall the accuracy (as measured by the *MCC*) didn't change dramatically, but was mostly better for the higher thresholds. Therefore, we chose the "warning"-level as minimum severity for all further examinations. For Flawfinder we chose the highest level (5) for optimal accuracy.

5.2 Fault Clusters Recall

In Figure 5.2 we show the recall of the tools on the top four software fault clusters (SFP). It is noticeable, that the sound tools Frama-C and IKOS excel particularly in the clusters "Memory Access" and "Risky Values", while having only average recall in "Resource Management" and "Tainted Input". In general, with the exception of AdLint, none of the tools cover all the clusters equally. Further, some tools even cover only one cluster at all, such as Pscan with "Tainted Input".

Figure 5.1: **Flaw detection metrics depending on the chosen severity threshold on the Juliet Test Suite**

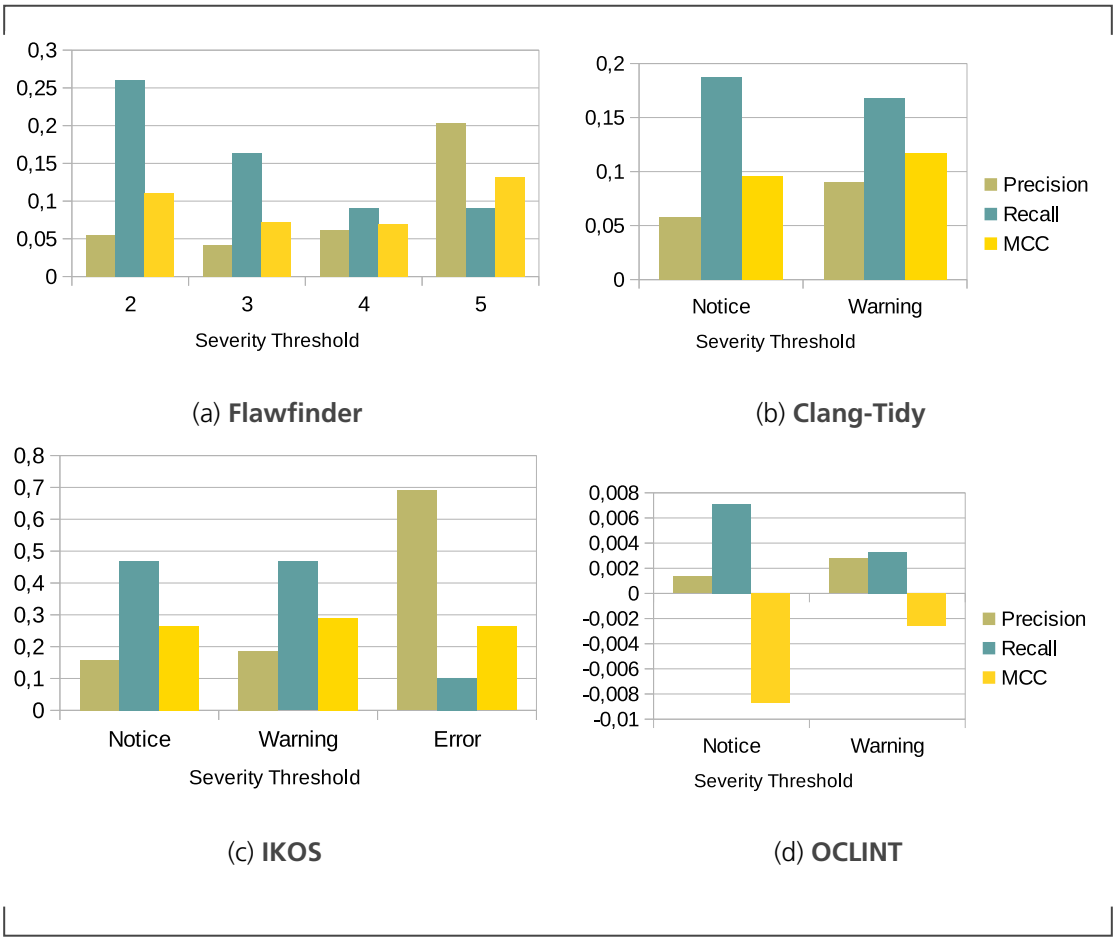
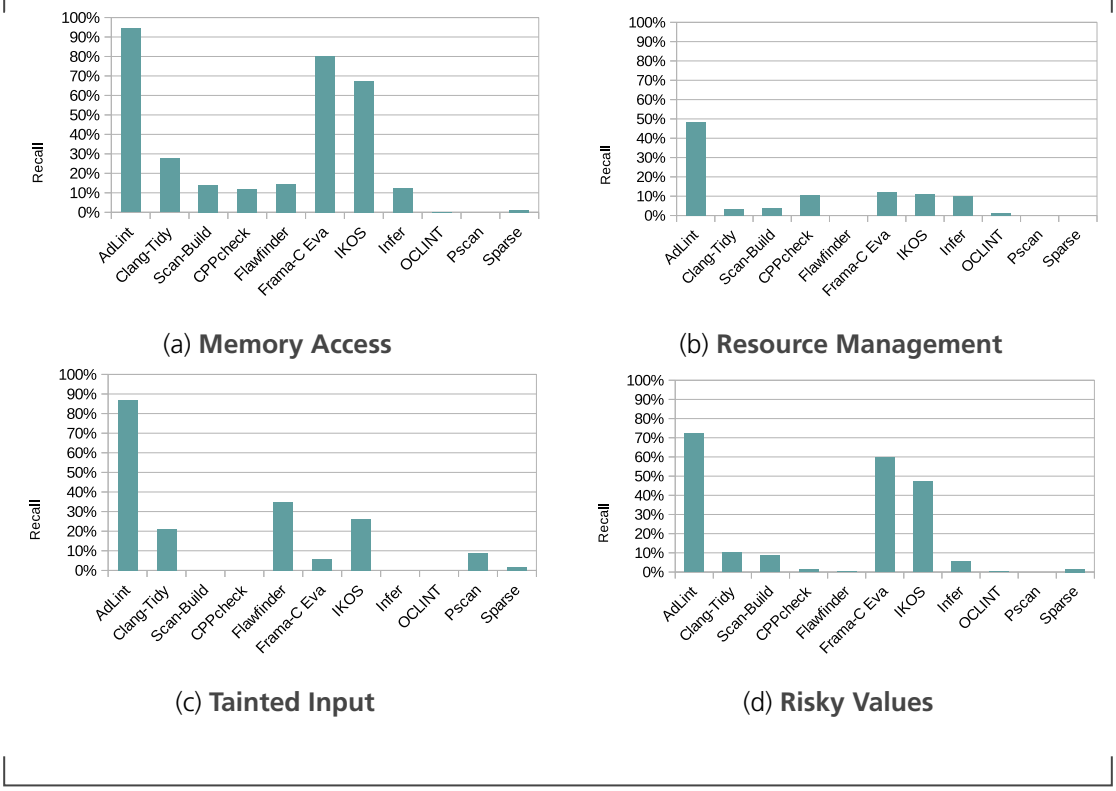


Figure 5.2: **Recall on the Juliet Test Suite for several software fault clusters (SFP)**



(a) **Memory Access**

(b) **Resource Management**

(c) **Tainted Input**

(d) **Risky Values**

5.3 Overall Tool Accuracy

In Figure 5.3 we present the measured *precision*, *recall* and MCCs for all tools on the Juliet Test Suite with the chosen minimum severity level of "warning" (for Flawfinder we chose 5). Apparently, there are big differences in the performance of the tools. With respect to *recall*, some of the tools found nearly nothing (OCLINT), while others found more than 50% of all flaws (AdLint, Frama-C). Regarding the *precision* there are tools where almost every second finding is a true positive (Pscan), and others where the developer would have to go through a hundred of findings to have only one true positive (OCLINT).

Two tools show an extreme gap between *precision* and *recall*. PScan seems to be highly specialized and therefor has only a very low *recall*, but a very high *precision* in that, what it finds. On the other hand the tool AdLint apparently has a very good *recall* in all clusters (67% overall), but this comes at the cost of a very bad *precision* (2.5%). In fact, AdLint produced warnings for about 1/8th of the whole test code, resulting in about 800k warnings in total, which could explain its outstanding *recall* (if a tool highlights just every line of code as "flaw", then it always gains 100% recall). This example points out, why both - precision and recall - are important metrics for the tool performance. Finally we can observe, that the different characteristics of *precision* and *recall* of the tools as seen in Figure 5.3 lead to nearly aligned MCCs for some tools. With respect to this, the performance of AdLint, Clang-Tidy, CPPcheck and Flawfinder are almost equal while Frama-C and IKOS are still outstanding. This is in line with the findings from Chatzieleftheriou and Katsaros (2011), Arusoaie et al. (2017), Lu et al. (2018) and Moerman (2018), in which Frama-C also ranked similarly high. As IKOS has never been tested before we have no comparison for it. Other high ranked tools which appear in the latter three publications were Clang and CppCheck. We can confirm a high precision of CppCheck and a fair recall of Clang in our study. The low precision of Infer is in contrast to the findings from Stikkelorum (2016), but may reflect the problems of sound tools with an imperfect test suite.

Chatzieleftheriou and Katsaros (2011): Frama-C (0.8) and CppCheck and Splint ranked at the lower end (<0.6). Arusoaie et al. (2017) Clang, Frama-C, CppCheck Lu et al. (2018) CppCheck (0.34), Frama-C (0.3) and Clang (0.3). Moerman (2018) Clang and Frama-C lead >0.9, followed by Infer (0.88) and Cppcheck (0.78)

Figure 5.3: **Precision, recall and MCC metrics on the Juliet Test Suite**

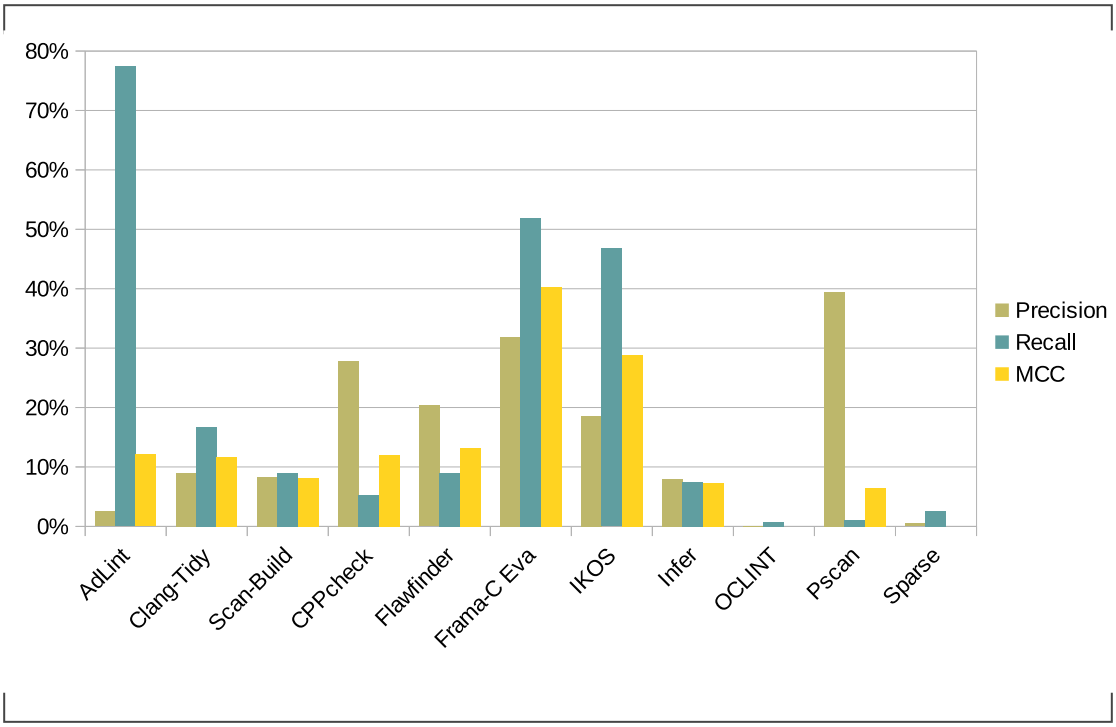


Table 5.1: **Overlap of findings for various tools**

	Clang-T.	CPPcheck	Flawf.	Infer	Pscan	Scan-B.	Frama-C
Clang-Tidy		47.5%	9.5%	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%	4.3%	13.2%	0.0%	16.5%	
IKOS	17.4%	36.6%	8.1%	12.7%	50.0%	8.9%	57.9%

5.4 Tool Overlap

In Table 5.1 we see the overlap of findings of the four tools with the most findings (Clang-Tidy, Flawfinder, Frama-C and IKOS). For example: Clang-Tidy found 75.6% of the issues that also Clang Scan-Build found, Flawfinder found 100% of the findings from Pscan, and so on. This last example shows, that if one is using Flawfinder it would be redundant to also use Pscan. As the former section about the tools recall suggests, this also reflects the differences, in what the tools find.

5.5 Trial on Production Software

In Table 5.2 we present the results of our trial of some selected tools with a production software test suite with CVEs (SARD-94, Wireshark v1.8). It is apparent, that all tools had much more difficulties finding the flaws, as compared to the synthetic data. All together, only 17 out of the 83 CVEs were found. There was no overlap, each tool found different bugs. CPPcheck and Infer did not find any of the CVEs. On the other hand IKOS produced so many warnings because of unknown side effects of included code, that we took it out of evaluation for this trial. Also, the other sound tool Frama-C did not perform as good as on the synthetic data. In fact, Flawfinder outperformed Frama-C with 2 findings more, although this might not be statistical significance.

5.6 Tool Usability

In Table 5.3 we gathered the usability properties of the tools. For instance, as also already mentioned, AdLint has several usability impediments: firstly, it needs all includes for the

Table 5.2: **Results on the SARD-94 Test Cases (Wireshark)**

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	-	-

Table 5.3: **Usability rating of the evaluated static analysis tools**

	no *includes* needed	compile-db support	severities
AdLint	-	-	-
Clang-Tidy	-	+	+
Clang Scan-Build	-	+	-
CPPcheck	+		+
Flawfinder	+		+
Frama-C Eva	-	+	-
IKOS	-	+	+
Infer	-	+	-
OCLint	+		+
Pscan	+		-
Sparse	+		+

source code to be analyzed. Unfortunately, this dependencies can not be delivered via a compilation database, but must be registered in a AdLint-specific configuration file, which then is global for the project to be analyzed. This way it is not possible, to give different dependencies for different parts of the same project. Secondly, AdLint has no severities for its more than 1000 different messages (which include everything from style-checking to buffer-overflows), which makes prioritizing alerts difficult. All this makes AdLint very inconvenient to use. On the other side of the spectrum, there are CPPcheck and Flawfinder for example, which don't need includes to function, but offer message severities - which makes them very convenient to use tools. In the middle field, we have the tools with severities and compilation database support.

5.7 Threats to Validity

In this section we want to discuss the threats to the validity of this study and its results.

Tool selection First to mention, a bias is introduced through the selection of tools for evaluation. Since we didn't consider commercial tools and maybe missed some very new academic proof-of-concepts, we couldn't give a full picture of the true capabilities of static analysis. Nonetheless, we think that we have covered all mature and maintained open source tools (see Related Work (chapter 2)). Another problem related to the tool selection is the version which was chosen to test. Since we used the latest stable version available for Ubuntu 18.04, some versions are already outdated to the writing of this publication. This is especially true for the Clang-tools which we tested in version 4, and are now available at version 7 in Ubuntu 18.04. First tests indicate that especially Scan-Build improved a lot since version 4, as additional checks have been implemented. This "outdating" is a general problem of such benchmarks, as many tools are steadily updated and improved. To establish a continuous SAST-tools benchmark facility with automated evaluation and publishing of reports could help to cope with this issue.

Test cases Since our focus was on open source tools which mainly didn't support the Windows Platform, we left out all Windows-test cases from the Juliet Test Suite. Also, since not all tools fully support C++ code, we left out all C++ test cases. This is no serious flaw of our evaluation, but should be considered before one draws any conclusions on the performance of the tools with Windows targeted code and/or C++. Also, as already mentioned, the realism of synthetic test code as we used with the Juliet Test Suite is not very high, and does probably not reflect the performance of the tools on production code. There is no easy solution for this, because - as already stated in the Procedure section - there is always a trade off between realism and statistical significance and/or unknown ground truth, when the different options for bench-marking SAST-tools are compared. This is why in the latest SAMATE (Delaitre et al., 2018) all three kinds of test code has been used. Because of limited resources we decided to focus mainly on the synthetic test code, since it is more easier to evaluate.

Tool coverage Since we tested all tools with all test cases - regardless of their defect coverage - our results might differ from other published ones, especially the Recall. For example in the SAMATE (Delaitre et al., 2018) they have a special measure "Applicable Recall", where they take the supposed tool coverage into account, which "enhances"

their Recall significantly in most cases. We have decided against such a measure, to make the results more comparable, as already justified in the Introduction.

Tool configuration A major threat for the validity is the bias which can be introduced through more or less existing knowledge regarding the configuration options of the SAST tools, by their user. An experienced user would probably use more or different options for one or the other tool, and thus gain better results, regarding Precision and especially Recall. In fact, we used options for three tools notably: Clang-Tidy, IKOS and Sparse. For Sparse we enabled all warnings to enhance the Recall. Since the results are still at the very low end, this bias can probably be neglected. In the case of IKOS we chose a more accurate numerical abstraction than the default one. As IKOS is a sound tool, this mainly affects the false positive rate, and therefore influences only the MCC. The latter is still lower for IKOS as for Frama-C, which we used in default configuration. In the case of Clang-Tidy, we disabled all the style checks to reduce "false positives" and enabled some additional security checks. This might have given it some advantage over AdLint where we didn't turn off style checks (we didn't know how). In general, the decision of "if and how" the tools should be configured to make up a fair comparison remains an open research question.

Procedure As discussed in the procedure section, our algorithm for counting true positives by matching findings with exact code locations is based on the assumption, that it is unlikely, that a tool would hit the right location, but finds a complete different error in it. Therefor, our results rely on this assumption, since Precision and Recall are calculated from that. A very noisy tool that marks almost every line of code as erroneous with a random report would have a very low Precision at one hand, but anyway would gain a high Recall, since we do not check if it is really reporting the right defect type. If one would average Precision and Recall by the F-Score, such a tool would finally gain a reasonable ranking. Such is what we observed for AdLint. To compensate for this effect, we computed the MCC instead of the F-Score for all tools which also rewards true negatives and therefore penalizes such noisy tools. Nonetheless, our approach also penalizes tools which finds the defect at a different code location as intended. Also penalized are all tools, which finds other defects in the test suite than the intended ones. This is especially true for the sound tools in our evaluation (Frama-C, IKOS), since those even stop the whole analysis when they find an error, and so are unable to detect further errors in the current test case. That means, if such an unintended defect is approached before the intended one, a sound tool would gain one "false positive" for finding the unintended defect, and further gain one "false negative" for not finding the intended one, as the analysis already stopped. This is why the sound tools gained such a low Precision of under 50% in our evaluation and

also a relatively lower Recall in the "Memory Access" and "Risky Values" fault clusters - which are their specialties. It is evident, that those unintended defects exist in the Juliet Test Suite as seen in former research work (see Stikkelorum (2016)). One solution would be to use the Juliet Test Suite as intended, which in turn would introduce other problems as described in former sections. So there is a trade off in either direction. In the case of the sound tools, it is apparent, that unintended defects in test cases has to be avoided. Further, it seems the absence of unintended defects to be a general requirement to a test suite, since we don't know the inner workings of each tool and have to assume that each tool could stop its analysis when it reaches a severe error in the code. Also, the problem with false-"false positives" would be mitigated, if there are no "other" defects in the test cases. The creation of such a test suite - as exhaustive in "depth" and "width" as the Juliet Suite - but free from unintended defects and with better annotations for all eligible defect types, remains an open task for future works in this field.

6 Conclusion

With this work we want to support developers who want to use static analysis tools for security testing and need an overview which tools exist, and how reliably they find vulnerabilities in C code without being annoyed from too much false positives. For this we did a preliminary investigation on which fault patterns or weaknesses occur in real open source code, by examination of CVEs related to C-based open source projects distributed with Debian Linux. We found out, that half of all the vulnerabilities are coding errors related to memory management issues alone, like e.g. buffer overflows. Other important vulnerability causes are resource management errors and insufficient input validation. Then we decided to use the Juliet Test Suite for C/C++ for our evaluation, since it covers well all of these weaknesses with a similar weighting. Since only one of the tools under test had an option to report CWEs, we decided not to match the defect types of the tests with the findings of the tools, to avoid more effort - but instead to match directly on the line of code where the defect was located. We could approve some findings from other publications, which used other test suites and/or evaluation procedures. We can confirm that Frama-C is one of the best open source static analysis tools for C available, even though it is not specifically focused on security. Another sound tool "IKOS", which has not been evaluated in any publication before, performed similar good and ranks second in our benchmark. When using sound tools, the external dependencies have to be modeled by hand to make the tools reason properly about their behavior. The benefit is, that with sound tools it is theoretically possible, to prove the total absence of defects in the code. Therefor sound tools would be our suggestion for projects where safety and/or security have a high priority. The drawback of them is, that their usage doesn't scale very well because of the necessary dependencies modeling, and is only recommended for smaller projects and/or projects with few dependencies. On the side of unsound tools Clang and CppCheck would be recommended, which also ranked high in several other publications. Those tools can be easily added to any CI system, and scale well with bigger software with a lot of dependencies. Although they are able to find a lot of common programming errors, they can not guarantee the absence of run-time errors as sound tools can.

Future benchmarks would benefit from a new or overhauled test suite, which solves some problems we already mentioned, like absence of unintended defects or better annotations. It would be of general interest to develop such a suite like an open standard, where tool

developers and security researchers could work together. Also, when it comes to the benchmark itself it could be more fair if the tool developers would be involved, such as with delivering the tool configuration for optimal results. To cope up with the rapid out dating of such benchmarks, the establishment of a regular and an automated evaluation would be a valuable contribution to more secure open source software.

Bibliography

Andrei Arusoaie, Ştefan Ciobâcă, Vlad Craciun, Dragos Gavrilut, and Dorel Lucanu. A comparison of open-source static analysis tools for vulnerability detection in C/C++ code. In *19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2017, Timisoara, Romania, September 21-24, 2017*, pages 161–168, 2017. doi: 10.1109/SYNASC.2017.00035. URL https://doi.org/10.1109/SYNASC.2017.00035.

Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, pages 3–11, 2015. doi: 10.1007/978-3-319-17524-9_1. URL https://doi.org/10.1007/978-3-319-17524-9_1.

G. Chatzieleftheriou and P. Katsaros. Test-driving static analysis tools in search of c code vulnerabilities. In *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, pages 96–103, July 2011. doi: 10.1109/COMPSACW.2011.26.

Davide Chicco. Ten quick tips for machine learning in computational biology. *BioData Mining*, 10(1):35:1–35:17, 2017. doi: 10.1186/s13040-017-0155-3. URL https://doi.org/10.1186/s13040-017-0155-3.

Debian. Security auditing tools. https://www.debian.org/security/audit/tools.en.html, 2019.

Aurelien Delaitre, Bertrand Stivalet, Paul E. Black, Vadim Okun, Athos Ribeiro, and Terry S. Cohen. Sate v report: Ten years of static analysis tool expositions. Technical report, NIST, 2018. URL https://samate.nist.gov/SATE5/SATE5%20Report.pdf.

Eclipse Foundation. Iot developer survey 2017, 2017. URL https://www.slideshare.net/IanSkerrett/iot-developer-survey-2017.

Katerina Goseva-Popstojanovaa and Andrei Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 2015. doi: 10.1016/j.infsof.2015.08.002. Juliet test suite.

Jörg Herter, Daniel Kästner, Christoph Mallon, and Reinhard Wilhelm. Benchmarking static code analyzers. In *Computer Safety, Reliability, and Security - 36th International Conference, SAFECOMP 2017, Trento, Italy, September 13-15, 2017, Proceedings*, pages 197–212, 2017. doi: 10.1007/978-3-319-66266-4_13. URL https://doi.org/10.1007/978-3-319-66266-4_13.

Rick Kuhn, Mohammad S. Raunak, and Raghu Kacker. Can reducing faults prevent vulnerabilities? *IEEE Computer*, 51(7):82–85, 2018. doi: 10.1109/MC.2018.3011039. URL https://doi.org/10.1109/MC.2018.3011039.

Bailin Lu, Wei Dong, Liangze Yin, and Li Zhang. Evaluating and integrating diverse bug finders for effective program analysis. In Lei Bu and Yingfei Xiong, editors, *Software Analysis, Testing, and Evolution*, pages 51–67, Cham, 2018. Springer International Publishing. ISBN 978-3-030-04272-1. doi: 10.1007/978-3-030-04272-1_4. URL https://link.springer.com/content/pdf/10.1007%2F978-3-030-04272-1_4.pdf.

B.W. Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2):442 – 451, 1975. ISSN 0005-2795. doi: https://doi.org/10.1016/0005-2795(75)90109-9. URL http://www.sciencedirect.com/science/article/pii/0005279575901099.

Jonathan Moerman. Evaluating the performance of open source static analysis tools, 2018. URL https://www.cs.ru.nl/bachelors-theses/2018/Jonathan_Moerman__4436555___Evaluating_the_performance_of_open_source_static_analysis_tools.pdf.

NSA-CAS. On analyzing static analysis tools. Technical report, National Security Agency Center for Assured Software, 2011. URL https://media.blackhat.com/bh-us-11/Willis/BH_US_11_WillisBritton_Analyzing_Static_Analysis_Tools_WP.pdf.

Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. Test suites for benchmarks of static analysis tools. In *2015 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Gaithersburg, MD, USA, November 2-5, 2015*, pages 12–15, 2015. doi: 10.1109/ISSREW.2015.7392027. URL https://doi.org/10.1109/ISSREW.2015.7392027.

Wouter Stikkelorum. Challenges of using sound and complete static analysis tools in

industrial software. Master's thesis, Universiteit van Amsterdam, 2016. URL http://www.scriptiesonline.uba.uva.nl/document/642711.

A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. doi: 10.1112/plms/s2-42.1.230. URL https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230.

W3Techs. Usage of operating systems for websites. https://w3techs.com/technologies/overview/operating_system/all, 2019.

Wikipedia. List of tools for static code analysis. https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis#C,_C++, 2019.

W. J. Youden. Index for rating diagnostic tests. *Cancer*, 3(1):32–35, 1950. doi: 10.1002/1097-0142(1950)3:1<32::AID-CNCR2820030106>3.0.CO;2-3. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/1097-0142%281950%293%3A1%3C32%3A%3AAID-CNCR2820030106%3E3.0.CO%3B2-3.