An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection

Stephan Lipp
Technical University of Munich
Germany

Sebastian Banescu
Technical University of Munich
Germany

Alexander Pretschner
Technical University of Munich
Germany

ABSTRACT

Static code analysis is often used to scan source code for security vulnerabilities. Given the wide range of existing solutions implementing different analysis techniques, it is very challenging to perform an objective comparison between static analysis tools to determine which ones are most effective at detecting vulnerabilities. Existing studies are thereby limited in that (1) they use synthetic datasets, whose vulnerabilities do not reflect the complexity of security bugs that can be found in practice and/or (2) they do not provide differentiated analyses w.r.t. the types of vulnerabilities output by the static analyzers. Hence, their conclusions about an analyzer's capability to detect vulnerabilities may not generalize to real-world programs. In this paper, we propose a methodology for automatically evaluating the effectiveness of static code analyzers based on CVE reports. We evaluate five free and open-source and one commercial static C code analyzer(s) against 27 software projects containing a total of 1.15 million lines of code and 192 vulnerabilities (ground truth). While static C analyzers have been shown to perform well in benchmarks with synthetic bugs, our results indicate that state-of-the-art tools miss in-between 47% and 80% of the vulnerabilities in a benchmark set of real-world programs. Moreover, our study finds that this false negative rate can be reduced to 30% to 69% when combining the results of static analyzers, at the cost of 15 percentage points more functions flagged. Many vulnerabilities hence remain undetected, especially those beyond the classical memory-related security bugs.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*.

KEYWORDS

static code analysis, vulnerability detection, empirical study

ACM Reference Format:
Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534380

1 INTRODUCTION

Context. Dealing with software weaknesses is an inherent part of software development. Organizations expend a non-negligible amount of effort on detecting such weaknesses as early as possible in the software life-cycle [29]. The security domain, with a consistently high number of Common Vulnerabilities and Exposures (CVE) submissions year after year, still sees C (and C++) among the programming languages that are at the root of most vulnerabilities [66]. Accordingly, researchers have been proposing ways to detect vulnerabilities, including techniques such as static code analysis, dynamic software testing, and formal verification.

Beller *et al.* [20] examined 168,214 open-source projects to find out if and how static code analyzers are used in practice. Their results show that the usage of such tools is widespread, *i.e.*, 77% of the projects employ at least one static analyzer. Static code analysis is thereby mostly used by software developers [60, 67] to automatically scan source code (without executing it) in order to find security vulnerabilities. Furthermore, static analyzers are usually cheaper to set up and execute than dynamic testing tools. For example, greybox fuzzers [21, 22, 36] or concolic execution engines [26, 57, 69] require a test harness as well as extensive code instrumentation to test a given piece of software. They also need to be run for several hours to increase the chances of detecting vulnerabilities, while static analyzers can fully scan large codebases in less than an hour. However, dynamic testing tools do not produce false positives, *i.e.*, findings in the code that are non-issues, as each observed program failure indicates an actual software weakness. This is a common criticism [30, 39, 43, 44] of static analyzers and has been addressed in many research papers [19, 46, 48, 52, 55, 59, 67]. However, a less studied limitation is their false negative rate, *i.e.*, software weaknesses that are not detected even though the static analyzer should be able to find them.

Problem and State-of-Practice. Existing studies measure the effectiveness of static code analyzers mainly on synthetic benchmark datasets [13, 16, 28, 33, 38, 40, 45, 56, 61, 63, 68]. These are datasets that contain software bugs added either automatically by so-called bug injection engines, as *e.g.* in the LAVA-M dataset [34], or manually by security experts such as in the Juliet Test Suite [12, 23]. However, the injected synthetic bugs are relatively easy to spot [25, 37], as they are usually inserted in the form of syntactic code changes to a single instruction (*e.g.*, off-by-one array access). Many evaluations [13, 28, 38, 40, 61, 63] performed on such benchmarks thereby report detection rates around 80%—for certain types of vulnerabilities even 100%—for some of the analyzers studied. Infer [10], for example, a static analyzer also used in our benchmark, detects in the Juliet Test Suite for C/C++ on average 79% of the vulnerabilities [63] across four different Common Weakness Enumeration (CWE) categories. However, it is questionable whether the high detection rates are representative in the sense that they also apply to the more complex vulnerabilities that occur in practice. Furthermore, this caveat calls into question the vulnerability types best and worst detected, as well as the performance increase that can be achieved by combining multiple analyzers, reported by some of those studies. This information, if available for real-world programs, would allow researchers and practitioners to gain deeper insights into the strengths and weaknesses of static code analyzers, as well as the trade-offs (detection increase *vs.* analysis overhead) when using multiple tools in combination. Up to now, we are not aware of any work that can answer these questions for real-world programs. Another thread observed in related studies [11, 32, 40, 41, 65, 71] is that they do not check whether the vulnerable code locations used as ground truth, *i.e.* fault, error, or failure locations [18], are also those that static analyzers are able to find in the first place. The wrong code granularity for approximating vulnerability detection can thereby render the entire evaluation invalid.

Solution and Contributions. To address the above gaps, we propose an automated and reproducible approach to assess the effectiveness of five free and open-source (FOS) and one commercial static C code analyzer(s) using a benchmark dataset that consists of 27 FOS projects with 192 known security bugs, *i.e.*, validated CVEs (ground truth). For this, we also examine the code locations typically marked by static analyzers and those of the vulnerabilities in our dataset. We do this to determine (1) if our dataset can generally be used to evaluate such analyzers and (2) what an appropriate code granularity is (*w.r.t.* our dataset) to approximate vulnerability detection. Furthermore, due to the lack of empirical research on the benefits of using multiple static analyzers in combination, we analyze the increase/trade-off in bug detection and number of flagged functions of single *vs.* combined analyzer usage. As final part of our study, we identify the types of vulnerabilities that were reliably detected, as opposed to those that remained largely undetected.

This research paper presents the following *contributions*:

(1) We perform an in-depth analysis determining function-level as the code granularity best suited to automatically evaluate the effectiveness of static code analyzers on CVE-based benchmark datasets (see Section 3.2).
(2) We conduct a large-scale empirical study of five FOS and one commercial static C analyzer(s), showing that when run on a benchmark dataset with known real-world vulnerabilities (192 validated CVEs)
   - even in the least restrictive evaluation scenario, the state-of-the-art static analyzers chosen detect not more than roughly half (53%) of the included software vulnerabilities (see Section 5.1),
   - using multiple analyzers can increase the detection rate by 21 to 34 percentage points (depending on the eval. scenario) compared to using only one tool, while flagging about 15pp more functions (see Section 5.2), and
   - vulnerabilities that belong to the weakness categories `CWE-{664,703}` are more effectively detected than those of `CWE-{682,707,691}` (see Section 5.3).
(3) We release all data and the analysis script to foster comparable and evidence-based studies on static code analysis: https://doi.org/10.5281/zenodo.6515687

2 STATIC CODE ANALYSIS

2.1 Techniques

Syntactic Static Analysis. This analysis technique searches for syntactic patterns in the source code that might indicate a vulnerability. Examples of such code patterns are calls to dangerous C functions such as `memcpy` or `strcpy`, which can lead to safety-critical system behavior when called with incorrect arguments. This technique can thereby be applied to all source files in a codebase as well as to specific parts of the source code as the analysis does not require compilable code artifacts.

Semantic Static Analysis. This technique takes the program semantics, *i.e.*, control- and/or data-flow information, into account when searching for vulnerabilities. More specifically, the source code is first lifted into a more abstract representation, such as an abstract syntax tree, call graph, or control-flow graph. Then, certain security checks are performed on that representation in order to find vulnerabilities. Despite the problem of undecidability [49] that comes with semantic static analysis[1], it allows searching for more complex vulnerabilities.

2.2 Selected Analyzers

We selected six different static code analyzers (five that are free and open-source and one commercial tool) that support C code. These tools implement state-of-the-art analysis techniques and were used in previous benchmarks [62] with synthetic software bugs and/or are popular among practitioners, using GitHub stars (★) as an indicator [24] for this.

2.2.1 Flawfinder (FLF). This static analyzer is licensed under the GPLv2. Here, we use version 2.0.11 of Flawfinder [9] (★ 250), released in February 2020. Flawfinder implements a syntactic analysis technique that scans C/C++ source code for potentially vulnerable code patterns stored in a local database. The integrated rules primarily identify functions that have been susceptible to vulnerabilities in the past and also assess their risk of triggering a security bug by analyzing the arguments.

2.2.2 Cppcheck (CPC). This static analyzer is released under the GPLv3. In this study, we employ version 2.3 of Cppcheck [6] (★ 3.9k), which was released in December 2020. Cppcheck scans C and C++ source code for security-critical bugs using lightweight data-flow analysis. Using a combination of syntactic and semantic analysis techniques, Cppcheck focuses on detecting software vulnerabilities caused by undefined C/C++ behavior as well as on critical source code constructs.

2.2.3 Infer (IFR). This static analyzer is developed by Meta (formerly known as Facebook, Inc.) and is released under the MIT License. In this study, we use Infer [10] (★ 12.9k) version 0.14.0[2], released in April 2018. Infer implements a semantic analysis approach that utilizes formal verification techniques such as separation logic [58] and bi-adduction [27], allowing to reason about

[1] Symbolic execution [47] and abstract interpretation [31] also fall into this category, but were not considered in this study.
[2] Although newer versions of Infer were available at the time of conducting the experiments, this version was the only one we could get run on an older Ubuntu GNU/Linux system needed to compile and analyze the older program versions of Binutils and FFmpeg.

pointer structures (such as those in C/C++ to manipulate memory), in order to detect software vulnerabilities.

2.2.4 CodeChecker (CCH). This refers to an entire static analysis platform that is released under the Apache-2.0 License. Here, we use CodeChecker [3] (★ 1.5k) version 6.12.0, released in May 2020. By default, it employs the LLVM/Clang static analysis toolchain consisting of the semantic analyzers Clang Static Analyzer [1] and Clang-Tidy [2]. This analysis platform is thereby not limited to these two C/C++ analyzers, *i.e.*, other static analyzers can be added to run them in combination.

2.2.5 CodeQL (CQL). This static analysis engine is released under a custom license that allows to freely use CodeQL [4] (★ 4.1k) for academic research. In this study, we use version 2.1.3 of CodeQL, released in May 2020. CodeQL implements a semantic analysis technique where the analysis engine is decoupled from the respective rules/checks, formulated in a query language called QL [17]. Predefined rule packages thereby already exist that can be used to search for security-relevant weaknesses in the code.

2.2.6 CommSCA (CSA). This static analyzer is the only commercial tool used in this study. Similar to the other analyzers, we made sure to use one of the latest versions of CommSCA. To protect the company behind this static code analyzer, we anonymized its name and neither reveal its exact version and release date, nor the implemented analysis technique.

2.3 Automated Analyzer Evaluation

Common Weakness Enumeration (CWE). CWE [5] refers to a category system for software (and hardware) weaknesses, including security vulnerabilities, which is also used in CVE reports and supported by many static analyzers [4, 6, 7, 9, 64]. Each weakness type included in this enumeration has a unique identifier as well as a description that indicates how it relates to other types. This relationship is thereby specified as a child-parent hierarchy, meaning that the child CWE is a more concrete software weakness instance of the parent CWE. Top-level CWEs with no further parent CWEs, such as `CWE-664`: "Improper Control of a Resource Through its Lifetime", can therefore be considered the lowest common denominator of all subjacent child CWEs. Accordingly, these high-level CWEs represent vulnerability classes, while all CWEs below indicate vulnerability types.

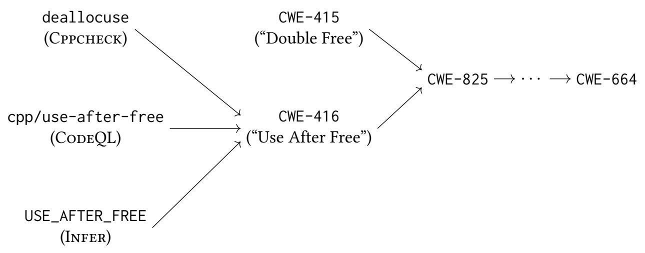

Figure 1: Example of mapping and grouping analyzer-specific use-after-free vulnerability identifiers to the vulnerability class `CWE-664`.

Table 1: Benchmark Programs

Subject	Version	LoC	# Functions	# Vulns.
Libpng	1.6.38	10,184	398	7
LibTIFF	4.1.0	19,527	826	13
Libxml2	2.9.10	85,442	2,982	17
OpenSSL	3.0.0	165,187	13,036	21
PHP	8.0.0-dev	209,407	9,145	15
Poppler	0.88.0	63,561	4,659	22
SQLite3	3.32.0	53,272	2,298	16
Binutils	2.29	134,767	4,071	59
FFmpeg	n3.3.2	413,353	17,788	22
Total		1,154,700	55,203	192

CWE Mapping and Grouping. Different static analyzers use different identifiers for the types of vulnerabilities they support. Flawfinder, for example, use CWEs, while others introduce their own vulnerability identifiers. These different identifiers make it difficult to automatically assess whether a static analysis tool is referring to the correct vulnerability type, which would allow for a more rigorous evaluation. For this reason, we created a mapping that assigns each analyzer-specific vulnerability identifier to the corresponding analyzer-agnostic CWE ID. An example of this mapping can be found in Fig. 1, where the use-after-free vulnerability identifiers of the analyzers Cppcheck, CodeQL, and Infer are mapped to `CWE-416`: "Use After Free".

Many C-specific vulnerabilities are closely related. For example, double-free vulnerabilities (`CWE-415`) are related to use-after-free ones (`CWE-416`); hence, `CWE-825`: "Expired Pointer Dereference" constitutes the parent of both types. Consequently, by comparing only low-level vulnerability types, we would diminish the effectiveness of the analyzers that do not output the exact CWE IDs, but closely related ones that are also correct. Therefore, we leverage the existing CWE hierarchy as proposed by Goseva-Popstojanova and Perhinschi in [40] to group related vulnerability types into classes. Using this CWE grouping, we can now automatically and reproducibly evaluate if the class of the vulnerability issued by the tools matches that of the vulnerability in the code.

3 BENCHMARKING

3.1 Benchmark Dataset — Ground Truth

Selection Criteria and Existing Datasets. For our evaluation, we searched for benchmark datasets with C programs that contain a representative and diverse set of well-documented security vulnerabilities. Existing work [13, 16, 28, 33, 38, 40, 45, 56, 61, 63, 68] thereby mostly utilize programs that include synthetic software bugs [8, 34, 61] such as those in the widespread Juliet Test Suite [12]. However, these bugs do not necessarily reflect the complexity of real-world vulnerabilities [25, 37]. Unfortunately, datasets with real-world security bugs are rather rare and those that are available contain only few vulnerabilities (mostly of the same type) [53] or are insufficiently documented, *e.g.*, they do not specify the vulnerability types [14, 15].

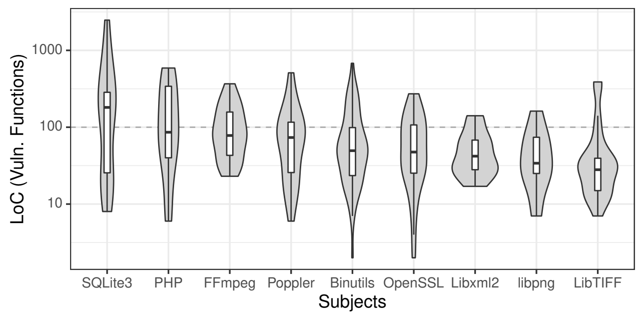

Figure 2: Log-scaled distribution of lines of code (LoC) of functions affected by ≥1 vulnerabilities across the different benchmark programs.

Magma plus Open-Source Programs. An exception to this is Magma [42], a benchmark dataset built from validated CVE reports, originally designed to evaluate the effectiveness of fuzzers. The included programs contain a large and diverse set of vulnerabilities that should also be detectable by static C analyzers (discussed below). Magma thereby uses a technique called front-porting, where security bugs found and publicly reported in the past are reinserted into the latest program version. For each ported vulnerability, Magma specifies—besides the root cause—the function(s) where it manifests and may lead to a program crash.

Besides Magma (version v1.1), which contains 111 vulnerabilities (CVEs), we also employ an older version of the Binutils suite, consisting of 19 programs for manipulating compiled code, and the video/audio processing tool FFmpeg, as they contain many well-documented vulnerabilities[3] (81 non-front-ported vulnerabilities in total). An overview of our benchmark programs can be found in Table 1. This table is supplemented by Fig. 2, showing that except for SQLite3, the arithmetic mean of lines of code of the functions affected[4] by one or more vulnerabilities is below 100 LoC.

Vulnerability Validation. Since many of the employed analyzers attach themselves to the build process, we checked[5] that none of the vulnerable source code was removed by the preprocessor due to an improper build configuration. In cases vulnerable code has been removed, we either reconfigured the build process or, if we could not adjust the configuration accordingly, omitted[6] the vulnerability from the evaluation.

Furthermore, when using real-world programs in such evaluations, there is the possibility that they contain vulnerabilities that have not been discovered yet. Accordingly, a static analyzer may detect a new vulnerability that is then not considered in the final rating. However, since we evaluate whether the static analyzers manage to find known and existing security bugs in our benchmark dataset, we consider this permissible to draw conclusions about their effectiveness.

[3] We manually examined every CVE-related commit message and code changes to extract the functions affected by the vulnerabilities.

[4] By "affected" we mean that a code block contains at least one incorrect instruction that (along with others) cause the vulnerability.

[5] We scanned the LLVM bitcode file(s) [50] of the respective programs for the vulnerable functions using a self-written compiler pass.

[6] The software vulnerabilities `CVE-2019-19959`, `CVE-2017-15286`, `CVE-2019-19925`, `CVE-2019-9936` in SQLite3, `CVE-2019-9022` in PHP, and a buffer-overread vulnerability in Libxml2 (denoted `BUG #758518` in Magma) could not be verified in the LLVM bitcode files and were therefore omitted from the evaluation. We also reported these vulnerabilities to the Magma creators.

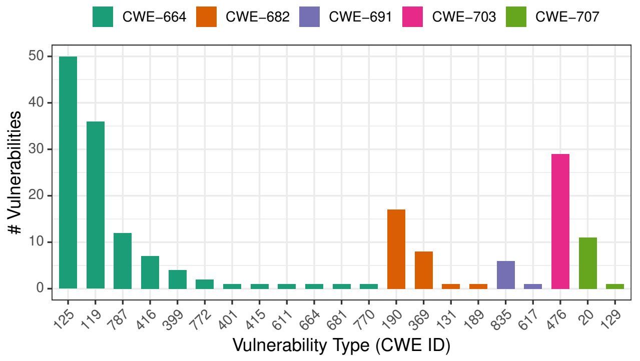

Figure 3: Distribution of the vulnerability types included in our benchmark dataset. Each type thereby belongs to one of the five vulnerability classes `CWE-{664,682,691,703,707}`, which count 117, 27, 7, 29, and 12 (in total 192) vulnerabilities, respectively.

Vulnerability Types and Classes. Figure 3 shows the distribution of vulnerability types (CWE IDs) of the 192 CVEs in our benchmark dataset. In total, it contains 21 different types, grouped into the five vulnerability classes described below.

3.1.1 Improper Control of a Resource Through its Lifetime (CWE-664). This class refers to memory-related vulnerabilities where the software improperly retains control of a resource. Instances are incorrect out-of-bound read/write operations (`CWE-{119,125,787}`), use-after-free vulnerabilities (`CWE-{415,416}`), resource management errors (`CWE-{399,770,772,401}`), and incorrect conversion between numeric types (`CWE-681`).

3.1.2 Incorrect Calculation (CWE-682). Vulnerabilities that belong to this class originate from incorrect calculations whose results are later used in security-critical source code such as memory allocations/accesses. Typical for this class are divide-by-zero (`CWE-369`) and integer-overflow vulnerabilities (`CWE-190`), which can lead to wrong buffer size calculations (`CWE-131`) like the one in function `xmlMemStrdupLoc` (see Fig. 4).

3.1.3 Insufficient Control-Flow Management (CWE-691). This class represents vulnerabilities where the program control-flow is manipulated so that the security of the software system is compromised. Examples are loops whose exit condition is never reached/satisfied (`CWE-835`), allowing attackers to consume excessive resources, or reachable assertions (`CWE-617`) that can be triggered by an attacker to initiate a denial-of-service (DoS) attack.

3.1.4 Improper Check or Handling of Exceptional Conditions (CWE-703). This class concerns vulnerabilities where exceptional conditions are not properly checked/handled in the source code. A concrete vulnerability type of this class would *e.g.* be a missing `if` statement that prevents a `NULL` pointer dereference (`CWE-476`), resulting in a program crash.

Table 2: Supported Vulnerability Classes

	Analyzer					
Vuln. Class	FLF	IFR	CPC	CCH	CQL	CSA
CWE-664	✓	✓	✓	✓	✓	✓
CWE-682	✓	✓	✓	✓	✓	✓
CWE-703		✓	✓	✓	✓	✓
CWE-707	✓			✓	✓	✓
CWE-691	✓		✓		✓	✓

3.1.5 Improper Neutralization (CWE-707). This refers to vulnerabilities where the program input/output is insufficiently neutralized against security threats. Examples include malformed strings passed via parameters or environment variables to a program that are not validated (CWE-20), as well as improper validation of array indices (CWE-129) that pose a security risk such as remote code execution.

Supported Vulnerability Classes. Table 2 shows for each vulnerability class if it is supported (✓) by the respective static analyzer, or not (blank). A class is thereby considered supported if the documentation states that the respective analyzer implements a security check for at least one of the vulnerability types in that specific class. As shown in the table, most vulnerability classes—especially CWE-{664,682}—are supported by the studied analyzers. However, class CWE-703 is not supported by Flawfinder and CWE-707 not by Infer and Cppcheck. Moreover, Infer and CodeChecker do not support CWE-691 vulnerabilities.

3.2 Vulnerability Detection Granularity

Root Cause vs. Manifestation. In general, static analyzers rather mark the code location(s) where a vulnerability potentially manifests as a security-critical program state (error) during execution, rather than the location(s) of the corresponding root cause (fault) [18]. One reason for this is to reduce the number of false positives, since not every fault necessarily manifests as a security bug. The error of a vulnerability can thereby occur at places in the code that are different from those of the fault.

Figure 4 gives an example of this issue, where a potential integer-overflow in function strlen on line 3 (root cause/fault) may manifest as a out-of-bounds write vulnerability (CWE-787) from line 6 onwards. Thereby, none of the six analyzers flag the line containing the root cause as vulnerable. Instead, CodeQL and CommSCA mark line 6 as containing CWE-401: "Missing Release of Memory after Effective Lifetime". Moreover, on line 18 CommSCA indicates CWE-676: "Use of Potentially Dangerous Function", while Flawfinder and CodeQL both output CWE-120: "Buffer Copy without Checking Size of Input". Lastly, CodeQL flags line 19 with CWE-401 (same as line 6 before). All lines marked by those analyzers thereby correctly indicate possible manifestations (including the correct vulnerability class) of this vulnerability.

Abstraction Level. As mentioned before, the benchmark dataset used in this study is based on CVE reports. However, the accuracy across different reports can vary widely [54], making it difficult to find an appropriate code abstraction to automatically check whether a vulnerability was detected by a static analyzer, or not.

```
1   char *xmlMemStrdupLoc(const char *str, const char *file, int
    ↪  line) {
2       char *s;
3       size_t size = strlen(str) + 1;
4       MEMHDR *p;
5       if (!xmlMemInitialized) xmlInitMemory();
6       p = (MEMHDR *) malloc(RESERVE_SIZE + size);
7       if (!p) goto error;
8       p->mh_tag  = MEMTAG;
9       p->mh_size = size;
10      p->mh_type = STRDUP_TYPE;
11      p->mh_file = file;
12      p->mh_line = line;
13      xmlMutexLock(xmlMemMutex);
14      p->mh_number = ++block;
15      xmlMutexUnlock(xmlMemMutex);
16      /* #define CLIENT_2_HDR(a) ((void *)(((char *)(a)) -
        ↪  RESERVE_SIZE)) */
17      s = (char *) HDR_2_CLIENT(p);
18      strcpy(s,str);
19      return(s);
20  error:
21      return(NULL);
22  }
```

Figure 4: Function xmlMemStrdupLoc (without debug code) in Libxml2 containing CVE-2017-5130 which shows the divergence between the root cause on line 3 and the lines 6, 18, and 19 (potential manifestation) marked by Flawfinder, CodeQL, and CommSCA.

Some CVEs in our dataset only name the function(s) where the vulnerability manifests and may lead to a program crash, without specifying the exact lines involved. Others instead describe the root cause (not manifestation) in the code through a software patch (*e.g.*, link to GitHub commit). These heterogeneous vulnerability descriptions (fault *vs.* error and function- *vs.* instruction-level) in the CVEs, and the fact that static analyzers mark the manifestation of a vulnerability in the code rather than its root cause, led us to choose function-level as the code abstraction for our evaluation. Note that using a more fine-grained code abstraction, *e.g.* lines or basic blocks, would require to manually determine all possible instructions where a vulnerability may manifest. However, this not only requires a lot of time and effort for the 192 vulnerabilities, but is also very subjective and can distort the entire evaluation.

Fault-Error Location Conformity. For our evaluation to work, fault (root cause) and error (manifestation) of the vulnerabilities in our benchmark must occur within the same functions. Otherwise, for CVEs that only specify the location(s) of the fault, but not that of the error, we cannot tell if the vulnerability was detected or not, as a static analyzer may mark the correct manifestation in a function other than that containing the fault. To verify this, we use the metric

$$\textit{Fault-Error Conformity (FEC)} = \frac{|\mathcal{F}_{\text{fault}} \cap \mathcal{F}_{\text{error}}|}{|\mathcal{F}_{\text{error}}|} \qquad (1)$$

which for a given vulnerability (CVE) computes the ratio of error-containing functions ($\mathcal{F}_{\text{error}}$) that also include the underlying fault ($\mathcal{F}_{\text{fault}}$). Accordingly, the higher the FEC ratio, the more fault and error locations of a vulnerability overlap within the same function(s).

Table 3: Fault-Error Conformity Results

		Proportion of vulns. with		
Subject	**# Vulns.**	FEC = 1.0	1.0 > FEC > 0.0	FEC = 0.0
Libpng	7	0.57	0	0.43
LibTIFF	13	1	0	0
Libxml2	17	0.94	0	0.06
OpenSSL	21	1	0	0
PHP	15	0.93	0	0.07
Poppler	22	1	0	0
SQLite3	16	1	0	0
Mean		0.92	0	0.08

Note that we can perform this analysis only for the vulnerabilities in Magma as the provided CVE patch files additionally specify the functions in which they manifest. For our evaluation, we assume similar Fault-Error Conformity results for Binutils and FFmpeg as they are comparable to the Magma programs in terms of program size, application domain, and vulnerability types contained.

Table 3 shows for each Magma program the proportion of the 111 vulnerabilities where (1) fault and error location(s) fully overlap in the same function(s) (FEC = 1.0), (2) some intersecting functions exist (1.0 > FEC > 0.0), and (3) both fault and error occur in disjoint functions (FEC = 0.0). Except for Libpng, where only four of the seven vulnerabilities (57%) completely overlap, in almost[7] all other programs fault and error of the vulnerabilities provided by Magma lie within the same function(s).

Summary (FEC). On average, the root cause (fault) and the manifestation (error, marked by static analyzers) both lie within the same function(s) in 92% of the Magma vulnerabilities. Assuming this also holds for the vulnerabilities in Binutils and FFmpeg (for which we only know the faults), function-level is thus a suitable code abstraction to evaluate the effectiveness of such tools using our CVE-based benchmark dataset.

4 EVALUATION SETUP

4.1 Research Questions

The evaluation presented in this work aims to answer the following research questions:

RQ$_1$ *Static Analyzer Effectiveness.* How effective are state-of-the-art static C code analyzers at detecting vulnerabilities in real-world codebases?

RQ$_2$ *Effectiveness Increase by Analyzer Combinations.* How much more effective is the best combination of static C analyzers than the best single analyzer?

RQ$_3$ *Best vs. Worst Detected Vulnerabilities.* Which classes of vulnerabilities are detected best and worst by static C analyzers?

[7] The vulnerabilities where the affected functions do not entirely overlap are: CVE-2014-9495, CVE-2019-7317, a use-after-free vulnerability that has not yet been assigned a CWE ID (all three in Libpng), CVE-2017-8872 (Libxml2), and CVE-2018-14883 (PHP). All five vulnerabilities belong to the class CWE-664: "Improper Control of a Resource Through its Lifetime".

4.2 Evaluation Metrics and Scenarios

Effectiveness Measures. The goal of the chosen static analyzers is to detect as many vulnerabilities in the code as possible, while minimizing at the same time the number of false analyzer alarms. To evaluate this, we use the metrics:

$$\textit{Vuln. Detection Ratio} = \frac{\textit{\# Detected vulns.}}{\textit{\# All vulns. in benchmark}} \tag{2}$$

$$\textit{Marked Function Ratio} = \frac{\textit{\# Marked Functions}}{\textit{\# All functions in benchmark}} \tag{3}$$

The first formula (*a.k.a.* recall) calculates the proportion of detected vulnerabilities included in the benchmark. Instead of the precision measure—for which we do not have ground truth data (see Section 3.1)—we use the proportion of functions marked as potentially vulnerable by an analyzer (second formula) to approximate the extent of false positives. We consider this a valid approach, since the ratio of functions affected by one or more of the 192 vulnerabilities (*i.e.*, $223/55203 \approx 0.004$) is very small.

Table 4: Evaluation Scenarios

		Comparison of vuln. class	
		No	Yes
# Functions affected by vuln. to detect	≥1	Scenario 1 ($S_{1\text{-}1}$)	Scenario 2 ($S_{1\text{-}2}$)
	All	Scenario 3 ($S_{2\text{-}1}$)	Scenario 4 ($S_{2\text{-}2}$)

Vulnerability Detection Scenarios. Inspired by Thung *et al.* [65], we evaluate the vulnerability detection capabilities of the static analyzers with respect to four different scenarios:

- **Scenario 1 ($S_{1\text{-}1}$):** A vulnerability is considered detected if at least one affected function was marked by the static analyzer, regardless of the issued vulnerability class.
- **Scenario 2 ($S_{1\text{-}2}$):** A vulnerability is considered detected if at least one affected function was marked by the static analyzer together with the correct vulnerability class.
- **Scenario 3 ($S_{2\text{-}1}$):** A vulnerability is considered detected if all affected functions were marked by the static analyzer, regardless of the issued vulnerability class.
- **Scenario 4 ($S_{2\text{-}2}$):** A vulnerability is considered detected if all affected functions were marked by the static analyzer together with the correct vulnerability class.

These evaluation scenarios (summarized in Table 4) allow to examine the effectiveness of the static analyzers from different perspectives. $S_{1\text{-}2}$ and $S_{2\text{-}2}$ thereby tighten our approximation by additionally requiring the analyzers to return the correct vuln. class for a security bug to be counted as detected. Accordingly, tools that perform well in the these two scenarios can accelerate the manual search for *resp.* remediation of vulnerabilities by providing both the right code locations and non-misleading vulnerability types.

4.3 Analyzer and Subject Configuration

Static Code Analyzers. For each analyzer, we studied its documentation, and if a check for a vulnerability type in our benchmark supported by the tool is not enabled by default, we enabled it.

Checks that do not focus on security vulnerabilities, such as code smells or unreachable code, were disabled. For CodeQL, we used the external libraries and queries (security checks) of version 1.23.1, provided by Semmle[8].

Subject Programs. We changed the build process of SQLite3 so that all C files are compiled (and analyzed) separately, instead of merging all source files into a single code file before compilation. Otherwise, this would prevent an automated evaluation of static analyzers that attach to the build process. In Binutils, the vulnerabilities are spread over the 19 programs and also over the code for the manipulations of the different binary formats. Therefore, we (cross-)compiled and analyzed each of the affected Binutils binary format variants separately. Here, we also made sure that we did not consider shared code fragments twice in our evaluation.

4.4 Infrastructure

In this study, we performed all experiments on a machine with an Intel® Xeon® E5-1650v2 processor containing 12 logical cores that run at 3.5 GHz, with access to 128 GB main memory and GNU/Linux Ubuntu 16.04 (64-bit) as operating system.

5 EVALUATION RESULTS

Note that CodeQL did not output any analysis results for all Binutils programs after more than two weeks of running and multiple retries. For this reason, we evaluated CodeQL with zero vulnerabilities found on Binutils.

5.1 RQ.1: Static Analyzer Effectiveness

Program-specific Performance. The low detection rates in Fig. 5 show that many vulnerabilities could not be found by the selected static analyzers. The analyzers performed particularly poorly in Poppler, FFmpeg, and Libpng, possibly due to the following reasons. With respect to Poppler, it is the only C++ program in our benchmark. Although all employed tools support C/C++, it seems they focus primarily on plain C and provide only rudimentary support for C++. As for FFmpeg, it is the largest program in our benchmark with 413,353 lines of code, which may force the static analyzers to abort the analysis when *e.g.* the nesting depth of `if` or `#ifdef` statements reaches a certain limit. Regarding Libpng, the low detection rates may be attributed to the divergent functions of fault and error of some of its vulnerabilities (see Table 3). Furthermore, these results indicate no observable performance difference on programs containing front-ported vulnerabilities and Binutils and FFmpeg, which contain normal vulnerabilities.

Analyzer-specific Performance. Figure 6 shows substantial performance differences between the static analyzers. The most effective ones are CommSCA, CodeQL, and Flawfinder, whereas Cppcheck, CodeChecker, and Infer are those with the fewest vulnerabilities detected. The commercial analyzer CommSCA outperforms in all evaluation scenarios the next best free and open-source static analyzer, *i.e.* CodeQL *resp.* Flawfinder, by 45 (24 percentage points), 26 (13pp), 22 (11pp), and 12 (6pp) more security bugs found. CommSCA thereby marks slightly fewer functions than CodeQL and is therefore likely to return fewer false positives. Interestingly, with about the same number of marked functions, Flawfinder outperforms Infer in all four scenarios. Also, it has roughly the same detection rates as CodeQL in $S_{1\text{-}\{1,2\}}$, while flagging 11pp less functions. This shows that semantic analysis methods are not always more effective than the less complex syntactic ones.

Summary (RQ$_1$). Our empirical evaluation shows that state-of-the-art static C code analyzers overlook a large number of real-world vulnerabilities. Depending on the different evaluation scenarios, even the top-performing analyzer (CommSCA) fails to detect 47% ($S_{1\text{-}1}$), 70% ($S_{1\text{-}2}$), 64% ($S_{2\text{-}1}$), and 80% ($S_{2\text{-}2}$) of the 192 vulnerabilities included in our benchmark dataset.

5.2 RQ.2: Effectiveness Increase by Analyzer Combinations

Best-performing Analyzers and Combinations. Here, we selected the static analyzers and combinations thereof (free and open-source *vs.* commercial) with the most vulnerabilities found in all benchmark programs. A vulnerability is thereby considered found if at least one analyzer from the respective group was able to detect it. Since multiple combinations found the same number of bugs, we selected those that contain the fewest analyzers and thus also output the fewest false positives. As shown in Fig. 7, all selected combinations that contain CommSCA count less than six static analyzers. This implies that CommSCA subsumes all vulnerabilities found by Cppcheck in the scenarios $S_{\{1,2\}\text{-}1}$ and by CodeChecker in $S_{\{1,2\}\text{-}2}$. Furthermore, most of these combinations include Infer, Cppcheck, and CodeChecker, which are rather ineffective when run alone (see Fig. 6)—apparently, they manage to find security bugs that the others overlook. This supports the suggestion of Fatima *et al.* [35] of using multiple analyzers to detect more vulnerabilities.

Performance Improvements. As shown in Fig. 7, the best analyzer combination (Flawfinder-Infer-CodeQL-CodeChecker-CommSCA) detects in scenario $S_{1\text{-}1}$ 34 (17pp) and in $S_{2\text{-}1}$ 30 (16pp) more vulnerabilities than the best single static analyzer (CommSCA), while marking 15pp more functions. In $S_{1\text{-}2}$ and $S_{2\text{-}2}$, Flawfinder-Cppcheck-Infer-CodeQL-CommSCA outperforms CommSCA with 24 (13pp) and 21 (11pp) more vulnerabilities found, again with 15pp more flagged functions. Interestingly, the best combination detects in $S_{1\text{-}1}$, $S_{2\text{-}1}$, and $S_{2\text{-}2}$ more than twice as many security bugs as CodeQL, however, at the cost of flagging roughly double the number of functions. For scenario $S_{1\text{-}2}$, with 6 times more functions marked, the best combination finds more than three times as many vulnerabilities as Flawfinder. Moreover, the best combinations of free and open-source analyzers detect in all four scenarios at least as many vulnerabilities as the commercial tool CommSCA.

Summary (RQ$_2$). Our empirical evaluation shows that using multiple static C code analyzers can improve vulnerability detection in the different evaluation scenarios by 21 to 34 percentage points compared to a single, top-performing analyzer, while marking 15pp more functions as potentially vulnerable. Nonetheless, the best combination(s) still miss 30% ($S_{1\text{-}1}$), 57% ($S_{1\text{-}2}$), 43% ($S_{2\text{-}1}$), and 69% ($S_{2\text{-}2}$) of the 192 vulnerabilities.

[8] https://www.semmle.com/

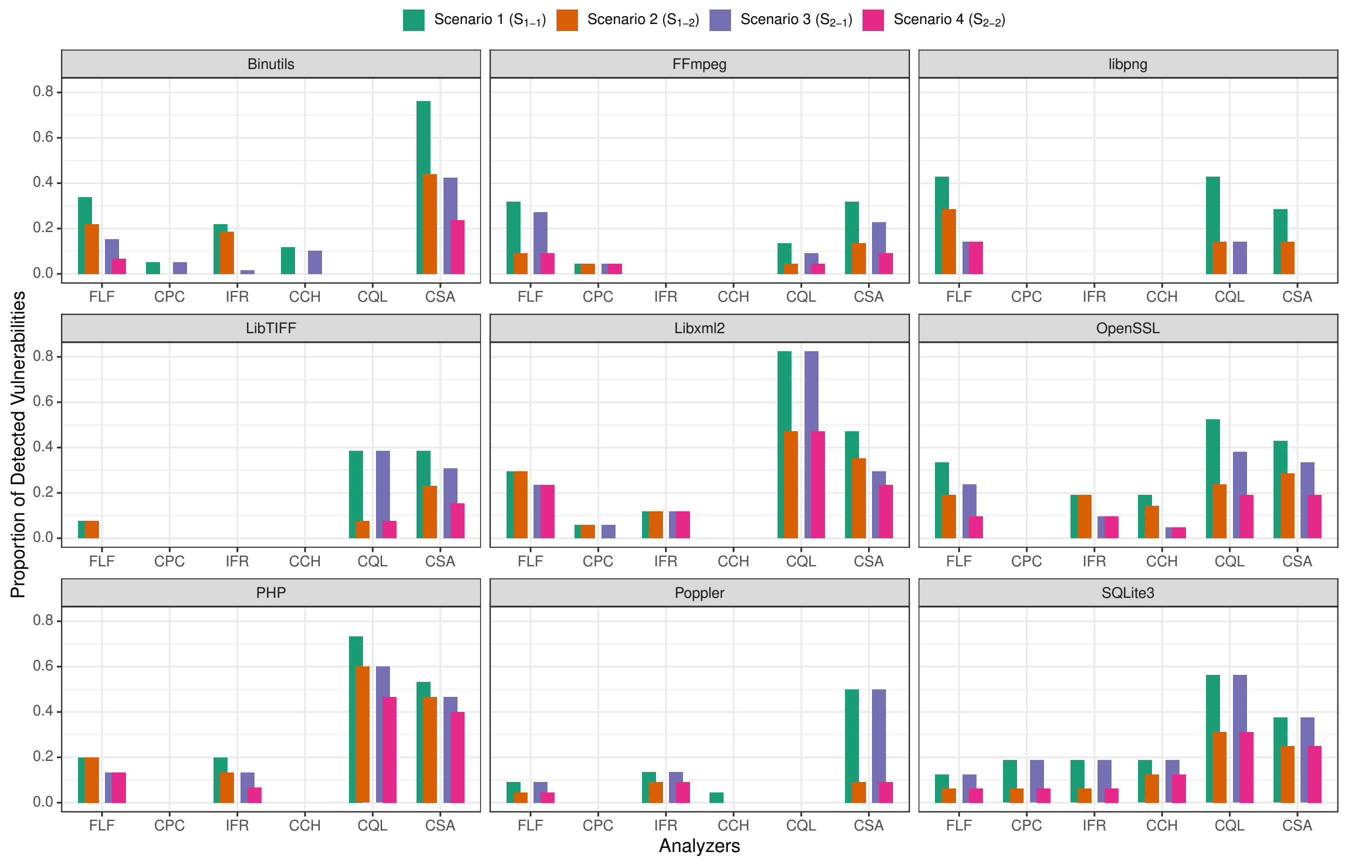

Figure 5: Proportion of vulnerabilities detected by the static C analyzer in the different benchmark programs.

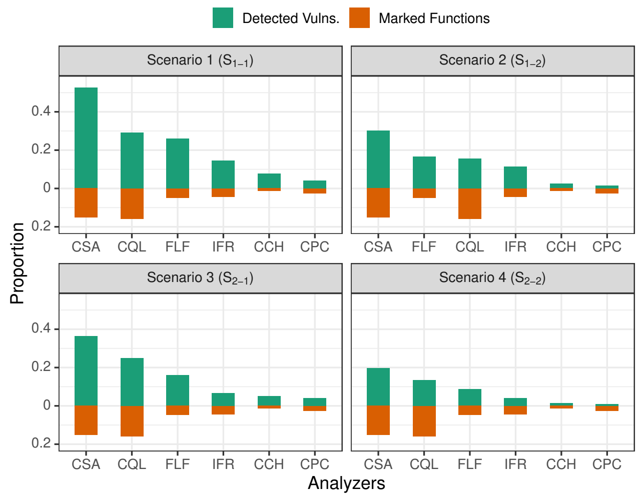

Figure 6: Proportion of detected vulnerabilities and flagged functions (mirrored y-axis scale) by the different static C analyzers in all benchmark programs.

5.3 RQ.3: Best vs. Worst Detected Vulnerabilities

Figure 8 shows the detection rates of the vulnerability classes best and worst detected by the six static analyzers. Note that scenarios S_{1-1} and S_{2-1} ignore the vulnerability class output by the analyzers, hence allowing only limited insights here.

Best Detected Classes. The two vulnerability classes supported by most of the employed analyzers, *i.e.* CWE-{664,703} (see Table 2), are also the ones whose vulnerabilities were detected most frequently in the scenarios S_{1-2} and S_{2-2}. However, 50% (and more) of the CWE-{664,703} vulnerabilities were still overlooked in these two scenarios, revealing once again the deficiencies of state-of-the-art static C analyzers. Also note that CWE-682 vulnerabilities (supported by all six analyzers) are best detected in the scenarios S_{1-1} (78%) and S_{2-1} (70%), which might be an indicator of insufficiently differentiated vulnerability types in these tools.

Worst Detected Classes. The worst detected vulnerability classes, *i.e.* CWE-{691,707}, also coincide with the ones supported by the fewest analyzers. In both scenarios, S_{1-2} and S_{2-2}, one out of seven (14%) included CWE-707 vulnerabilities and only one out of twelve (8%) CWE-691 ones could be found. Now, given that FLAWFINDER, CPPCHECK (CWE-691) *resp.* CODECHECKER (CWE-707), CODEQL, and

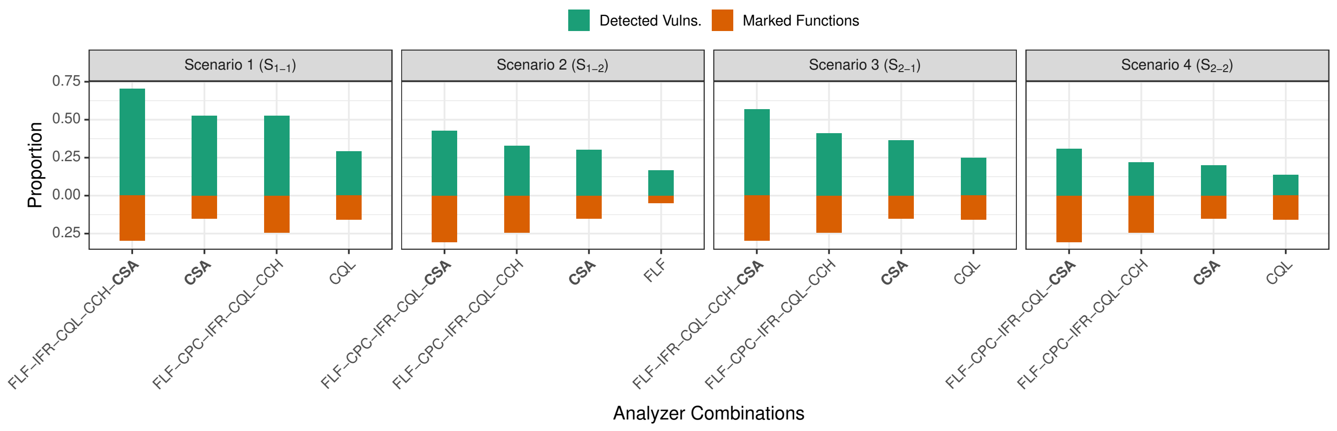

Figure 7: Proportion of detected vulnerabilities and flagged functions by the best single static C analyzers and the best analyzer combinations (free and open-source/commercial) in all benchmark programs.

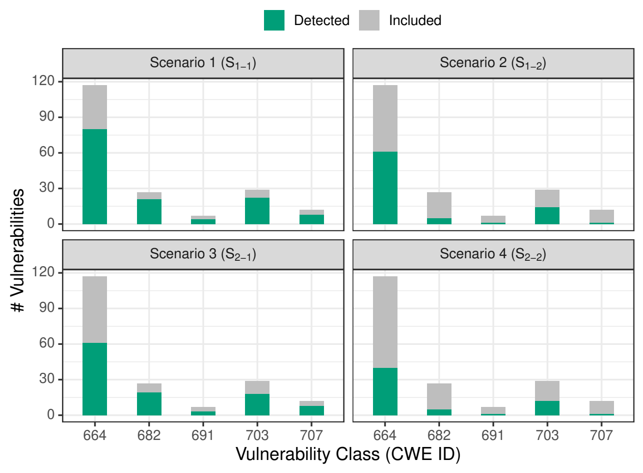

Figure 8: Number of vulnerabilities detected by the static C analyzers in the different vuln. classes.

CommSCA support these vulnerability classes, the chances of detecting such vulnerabilities in real-world programs through static analysis are rather low.

Summary (RQ$_3$). Our empirical evaluation shows that vulnerabilities of the classes `CWE-{664,703}` were more effectively detected by the static C code analyzers than those belonging to `CWE-{682,707,691}`. However, depending on the evaluation scenario, 32%–66% of the 117 `CWE-664` vulnerabilities and 24%–59% of the 29 `CWE-703` ones are missed by the tools.

6 THREATS TO VALIDITY

External Validity. This threat relates to the degree to which our results can be generalized to and across programs and static analysis tools outside of our benchmark. To mitigate this threat, we use a diverse set of 27 real-world programs with a total of 192 vulnerabilities (CVEs). Furthermore, we employ six different static C code analyzers, including one commercial tool, that implement both modern and older but proven analysis techniques.

Internal Validity. The threats discussed hereafter concern the degree to which our empirical study minimizes potential methodological mistakes.

One concern relates to the correctness of the CWE mapping for the static analyzers Infer and CodeChecker. Here, for each analyzer-specific vulnerability identifier, we checked the corresponding description in the documentation to ensure that we assigned an appropriate CWE. For the identifiers that either did not provide a description or one that was unclear to us, we contacted the developers for additional information or let them validate our mapping, respectively.

Another concern is that the front-ported vulnerabilities in the Magma programs negatively impact the effectiveness of the employed static analysis tools, *i.e.*, the code of the newer program versions may make it harder for the tools to detect older, front-ported bugs. To reduce this potential bias, we added 20 more open-source programs (FFmpeg plus the Binutils suite) and thus 82 additional vulnerabilities to our benchmark for which the chosen program versions contain known vulnerabilities.

Moreover, whenever using real-world programs in such an evaluation, there is the chance that they contain further vulnerabilities that have not been detected yet. However, since we evaluate whether the selected static code analyzers succeed in finding known and existing software vulnerabilities, we believe that this allows drawing valid conclusions about their effectiveness.

The last threat to validity concerns our assumption that the code location(s) of the fault (ground truth) and that of the corresponding manifestation (marked by the static analyzers) of the vulnerabilities in FFmpeg and Binutils also lie within the same functions. We cannot guarantee this, but since these programs are comparable to those provided by Magma in terms of program size, application domain, and vulnerability types included, we consider this a valid assumption.

7 RELATED WORK

The relevance of evaluating static code analyzers against real-world vulnerabilities is underlined by a project called OpenSSF CVE Benchmark [11], initiated by the Open Source Security Foundation (OpenSSF)[9] to facilitating a uniform comparison of static JavaScript analyzers. At the time of writing this paper, their benchmark includes three analyzers and around 200 vulnerabilities (CVEs).

The work of Zitser *et al.* [71] also evaluates the performance of several static C (and C++) analyzers. However, different from our study, they focus on buffer-overflow vulnerabilities and discuss the analyzers' false positive rates. In contrast, we analyses the extent of false negatives, thereby also considers a much wider range of vulnerability types. Also, we evaluate the static analyzers on real-world codebases, while Zitser *et al.* used synthetic programs (with a total of 14 vulnerabilities) due to limitations of the employed analyzers. Interestingly, although their work is over 15 years old, the detection rates have not improved much today.

Zheng *et al.* [70] compared three commercial static C/C++ analyzers on three Nortel network service software products using several metrics. Among others, they evaluated the defect detection rates of the analyzers, taking mainly into account non-security-related bugs. The false negative rates they report are thereby slightly lower than what we observed, but around the same order of magnitude. However, they neither include free and open-source (FOS) static analyzers nor benchmark programs outside the network domain, which may limit the generalizability of their results. In contrast, we use an automated approach to evaluate the effectiveness of six different static analyzers (FOS and commercial) in detecting vulnerabilities in 27 programs from different domains.

Chatzieleftheriou and Katsaros [28] conduct an evaluation of six static code analyzers, including two commercial ones, using a synthetic dataset targeted at common C/C++ vulnerabilities. Therein, they compare the tools individually, but not in combination, and present the analyzers' precision and recall scores. The only analyzer also found in our study is Cppcheck, which similarly to our results performs poorly compared to the other tools.

Goseva-Popstojanova and Perhinschi [40] assess the effectiveness of three commercial static analyzers for C/C++ and Java on the synthetic Juliet Test Suite [12] and two free and open-source C projects, containing 12 real-world vulnerabilities. In contrast, our evaluation also includes FOS analyzers and is performed on 27 real-world programs with a total of 192 vulnerabilities. Contrary to our observations, the commercial tools they chose show no significant performance difference. However, they also conclude that the performance of static code analyzer depends on the vulnerability type, with some CWEs being better detected than others.

Another related work is provided by D'abruzzo Pereira and Vieira in [32], in which they evaluate the effectiveness of Cppcheck and Flawfinder in detecting real-world vulnerabilities [14, 15]. Unlike our work, their study is limited to these two analyzers, because the used benchmark programs do not support tools that attach to the build process. Moreover, they count a vulnerability detected if the affected source files are marked. This can lead to overly optimistic results, as reflected by the detection rates of 83.5% (Cppcheck) and 36.2% (Flawfinder) that deviate from our numbers.

[9] https://openssf.org/

Kaur and Nayyar [45] conduct a similar empirical study as we do, with the difference that besides static C/C++ analyzers, they also examine tools for Java programs. Their set of analyzers also includes Flawfinder and Cppcheck, which are evaluated with respect to 10 different CWEs, some of which are also included in our benchmark. However, they use the synthetic Juliet Test Suite, while we evaluate the analyzers on real-world software projects with known vulnerabilities. According to their results, of the 118 vulnerabilities they targeted, Cppcheck outperforms Flawfinder, whereas the exact opposite holds true in our evaluation.

Thung *et al.* [65] conduct a study that is strongly related to our work, yet different in the sense that they evaluate three static Java analyzers to check their effectiveness on three large free and open-source programs. The benchmark programs contain 200 real-world software weaknesses, but not all of them manifest as security vulnerabilities. Interestingly, they also encountered software projects where all static analyzers combined were unable to detect 50% of the weaknesses in their benchmark.

Another study led by Habib and Pradel [41], who used three static Java analyzers on 15 real-world projects with a total of 597 bugs, found that as many as 95.5% of the defects were not detected. However, the results of Java analyzers are not necessarily transferable to C analyzers due to the different language constructs (*e.g.*, memory pointers) and the associated vulnerability types.

In sum, our work differs from the state of the art in terms of the considered (1) programming languages, (2) benchmark programs (FOS *vs.* commercial) and hence reproducibility, (3) static code analyzers, (4) nature of vulnerabilities (synthetic *vs.* real-world) & weakness categories (CWEs), and (5) detection code granularity (lines *vs.* functions *vs.* modules/files). Moreover, some related studies have been conducted more than a decade ago.

8 CONCLUSION AND FUTURE WORK

We evaluated the vulnerability detection capabilities of six state-of-the-art static C code analyzers against 27 free and open-source programs containing in total 192 real-world vulnerabilities (*i.e.*, validated CVEs). Our empirical study revealed that the studied static analyzers are rather ineffective when applied to real-world software projects; roughly half (47%, best analyzer) and more of the known vulnerabilities were missed. Therefore, we motivated the use of multiple static analyzers in combination by showing that they can significantly increase effectiveness; up to 21–34 percentage points (depending on the evaluation scenario) more vulnerabilities detected compared to using only one tool, while flagging about 15pp more functions as potentially vulnerable. However, certain types of vulnerabilities—especially the non-memory-related ones—seemed generally difficult to detect via static code analysis, as virtually all of the employed analyzers struggled finding them.

We consider this work as a basis for future research on the effectiveness of static code analysis for vulnerability detection. Here, we plan to investigate the underlying reasons as to why so many vulnerabilities could not be detected, even though they are supported by the respective analyzers. In doing so, we hope to not only find ways to improve them, but also to gain a better understanding of the general limitations of such tools.

9 DATA AVAILABILITY STATEMENT

We release all evaluation data and the analysis script [51] to replicate the results of this work and to encourage further studies on static code analysis.

ACKNOWLEDGMENTS

We would like to thank our colleagues Daniel Elsner and Markus Schnappinger, as well as the anonymous reviewers, for their valuable feedback.

REFERENCES

[1] [n. d.]. Clang Static Analyzer. https://clang-analyzer.llvm.org/. Accessed: 2021-07-23.
[2] [n. d.]. Clang-Tidy: Extra Clang Tools. https://clang.llvm.org/extra/clang-tidy/. Accessed: 2021-07-23.
[3] [n. d.]. CodeChecker. https://codechecker.readthedocs.io/en/latest/. Accessed: 2021-07-23.
[4] [n. d.]. CodeQL for Research. https://securitylab.github.com/tools/codeql/. Accessed: 2021-07-23.
[5] [n. d.]. The Common Weakness Enumeration (CWE) Initiative. https://cwe.mitre.org/. Accessed: 2022-01-19.
[6] [n. d.]. Cppcheck: A Tool for Static C/C++ Code Analysis. http://cppcheck.sourceforge.net/. Accessed: 2021-07-23.
[7] [n. d.]. CWE-Compatible Products and Services. https://cwe.mitre.org/compatible/compatible.html. Accessed: 2022-01-12.
[8] [n. d.]. Cyber Grand Challenge Corpus. http://www.lungetech.com/cgc-corpus/. Accessed: 2021-07-10.
[9] [n. d.]. Flawfinder. https://dwheeler.com/flawfinder/. Accessed: 2021-07-23.
[10] [n. d.]. Infer: A Tool to Detect Bugs in Java and C/C++/Objective-c Code. https://fbinfer.com/. Accessed: 2021-07-23.
[11] [n. d.]. Introducing the OpenSSF CVE Benchmark. https://openssf.org/blog/2020/12/09/introducing-the-openssf-cve-benchmark/. Accessed: 2021-08-13.
[12] [n. d.]. Juliet Test Suite. https://samate.nist.gov/SRD/testsuite.php. Accessed: 2021-07-10.
[13] Midya Alqaradaghi, Gregory Morse, and Tamás Kozsik. 2021. Detecting Security Vulnerabilities with Static Analysis — A Case Study. *Pollack Periodica* (dec 2021). https://doi.org/10.1556/606.2021.00454
[14] Henrique Alves, Baldoino Fonseca, and Nuno Antunes. 2016. Experimenting Machine Learning Techniques to Predict Vulnerabilities. In *Proceedings of the Latin-American Symposium on Dependable Computing*. 151–156. https://doi.org/10.1109/LADC.2016.32
[15] Henrique Alves, Baldoino Fonseca, and Nuno Antunes. 2016. Software Metrics and Security Vulnerabilities: Dataset and Exploratory Study. In *Proceedings of the European Dependable Computing Conference*. 37–44. https://doi.org/10.1109/EDCC.2016.34
[16] Andrei Arusoaie, Stefan Ciobaca, Vlad Craciun, Dragos Gavrilut, and Dorel Lucanu. 2018. A Comparison of Open-source Static Analysis Tools for Vulnerability Detection in C/C++ Code. In *Proceedings of the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, 161–168. https://doi.org/10.1109/SYNASC.2017.00035
[17] Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *Leibniz International Proceedings in Informatics*, Vol. 56. 1–25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2
[18] Algirdas Avižienis, Jean Claude Laprie, Brian Randell, and Carl Landwehr. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2004), 11–33. https://doi.org/10.1109/TDSC.2004.2
[19] Nathaniel Ayewah, David Hovemeyer, David J. Morgenthaler, John Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (2008), 22–29. https://doi.org/10.1109/MS.2008.130
[20] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 470–481. https://doi.org/10.1109/saner.2016.105
[21] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the Conference on Computer and Communications Security*. ACM, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020
[22] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* 45, 5 (may 2019), 489–506. https://doi.org/10.1109/TSE.2017.2785841
[23] Tim Boland and Paul E. Black. 2012. Juliet 1.1 C/C++ and Java Test Suite. *Computer* 45, 10 (oct 2012), 88–90. https://doi.org/10.1109/MC.2012.345
[24] Hudson Borges and Marco Tulio Valente. 2018. What's in a Github Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* 146 (2018), 112–129. https://doi.org/10.1016/j.jss.2018.09.016
[25] Joshua Bundt, Andrew Fasano, Brendan Dolan-Gavitt, William Robertson, and Tim Leek. 2021. Evaluating Synthetic Bugs. In *Proceedings of the Asia Conference on Computer and Communications Security*. 716–730. https://doi.org/10.1145/3433210.3453096
[26] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2019. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 209–224. https://doi.org/10.5555/1855741.1855756
[27] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. In *Journal of the ACM*, Vol. 58. New York, NY, USA. https://doi.org/10.1145/2049697.2049700
[28] George Chatzieleftheriou and Panagiotis Katsaros. 2011. Test-Driving Static Analysis Tools in Search of C Code Vulnerabilities. In *Proceedings of the International Computer Software and Applications Conference*. IEEE, 96–103. https://doi.org/10.1109/COMPSACW.2011.26
[29] Brian Chess and Gary Mcgraw. 2004. Static Analysis for Security. *IEEE Security and Privacy Magazine* 2, 6 (2004), 76–79. https://doi.org/10.1109/MSP.2004.111
[30] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the International Conference on Automated Software Engineering*. 332–343. https://doi.org/10.1145/2970276.2970347
[31] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the Symposium on Principles of Programming Languages*, Vol. Part F1307. 238–252. https://doi.org/10.1145/512950.512973
[32] Jose D'abruzzo Pereira and Marco Vieira. 2020. On the Use of Open-Source C/C++ Static Analysis Tools in Large Projects. In *Proceedings of the European Dependable Computing Conference*. 97–102. https://doi.org/10.1109/EDCC51268.2020.00025
[33] Aurelien Delaitre, Bertrand Stivalet, Elizabeth Fong, and Vadim Okun. 2015. Evaluating Bug Finders: Test and Measurement of Static Code Analyzers. In *Proceedings of the International Workshop on Complex Faults and Failures in Large Software Systems*. IEEE, 14–20. https://doi.org/10.1109/COUFLESS.2015.10
[34] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 110–121. https://doi.org/10.1109/SP.2016.15
[35] Anum Fatima, Shazia Bibi, and Rida Hanif. 2018. Comparative Study on Static Code Analysis Tools for C/C++. In *Proceedings of the International Bhurban Conference on Applied Sciences and Technology*, Vol. 2018-Janua. IEEE, 465–469. https://doi.org/10.1109/IBCAST.2018.8312265
[36] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proceedings of the USENIX Workshop on Offensive Technologies*.
[37] Sijia Geng, Yuekang Li, Yunlan Du, Jun Xu, Yang Liu, and Bing Mao. 2020. An Empirical Study on Benchmarks of Artificial Software Vulnerabilities. (mar 2020). arXiv:2003.09561 http://arxiv.org/abs/2003.09561
[38] Christoph Gentsch. 2020. *Evaluation of Open Source Static Analysis Security Testing (SAST) Tools for C*. Technical Report. German Aerospace Center (DLR DW). 37 pages. https://elib.dlr.de/133945/
[39] Anjana Gosain and Ganga Sharma. 2015. Static Analysis: A Survey of Techniques and Tools. *Advances in Intelligent Systems and Computing* 343 (2015), 581–591. https://doi.org/10.1007/978-81-322-2268-2_59
[40] Katerina Goseva-Popstojanova and Andrei Perhinschi. 2015. On the Capability of Static Code Analysis to Detect Security Vulnerabilities. *Information and Software Technology* 68 (dec 2015), 18–33. https://doi.org/10.1016/j.infsof.2015.08.002
[41] Andrew Habib and Michael Pradel. 2018. How Many of All Bugs Do We Find? A Study of Static Bug Detectors. In *Proceedings of the International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 317–328. https://doi.org/10.1145/3238147.3238213
[42] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2021. Magma: A Ground-Truth Fuzzing Benchmark. *Proceedings of the International Conference on Measurement and Modeling of Computer Systems* 4, 3 (2021), 81–82. https://doi.org/10.1145/3410220.3456276 arXiv:2009.01120
[43] Nasif Imtiaz, Akond Rahman, Effat Farhana, and Laurie Williams. 2019. Challenges with Responding to Static Analysis Tool Alerts. In *Proceedings of the International Working Conference on Mining Software Repositories*. IEEE, 245–249. https://doi.org/10.1109/MSR.2019.00049
[44] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the International Conference on Software Engineering*. 672–681. https://doi.org/10.1109/ICSE.2013.6606613
[45] Arvinder Kaur and Ruchikaa Nayyar. 2020. A Comparative Study of Static Code Analysis Tools for Vulnerability Detection in C/C++ and JAVA Source Code. In

Procedia Computer Science, Vol. 171. 2023–2029. https://doi.org/10.1016/j.procs.2020.04.217

[46] Sunghun Kim and Michael D. Ernst. 2007. Which Warnings Should I Fix First?. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. 45–54. https://doi.org/10.1145/1287624.1287633

[47] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[48] Ted Kremenek and Dawson Engler. 2003. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. *Lecture Notes in Computer Science* 2694 (2003), 295–315. https://doi.org/10.1007/3-540-44898-5_16

[49] William Landi. 1992. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems* 1, 4 (1992), 323–337. https://doi.org/10.1145/161494.161501

[50] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[51] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. *Artifacts for the ISSTA 2022 Paper: An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection*. https://doi.org/10.5281/zenodo.6515687

[52] V Benjamin Livshits and Monica S Lam. 2005. Finding Security Errors in Java Programs with Static Analysis. In *Proc. Usenix Security Symposium*. 271–286.

[53] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. BugBench: Benchmarks for Evaluating Bug Detection Tools. In *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*. 1–5. https://doi.org/10.1.1.134.8941

[54] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the Reproducibility of Crowd-reported Security Vulnerabilities. In *Proceedings of the USENIX Security Symposium*. 919–936.

[55] Tukaram Muske, Rohith Talluri, and Alexander Serebrenik. 2018. Repositioning of Static Analysis Alarms. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, 187–197. https://doi.org/10.1145/3213846.3213850

[56] Vadim Okun, Aurelien Delaitre, and Paul E Black. 2011. *Report on the Static Analysis Tool Exposition (SATE) IV*. Technical Report. https://doi.org/10.6028/NIST.SP.500-297

[57] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic Execution with SymCC: Don't Interpret, Compile!. In *Proceedings of the USENIX Security Symposium*.

[58] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the Symposium on Logic in Computer Science*. IEEE Comput. Soc, 55–74. https://doi.org/10.1109/lics.2002.1029817

[59] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. 2008. Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach. In *Proceedings of the International Conference on Software Engineering*. 341–350. https://doi.org/10.1145/1368088.1368135

[60] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (2018), 58–66. https://doi.org/10.1145/3188720

[61] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. 2015. Test Suites for Benchmarks of Static Analysis Tools. In *Proceedings of the International Symposium on Software Reliability Engineering Workshops*. IEEE, 12–15. https://doi.org/10.1109/ISSREW.2015.7392027

[62] Darko Stefanović, Danilo Nikolić, Dušanka Dakić, Ivana Spasojević, and Sonja Ristić. 2020. Static Code Analysis Tools: A Systematic Literature Review. In *Proceedings of the International Symposium on Intelligent Manufacturing and Automation*, Vol. 31. 565–573. https://doi.org/10.2507/31st.daaam.proceedings.078

[63] Wouter Stikkelorum. 2016. *Challenges of Using Sound and Complete Static Analysis Tools in Industrial Software*. mathesis. University of Amsterdam. https://scripties.uba.uva.nl/scriptie/618182

[64] Patrick Thomson. 2022. Static Analysis. *Commun. ACM* 65, 1 (jan 2022), 50–54. https://doi.org/10.1145/3486592

[65] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T. Devanbu. 2012. To What Extent Could We Detect Field Defects? An Empirical Study of False Negatives in Static Bug Finding Tools. In *Proceedings of the International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 50–59. https://doi.org/10.1145/2351676.2351685

[66] Stephen Turner. 2014. Security Vulnerabilities of the Top Ten Programming Languages: C, Java, C++, Objective-C, C#, Php, Visual Basic, Python, Perl, and Ruby. *Journal of Technology Research* 5 (2014), 1–16.

[67] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C. Gall. 2018. Context Is King: The Developer Perspective on the Usage of Static Analysis Tools. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 38–49. https://doi.org/10.1109/SANER.2018.8330195

[68] Andreas Wagner and Johannes Sametinger. 2014. Using the Juliet Test Suite to Compare Static Security Scanners. In *Proceedings of the International Conference on Security and Cryptography*. SCITEPRESS, 244–252. https://doi.org/10.5220/0005032902440252

[69] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the USENIX Security Symposium*. 745–761. https://doi.org/10.5555/3277203.3277260

[70] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl, and Mladen A. Vouk. 2006. On the Value of Static Analysis for Fault Detection in Software. *IEEE Transactions on Software Engineering* 32, 4 (apr 2006), 240–253. https://doi.org/10.1109/TSE.2006.38

[71] Misha Zitser, Richard Lippmann, and Tim Leek. 2004. Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. ACM, New York, NY, USA, 97. https://doi.org/10.1145/1029894.1029911