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:

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, grey-box 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 analyzers 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, 783} are more effectively detected than those of CWE–{682, 787, 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, 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 0.14.02, 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

\footnote{Symbolic execution [47] and abstract interpretation [31] also fall into this category, but were not considered in this study.}
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] (0.15k) 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] (0.1k) 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 subclassification 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 Flawfinder, 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].
Vulnerability Validation. Since many of the employed analyzers attach themselves to the build process, we checked\(^3\) 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\(^4\) 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.

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 refers to 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.

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.

Since many of the employed analyzers vulnerability validation. This table is supplemented by Fig. 2, showing that except for SQLite3, the arithmetic mean of lines of code of the functions affected\(^3\) by one or more vulnerabilities is below 100 LoC.

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 specified 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\(^3\) by one or more vulnerabilities is below 100 LoC.

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 refers to 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.

Since many of the employed analyzers vulnerability validation. This table is supplemented by Fig. 2, showing that except for SQLite3, the arithmetic mean of lines of code of the functions affected\(^3\) by one or more vulnerabilities is below 100 LoC.

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 specified 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\(^3\) 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\(^4\) 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\(^5\) 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.

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 refers to 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.
As mentioned before, the benchmark dataset contains whether a vulnerability was detected by a static analyzer, or not. 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.

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-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.

### Table 2: Supported Vulnerability Classes

<table>
<thead>
<tr>
<th>Vulnerability Class</th>
<th>Analyzer</th>
</tr>
</thead>
<tbody>
<tr>
<td>CWE-664</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>CWE-682</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>CWE-703</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>CWE-707</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>CWE-691</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
</tbody>
</table>

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 an 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-128: "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.

```c
char *xmlMemStrdupLoc(const char *str, const char *file, int line)
{
    char *s;
    size_t size = strlen(str) + 1;
    memcpy(&p, (xmlMemInitialized) malloc(REVERSE_SIZE + size));
    if (!p goto error;
    p->mh_tag = MENTAG;
    p->mh_size = size;
    p->mh_type = STRUP_TYPE;
    p->mh_file = file;
    p->mh_line = line;
    xmlMutexLock(xmlMemMutex);
    p->mh_number = ++block;
    xmlMutexUnlock(xmlMemMutex);
    /* #define CLIENT_2_HDR(a) (((void *) (((char *)(a)) -
                                REVERSE_SIZE)) */
    z = (char *) CLIENT_2_CLIENT(p);
    strcpy(s, z);
    return(s);
    error:
    return(NULL);
}
```

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

\[ \text{Fault-Error Conformity (FEC)} = \frac{|F_{\text{fault}} \cap F_{\text{error}}|}{|F_{\text{error}}|} \]

which for a given vulnerability (CVE) computes the ratio of error-containing functions (F_{\text{error}}) that also include the underlying fault (F_{\text{fault}}). Accordingly, the higher the FEC ratio, the more fault and error locations of a vulnerability overlap within the same function(s).
Scenario 2 (S2-2): A vulnerability is considered detected if all affected functions were marked by the static analyzer together with the correct vulnerability class.

Scenario 4 (S2-4): A vulnerability is considered detected if all affected functions were marked by the static analyzer together with the correct vulnerability class.

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. 3 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 frontported 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 S1-1-2, while flagging 11pp less functions. This shows that semantic analysis methods are not always more effective than the less complex syntactic ones.

**Summary (RQ1).** Our empirical evaluation shows that state-of-the-art static 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% (S1-1), 70% (S1-2), 64% (S2-1), and 80% (S2-2) 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 S1-1-2 and by CodeChecker in S1-2-2. 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 S1-1 34 (17pp) and in S2-1 30 (16pp) more vulnerabilities than the best single static analyzer (CommSCA), while marking 15pp more functions. In S1-2 and S2-2, Flawfinder-Cppcheck-Infer-CodeQL-CommSCA outperforms CommSCA with 24 (13pp) and 21 (1pp) more vulnerabilities found, again with 15pp more flagged functions. Interestingly, the best combination detects in S1-1, S2-1, and S2-2 more than twice as many security bugs as CodeQL, however, at the cost of flagging roughly double the number of functions. For scenario S1-2, 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 (RQ2).** 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% (S1-1), 57% (S1-2), 43% (S2-1), and 69% (S2-2) of the 192 vulnerabilities.

---

\(^8\) [https://www.semmle.com/](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 S1-1 and S2-1 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 S1-2 and S2-2. 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 S1-1 (78%) and S2-1 (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, S1-2 and S2-2, 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
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.

**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)\(^6\) 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 analyze the extent of false negatives, thereby also considering 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 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’abruzzi 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.

\(^6\)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 CVEs, 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 (CVEs), 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 15% 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
