A Comparative Study of Fuzzers and Static Analysis Tools for Finding Memory Unsafety in C and C++

KENO HASSLER* and PHILIPP GÖRZ*, CISPA Helmholtz Center for Information Security, Germany
STEPHAN LIPP, Technical University of Munich, Germany
THORSTEN HOLZ, CISPA Helmholtz Center for Information Security, Germany
MARCEL BÖHME, Max Planck Institute for Security and Privacy, Germany

Even today, over 70% of security vulnerabilities in critical software systems result from memory safety violations. To address this challenge, fuzzing and static analysis are widely used automated methods to discover such vulnerabilities. Fuzzing generates random program inputs to identify faults, while static analysis examines source code to detect potential vulnerabilities. Although these techniques share a common goal, they take fundamentally different approaches and have evolved largely independently.

In this paper, we present an empirical analysis of five static analyzers and 13 fuzzers, applied to over 100 known security vulnerabilities in C/C++ programs. We measure the number of bug reports generated for each vulnerability to evaluate how the approaches differ and complement each other. Moreover, we randomly sample eight bug-containing functions, manually analyze all bug reports therein, and quantify false-positive rates. We also assess limits to bug discovery, ease of use, resource requirements, and integration into the development process. We find that both techniques discover different types of bugs, but there are clear winners for each. Developers should consider these tools depending on their specific workflow and usability requirements. Based on our findings, we propose future directions to foster collaboration between these research domains.

1 INTRODUCTION

In recent years, more and more security vulnerabilities have been reported in critical software systems. Today, many of these vulnerabilities are due to memory unsafety. For instance, a third of *all* recorded security vulnerabilities (tracked as CVEs) and seven of the 20 most frequently reported bug classes (tracked as CWEs) are related to memory safety violations. More than 70% of reported vulnerabilities in the Chrome browser [77] as well as in the iOS, macOS [44], and Android [80] operating systems [21] are due to memory unsafety. In fact, 80% of vulnerabilities exploited in the wild are due to memory unsafety [1].

To scale bug finding beyond manual code audits, developers have turned to automated vulnerability discovery tools based on techniques such as static analysis and fuzz testing (in short *fuzzing*). For instance, Google considers fuzzing their first line of defense [78], while Meta reports finding 70% of vulnerabilities using static analysis [67]. Illustrating the popularity of fuzzing, the OSS-Fuzz project [75] continuously tests more than 1,000 open source software (OSS) projects, including many popular and security-critical projects. Recently, fuzzing has become a first-class citizen in the Go programming language [25] and has been integrated into the widely-used Visual Studio Code development environment [57]. Regarding the popularity of static analysis, Beller *et al.* [3] found back in 2016 that static analysis is used in about half of the 122 surveyed OSS projects. Indeed, a static analysis tool, CodeQL is a prominent part of GitHub's Security Lab, which has the goal of securing open-source software [23].

*Both authors contributed equally to this research.

Authors' addresses: Keno Hassler, keno.hassler@cispa.de; Philipp Görz, CISPA Helmholtz Center for Information Security, Saarbrücken, Germany, research@philipp-goerz.com; Stephan Lipp, Technical University of Munich, Munich, Germany, stephan.lipp@tum.de; Thorsten Holz, CISPA Helmholtz Center for Information Security, Saarbrücken, Germany, holz@cispa.de; Marcel Böhme, Max Planck Institute for Security and Privacy, Bochum, Germany, marcel.boehme@acm.org.

Fuzzing detects bugs by provoking unexpected program executions, while static analysis examines the program's source code. In a *fuzzing setup*, a fuzzer passes automatically generated inputs to the program under test via a *harness*—a fuzzer-specific entry point that is usually written manually. During compilation, the harness is linked to the program, and coverage instrumentation as well as sanitizer instrumentation are added via a custom compiler pass. In a *static analysis setup*, only the static analyzer is required, though in some cases the program needs to be compiled. Unlike fuzzers, static analyzers are known to report false positives, i.e., bugs that do not actually exist. But how do fuzzing and static analysis compare in their effectiveness at detecting real bugs?

Fuzzing and static analysis share the fundamental goal of finding bugs in programs, yet there has been little research comparing these two approaches. This can be explained by two factors: first, these tools are developed by largely separate communities; second, fuzzing and static analysis tools require fundamentally different project integrations, essentially requiring extra work to integrate the tools. These differences make a systematic comparison of both approaches particularly interesting. Is it worth the effort to employing both approaches in parallel, and can the approaches benefit from learning from each other?

In our work, we aim to bring both communities closer together, fostering collaboration to develop improved methods for finding software faults. In this article, we adopt the perspective of an open-source C/C++ project maintainer who wants to use a bug-finding tool to discover memory safety vulnerabilities. Currently, there are no clear guidelines for choosing one approach over the other. There is also no systematic analysis of their relative strengths and weaknesses. To develop such a systematic analysis, we define clear selection criteria and evaluate over 100 CVEs across seven open-source projects (using Magma [32]), employing five static analyzers and 13 different fuzzers.

Outline. We start with an overview of the relevant background and related work (Section 2). After defining the scope of our study (Section 3), we conduct our analysis in two parts. The first part is an *empirical evaluation* (Section 4) of freely available fuzzers and static analysis tools focusing on four research questions:

- *RQ.1 True Positives.* Analyzing the number of CVEs each approach can detect, we found that fuzzers find 43 of the expected CVEs and static analysis tools find 40. Among the analyzers, the semantic approach found more bugs (true positives) than the syntactic approach.
- *RQ.2 Complementarity*. We found that both approaches find different bugs, suggesting that they complement each other and should be used together. Interestingly, we found that different fuzzers find approximately the same CVEs. Static analyzer results are dominated by CodeQL, indicating a large divide between tool maturity.
- *RQ.3 Bug types.* This result is also reflected in our analysis of CVEs detected by types of vulnerability: while fuzzers perform similarly across all types of CVEs, syntactic analyzers only really work on memory corruption vulnerabilities, while semantic analyzers perform well on most types of vulnerabilities.
- *RQ.4 Overhead*. However, when analyzing the manually required overhead, we found that less than 1% of the generated bug reports were related to the CVEs that we expected to find. Overall, to find all 40 CVEs that static analysis could successfully detect, one would have to go through a total of 44K bug reports.

In the second part, we conduct a *qualitative analysis* (Section 5), based on the empirical part. We take the perspective of a code maintainer who decides on bug finding tooling for their project. Concretely, we assess the following aspects:

- *Manual validation.* We conduct a manual analysis of 49 static analysis-generated bug reports in 8 randomly chosen functions containing known CVEs. Only 8 of these 49 bug reports (16%) actually point to expected vulnerabilities. The other 41 reports are unrelated or false positives.[1]
- *Limits to Bug Discovery*. We examine the limits of both techniques based on concrete examples from the empirical analysis. We attribute their different limitations to their fundamentally different approaches and highlight that it is unrealistic to expect *one* tool that finds all bugs.
- *Usability and Cost*. Static Analyzers require less effort to set up than writing integration code for fuzzers; furthermore, they are significantly cheaper per bug found. Both approaches offer opportunities for fine-tuning.
- *Integration in the Development Process.* We look at different ways to integrate bug finding tools in the development pipeline and check on their real-world usage. Our analysis indicates that 80% of security-critical OSS projects already use automated bug finding for security.

We discuss the implications of our analysis for project maintainers and research communities in Section 6 and conclude the article in Section 7.

Data Availability. To make our findings reproducible and foster further research in this direction, we make our evaluation scripts and data openly available at doi:10.5281/zenodo.15516474.

2 BACKGROUND

In this section, we explain the types of vulnerabilities we are interested in, before briefly summarizing the state-of-the-art in bug finding tools and the benchmarks commonly used to evaluate them.

2.1 Memory Unsafety in C/C++

To focus our research, we limit our investigation to one of the most prevalent classes of security vulnerabilities: memory unsafety in C/C++ programs and related types of vulnerabilities. As discussed in the introduction, more than 70% of the reported vulnerabilities in Chrome, iOS, macOS, and Android are related to memory unsafety [77, 44, 80]; 80% of the security vulnerabilities exploited in the wild are due to memory safety violations [1]. Software systems can be subject to many types of bugs and vulnerabilities. However, memory safety violations are (almost) unique to C/C++ languages. While many modern languages offer memory safety by default, in C/C++ it is primarily the developer's responsibility to uphold the safety contract. In exchange, C/C++ offers more low-level control and (in absence of runtime checkers, such as a garbage collector) a high performance.[2]

To be specific, we exclude bugs that are unlikely to be security-critical, such as unused variables found by the compiler or violations of coding conventions reported by a linter. Furthermore, bugs in other languages, like gadget chains in Java, are not relevant for our analysis. Finally, security-critical bugs that are not related to memory safety violations, such as side-channel attacks or information leaks, are also considered out of scope. This means we are evaluating the bug-finding tools from a security perspective: while we touch upon this aspect in Section 5, we do *not* consider bug finding *primarily* as a tool to improve software quality.

2.2 Bug Finding: Static vs. Dynamic Methods

To focus our discussion, we limit our investigation to two prevalent automated bug-finding approaches, fuzzing and static analysis.

[1] We note that our conservative labeling approach employed in the first part would count 36 of these reports as correct.
[2] Technically, other languages exist that promise a similar level of efficiency while maintaining memory safety guarantees.

Static Analysis. Static Application Security Testing (SAST) refers to a collection of techniques that examine source code without actually executing the code. Notably, this always requires some degree of (over-)approximation: At the coarse end of the spectrum lies syntactic analysis, which is fundamentally pattern matching with known or suspicious antipatterns. This approach is easy to implement and inexpensive to run. A more sophisticated approach is symbolic execution, a computationally complex technique that simulates program paths by reasoning about symbolic states. The trade-off for this approximation is a varying number of false positive reports, i.e., reports that flag a piece of code as defective although it is in fact correct.

Fuzzing. Fuzzing [58] is a dynamic testing technique. The program under test (PUT) is executed on automatically generated, concrete test inputs. If an illegal state is reached, the fuzzer is notified (by crashing the PUT) that it has found a bug and saves the respective input to disk. While it is left to a human analyst to find the root cause of this bug, all reports did cause a crash and are thus true positives. Detection of illegal states is performed by sanitizers (cf. Song *et al.* [76]) that typically under-approximate, and thus, do not produce false positive reports. Since the advent of AFL [85], greybox fuzzers that use program coverage as inexpensive feedback loop have become state-of-the-art. The now-deprecated AFL has found large numbers of bugs itself [85], and its descendant AFL++ [20], which incorporates more recent research ideas, continues on that path.

2.3 Related Work

Both static analysis and fuzzing are active research domains, this includes extensive research on how tools should be evaluated. However, to the best of our knowledge, no cross-domain comparison has been attempted so far. In the following, we give an overview of existing work.

Static Analysis. Sadowski *et al.* [69] report on their experiences with the development of static analysis tools at Google. They suggest that tool authors should focus on the real needs of developers so that the tool can be integrated directly into the development workflow to find bugs early. They also suggest that tools should be made open-source so that they can better keep pace with future bug-finding challenges. Obviously, this also requires constant evaluation of new and established static analyzers to see how well they actually find (security) bugs.

Johnson *et al.* [41] find in a small survey that a lack of information in warning messages given to developers inhibits more wide-spread adoption. Christakis *et al.* [10] confirm this in a larger study, adding that developers expect a false-positive rate below 20% and the ability to configure the tool. A data scrape on `stackoverflow.com` by Imtiaz *et al.* [36] supports these results, calling false positives a "dominant issue" for the adoption of static analysis tools. Consequently, many research papers [47, 52, 45, 68, 2, 82, 60] have attempted to reduce the number of false positive warnings. Besides the false positive analyses, numerous papers [88, 87, 8, 79, 29, 30, 15, 43] evaluate the effectiveness of such analyzers in terms of how many bugs were found (true positives) versus how many were missed (false negatives), across different benchmarks. Closely related to our work, Lipp *et al.* [50] evaluate SAST tools on memory vulnerabilities, finding that the majority of bugs in the benchmark remain undetected by state-of-the-art tools. In this study, we go a step further and compare the bug-finding capabilities of SAST tools *with those of fuzzers*, aiming to find differences and potential benefits from combining these tools.

Fuzzing. Fuzzing has been deployed in large-scale evaluations, notably in Google's OSS-Fuzz [75], an open-source software fuzzing service that continuously tests hundreds of popular projects. Ding *et al.* [17] analyze more than twenty thousand bugs, including thousands of severe bugs, found via OSS-Fuzz, demonstrating its real-world impact. Furthermore, they underline that effective fuzzing is an iterative process that requires fixing detected bugs, allowing the campaign to progress. Google's Fuzzbench [56] framework offers a free benchmark service for fuzzer developers

running on Google infrastructure. The Magma benchmark [32] hand-picks a set of real-world bugs and provides perfect oracles for them, aiming to remove inaccuracies in comparisons. Böhme *et al.* have shown that code coverage is strongly correlated with the number of bugs found [7], but discovering new bugs becomes exponentially harder [4].

3 SCOPE OF COMPARATIVE STUDY

Our research is motivated by the common situation where a maintainer of a C/C++ project decides to add a bug-finding tool to discover memory safety vulnerabilities in their project. However, to assess a tool's performance for this task systematically, we first need to find a suitable benchmark.

Benchmark selection criteria. To provide a fair comparison between fuzzers and SAST tools, we need a ground-truth dataset with labeled bugs, including the type of each bug. We require that the bugs are security-critical and found in C or C++ programs. For generalizability, the bugs should capture different real-world application domains, different input structures, and a variety of operations and transformations. Finally, and crucially, we want to avoid using one tool group's output as "ground truth" for the other, as that would inherently place an upper bound on the other group's performance. Consequently, we require a benchmark containing bugs that were found *irrespective* of the technique (i.e., fuzzing or SAST) used to find these bugs.

Benchmark. The Magma benchmark (v1.1) [32] consists of 112 CVEs in seven open-source programs that range from image parsers and a cryptographic library to database and website engines. The (publicly known) bugs were re-inserted into a newer version of the respective code base. Most importantly, these bugs were chosen *irrespective* of the technique used to find them.[3] Thus, Magma fulfills our selection criteria. Note that we are aware of the possibility of an implicit selection bias in Magma due to the usage of bug-finding tools in the real world that influences the distribution of known vulnerabilities. But given that it is impossible to reason about the distribution of vulnerabilities that are yet undiscovered, and that the aforementioned bias would inherently be included in every (even hand-crafted) real-world dataset, we argue that Magma is the best choice for this purpose.

However, to make our evaluation as comprehensive as possible, we also consider other benchmarks. Unfortunately, benchmarks with synthetic bugs are not as well-suited for our evaluation, as the bug types are unknown, the bugs are not necessarily security-critical, or the programs are not realistic. For example, the Juliet benchmark [61] is a popular benchmark for static analysis tools. However, this benchmark is very artificial and does not resemble realistic programs, as most examples crash with a simple execution or do not take any input, making them unsuitable for our evaluation. The only other benchmark that satisfies our necessary selection criteria are the DARPA Cyber Grand Challenge (CGC) [31] binaries. This benchmark consists of custom-written, small programs containing exploitable vulnerabilities, however, they are thus less realistic than Magma. Therefore, to expand our evaluation, we also evaluate a subset of the tools on the CGC binaries. We reflect more on the availability of bug-based benchmarks for SAST tools in Section 6.2.

3.1 Tool Selection Criteria

To evaluate the discovery of memory safety vulnerabilities in C/C++ programs, we define selection and exclusion criteria for the tools in our evaluation.

[3]Regarding the collation of bugs, the Magma authors write: "No specific set of criteria was imposed on the bug selection process. However, throughout our porting efforts, we often prioritized more recent bug reports, since they correlate most closely to the latest code base, and are thus more likely to remain valid. Additionally, reports marked 'critical' were also given a higher priority than others." [33]

Table 1. Vulnerability Type Names and Their Description*

Codename	Description
Except	Improper Check or Handling of Exceptional Conditions
Data	Improper Neutralization (of Data)
Bounds	Improper Restrictions of Operations to a Memory Buffer
Math	Incorrect Calculations
Resource	Lifetime (of Memory and other Resources) and Type Errors
Logic	Logic Errors
Style†	Improper Adherence to Coding Standards
Other†	Various Other Errors

* We provide a list of all CWEs for each vulnerability type in Appendix B (Table 6).
† These vulnerability types are included for completeness, but are not central to our analysis.

Table 2. Static Analyzers in Our Evaluation

Analyzer	Version	License	Technique	✎	🔧	Vulnerability Classes
Clang SA	12.0.0	Apache2	Control/Data-flow analysis	✗	✓	Data, Except, Math, Resource, Smell
CodeQL	2.12.0	Proprietary	Control/Data-flow analysis	✓	✓	Bounds, Data, Except, Logic, Math, Other, Resource, Smell
Flawfinder	2.0.19	GPLv2	Syntactic analysis	✗	✗	Bounds, Data, Logic, Math, Other, Resource, Smell
Infer	1.1.0	MIT	Formal reasoning	✗	✓	Bounds, Except, Math, Other, Resource, Smell
SemGrep	1.24.0	LGPLv2.1	Syntactic analysis	✓	✗	Data, Other, Resource, Smell

✎ Customizable (decoupled rule set) 🔧 Depends on build process

In terms of selection criteria, we are interested in popular, state-of-the-art tools [81, 34] that can find memory safety violations in C/C++ programs, are open-source (or at least freely available), and cover a wide variety of approaches. We focus on tools that are widely used in practice, such as those used by large tech companies or having many GitHub stars. We exclude tools that report code quality issues rather than security-related issues, and tools that only have experimental support for memory safety issues. Furthermore, we exclude tools that require extra effort to set up (like grammar specifications for smart fuzzers), that are outdated or lack sufficient documentation for a reasonable setup, or that are incompatible with our experimental infrastructure (e.g., operating system and hardware).

Static analysis tools. Table 2 shows the analyzers that satisfied our criteria. We distinguish syntactic and semantic analyzers:

- *Syntactic (source code).* Flawfinder [83] encodes well-known vulnerability patterns as regular expressions that can be efficiently and most generally matched on the source code for vulnerability detection. SemGrep [71] additionally takes the semantics of a programming language into account, thereby reducing the number of false positives while keeping the analysis independent of the build system. Moreover, SemGrep decouples its rule set [72] from the analysis engine and provides a simple YAML interface for custom rules.
- *Semantic (logic).* Infer [18] and CodeQL [23] derive the computational model from the source code by interpreting the program's instructions. Infer is developed by Meta and uses separation logic to reason about heap-based pointer

structures in C/C++ programs. CodeQL is developed by GitHub and uses a deductive database to store program facts. Vulnerability patterns are encoded as declarative queries to the database. Given these reasoning capabilities, we expect the lowest false positive rate and the highest analysis time.

- *Semantic (symbolic execution).* Clang SA [53] follows a symbolic execution-based approach. Code fragments are symbolically executed to construct a symbolic state. It checks whether there is a satisfiable path to a dangerous state where security checks are violated. Clang SA aims at undefined behavior and dangerous code constructs, but can be extended with other types of checkers [11].

Except for Flawfinder and SemGrep, all analyzers need to be integrated into the program's build process. Apart from CodeQL and SemGrep, all analyzers have a fixed set of rules that may be enabled or disabled. CodeQL and SemGrep support the definition and addition of new detection rules.

Fuzzing tools. To get the broadest possible picture, we use all fuzzers available in Magma version 1.1. These are: AFL [85], AFLFast [6], AFL++ (3.00a) [20], Angora [9], Entropic [5], FairFuzz [48], Honggfuzz [26], LibFuzzer [74], MOpt-AFL [54], ParmeSan [62] and SymCC-AFL [66]. We use the pre-defined settings for these fuzzers and no external sanitizers. Additionally, we ported a more recent version of AFL++ [20] (4.08c) and libafl_libfuzzer [14] (version 0.13.1), a libAFL-based replacement for the now-deprecated libFuzzer. For AFL++, we are referencing the more recent version unless specified otherwise.

4 EMPIRICAL EVALUATION

We begin our evaluation by quantitatively comparing the selected static analysis (SAST) and fuzzing tools in terms of their bug-finding capabilities, specifically focusing on true positives. More specifically, we address the following research questions:

RQ.1 **True Positives**: How many CVEs in the Magma and CGC benchmarks are identified by each bug-finding tool and approach?

RQ.2 **Complementarity**: Do static analysis and fuzzing complement each other, both within and across approaches, or do they essentially identify the same set of bugs?

RQ.3 **Bug Types**: Are static analysis tools more effective at finding certain types of bugs compared to fuzzers, and vice versa?

RQ.4 **Overhead**: How much manual effort does a tool impose on a developer to extract actionable results (e.g., related to false positives or deduplication of findings)?

4.1 Experimental Design

The Magma benchmark includes *canaries*, a mechanism to detect if a CVE is triggered during runtime. This enables fuzzer evaluation without relying on potentially biased assertions or sanitizers. For evaluating static analysis tools, we restore the vulnerable code to its original state by removing these canaries.

The CGC benchmark does not have canaries, but instead provides vulnerability descriptions that include CWEs and their specific locations.

SAST Configuration. SAST tools offer a variety of configuration options that enable or disable specific checks, allowing users to fine-tune the analyzer for particular objectives (e.g., finding vulnerabilities instead of code smells). We configure each tool to optimize its performance and also enable all security-relevant options. Since code quality measurements

are outside the scope of this work, we disable code smell reporting where possible. For verifiability, we include this configuration mapping in our artifact.

Measuring false negatives. In this empirical evaluation, we can rely on automated tooling to evaluate the false *negative* rate, i.e., the number of expected CVEs found. However, we *cannot* reliably evaluate the false *positive* rate, i.e., the number of invalid bug reports per tool. To quantify this aspect of effort nevertheless, we empirically study the number of bug reports generated per every expected CVE (RQ2), and we manually classify all 49 SAST-generated bugs reported against eight randomly selected functions with known CVEs in Section 5.1.

Identifying whether a CVE is discovered. For fuzzing, we use the benchmark-provided canaries in Magma and manual crash inspection in CGC, as there are no canaries, to identify discovered CVEs.

For static analysis, we take a different approach since static analyzers may report unrelated bugs in functions containing known CVEs. For example, a tool might report a null pointer dereference when we expect a buffer overwrite. Following Lipp *et al.* [50], we identify detected CVEs by comparing the reported vulnerability type (CWE) with the expected type. We define expected CWEs based on Magma's provided data and CGC's vulnerability descriptions. We consider a CVE detected if the types match exactly or are sufficiently similar.[4] For example, if we expect an improper validation of array index (CWE-129), we accept reports of improper input validation (CWE-20) or out-of-bound reads (CWE-125).

Setup and Infrastructure. For fuzzing, we use Magma's (version 1.1) captain script to run campaigns. We rerun the Magma experiments instead of using data from the paper [32] since version 1.0 contained unreachable bugs that were removed in later versions [50]. Following recommendations from Klees *et al.* [46] and Schloegel *et al.* [70], each fuzzer runs for *48 hours* with *10 trials*. We double the corpus *tmpfs* size to 100 GiB and restrict workers to physical cores to avoid SMT-related variations. The experiments run on seven identical machines with 52-core Intel Xeon Gold 6230R CPUs and 188 GiB RAM. All machines use Ubuntu 22.04 LTS with kernel version 5.15.

For static analysis, we run the experiments in a container with an Intel Xeon Gold 6248R CPU using 16 cores at 3.0 GHz and 128 GiB RAM. The container uses Ubuntu 20.04 and Linux kernel version 6.5.

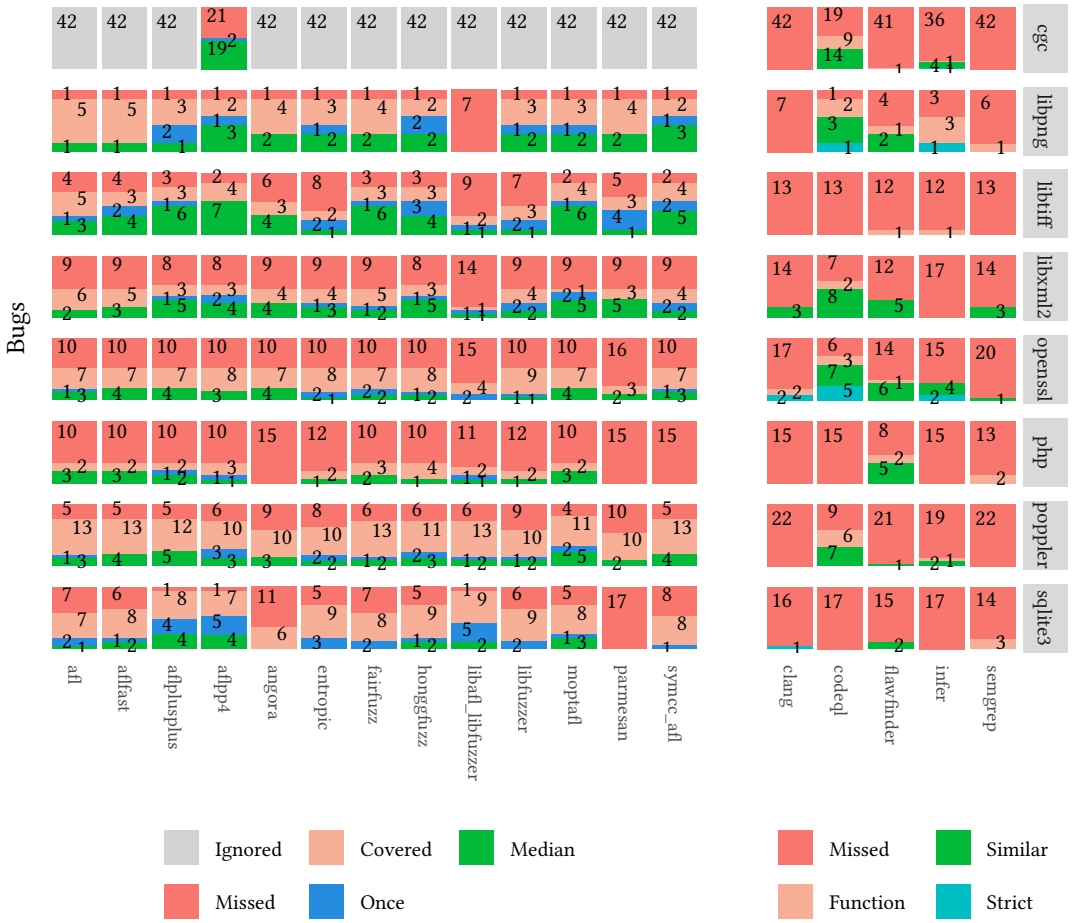
4.2 RQ.1 True Positives

We evaluate each fuzzer using ten 48-hour trials on the seven Magma programs. For CGC, we only evaluate AFL++ (aflpp4) with ten 48-hour trials due to the required manual verification of results. Figure 1a shows the number of bugs that were *Covered*, found at least *Once*, found in at least six trials (*Median*), or *Missed* across all runs. We use *Once* for RQ2 and RQ3 to assess the tools' potential performance, while *Median* is used in RQ4 to evaluate expected real-world performance.

The results for the five static analyzers are shown in Figure 1b. As described in Section 4.1, we classify detection accuracy into multiple categories: a vulnerability is considered detected if its vulnerability type (CWE) exactly matches the expected CWE (*strict*) or is sufficiently similar (*similar*), based on our CWE similarity criteria (Table 6). For undetected bugs, we track whether the analyzer reported any bug in the vulnerable function (*function*) or missed it entirely (*missed*).

Fuzzers. Looking at Figure 1, among the 112 bugs in Magma, the fuzzer AFL++ finds 37 bugs while the static analyzer CodeQL detects 31. This similar performance between the best tools in each category suggests that both fuzzing and static analysis have similar potential to detect bugs, though there still remains room for improvement for both techniques.

[4] We map each expected vulnerability type (CWE) to up to 27 related vulnerability types. A full list is available in Table 6.



(a) Fuzzing (b) Static Analysis

Fig. 1. Flags per tool and bug class on Magma and CGC.

For the *fuzzers* (Figure 1a), the number of covered and found bugs varies strongly by project but only weakly between fuzzers. In Libxml2, OpenSSL, and PHP, many undiscovered bugs are not even covered by the fuzzer. For Poppler, SQLite3, and Libpng, many bugs are covered but never triggered, suggesting specific failure conditions that fuzzers struggle to overcome.

SAST Tools. The *static analyzers* (Figure 1b) show significant variation in their performance. Their effectiveness differs based on their semantic reasoning capabilities. CodeQL found the most security bugs, likely due to its extensive reasoning capabilities. On the Magma dataset, Flawfinder ranks second but with a notable performance gap. Flawfinder detects potential vulnerabilities by searching for dangerous C standard library functions like `malloc`, `strcpy`, or `sprintf`, without reasoning if the function arguments are properly validated. This explains Flawfinder's poor performance on

the CGC benchmark, which rarely uses standard library functions. Across all analyzers, despite generating 44,332 bug reports, most functions containing security bugs receive no flags, even for unrelated bug types (see *function*).

> *Fuzzers find slightly more Magma bugs than static analyzers. While fuzzers perform similarly to each other, static analyzers show significant variation. The CGC results confirm our findings from Magma.*

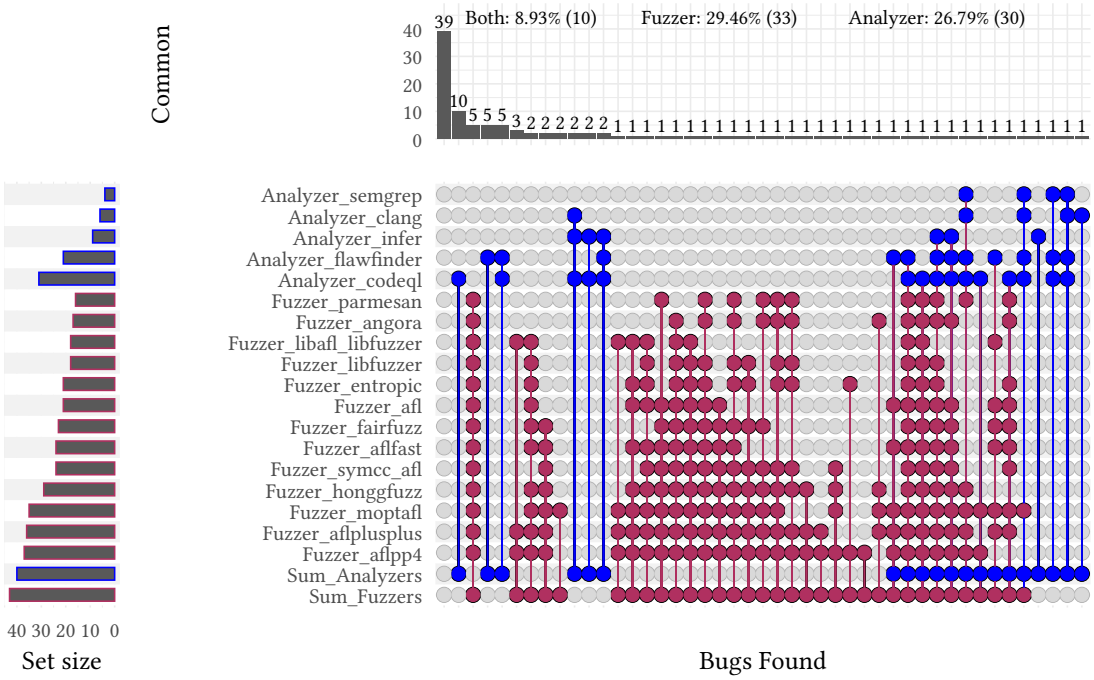


Fig. 2. Intersection plot [49] of Magma CVEs found by each analyzer or fuzzer, as well as by both groups (`Sum_*`). Every row in the intersection stands for the set of bugs found by a tool, and every column is a set of bugs commonly detected by a group of tools. A color-filled circle indicates that a tool is part of a tool group. Bar plots show the respective set sizes for tools and tool groups.

4.3 RQ.2 Complementarity

Figure 2 shows how static analysis and fuzzing complement each other in an UpSet (or intersection) plot [49]. Out of 112 CVEs in Magma, 39 are not found by any approach, 33 are found only by fuzzers, 30 are found only by static analyzers, and 10 are found by both approaches. These results indicate that both approaches find different types of bugs, making them complementary.

Fuzzers. Fuzzers tend to find the same CVEs. 50% of the CVEs are found by at least five fuzzers, and a quarter of those found by *any* fuzzer are found by *all* fuzzers. This homogeneity suggests that using a single well-performing fuzzer like AFL++, which finds 37/43 CVEs, is sufficient.

SAST Tools. For static analysis tools, CodeQL discovers most CVEs. Only Flawfinder finds more than one CVEs not already detected by CodeQL. Since CodeQL significantly outperforms other tools, using CodeQL alone will provide nearly all findings.

Static analysis tools and fuzzers are complementary approaches that find different CVEs. Fuzzers tend to find similar CVEs, so using a well-performing fuzzer like AFL++ is sufficient. Similarly, CodeQL covers most static analysis results, as other tools do not provide comparable results.

4.4 RQ.3 True Positives Across Bug Types

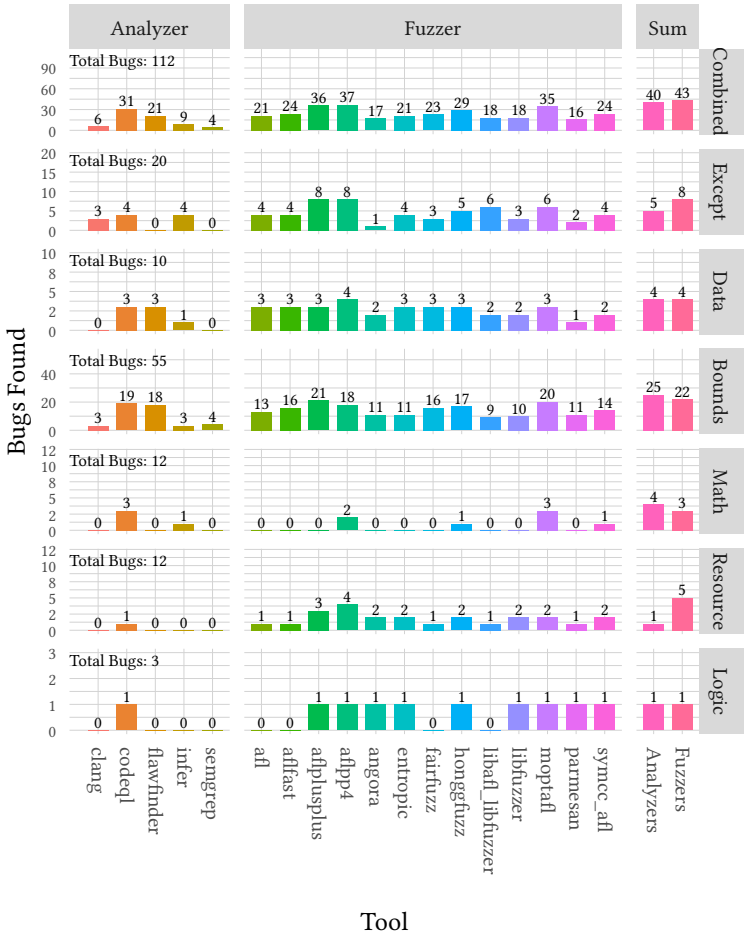


Fig. 3. Detected Magma CVEs per tool across bug types. For fuzzers, a bug is found if it is detected *once*. Bug types (see Table 1) not covered by Magma are omitted.

Figure 3 shows how many of Magma CVEs are detected across the six top-level vulnerability types (root CWEs). These vulnerability types are explained in Table 1. Note that we exclude CGC here, since we only have results for

one fuzzer. A detailed list of CWEs for each code name is available in the Appendix (Table 7). For static analyzers, we consider a CVE detected if its specific vulnerability type (CWE) is similar to the reported vulnerability type (see Section 4.1)

The rows in Figure 3 show results for each vulnerability type and their combined total. The columns show results per tool group and their sum total. Both approaches are moderately successful in detecting *Math*-, *Resource*-, and *Logic*-type vulnerabilities in Magma. However, we are hesitant to draw general conclusions from these results due to the small number of CVEs in each category. The *Math* class consists of division by zero and integer overflow bugs, which often require specific states to trigger. The *Resource* class consists of resource exhaustion (DOS) and use-after-free bugs, requiring specific inputs or understanding of state across function calls. The *Logic* class consists of infinite loop bugs and control flow errors. Fuzzers do not detect these bugs, while static analyzers struggle with the required reasoning. For Math and Resource bug types, fuzzers need to produce specific inputs to trigger the fault, while static analysis tools may not consider these CWEs within their threat model.

SAST Tools. Static analyzers perform best at detecting *Bounds* vulnerabilities and show strong results for *Data* vulnerabilities, though our data is limited. They achieve moderate success with *Except* and *Math* vulnerabilities. *Resource* vulnerabilities are challenging for static analyzers to detect, but our dataset for these is sparse. For *Logic* vulnerabilities, our data is too limited for meaningful conclusions.

Individual static analyzer tool performance varies significantly across analyzers. CodeQL performs best overall, though other tools match its performance for specific bug types. Flawfinder ranks second in overall performance but fails to detect certain bug classes. Clang SA, Infer, and SemGrep perform notably worse than the leading tools, primarily due to their limited detection of bounds-related vulnerabilities and complete misses of certain bug types.

Fuzzers. Fuzzers are most effective at detecting *Except*, *Data*, *Bounds*, and *Resource* vulnerabilities. They are less effective for *Math* vulnerabilities, though our data in this category is limited. Our dataset for *Logic* vulnerabilities is too small to draw meaningful conclusions.

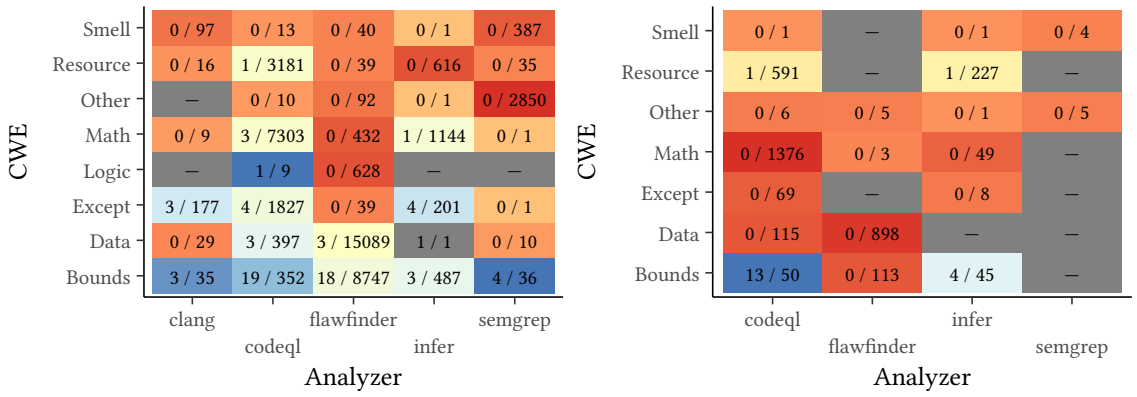
We find that individual fuzzer performance is remarkably similar across different bug classes. The performance of each fuzzer tends to rise and fall together across bug types, suggesting that their effectiveness is not dependent on specific vulnerability types, even though overall performance varies between tools.

While fuzzers perform similarly across bug types, static analysis tools vary greatly in their effectiveness for different kinds of vulnerabilities. Overall performance between fuzzers is relatively consistent, while CodeQL is the clear leader among static analyzers.

4.5 RQ.4 Overhead

After running automated bug finding tools, developers must investigate the results. Since this investigation cannot be fully automated, tools should minimize developer overhead. To quantify this overhead, for SAST tools we analyze the ratio of bug reports (or crashes) to expected bugs in the codebase and for fuzzers we discuss, which normally do not have false positives, we discuss other factors influencing overhead.

SAST Tools. For static analysis tools, the main overhead is sifting through generated bug reports, as these tools are known to generate many false positives [3, 28, 36]. In Figure 4a, we report the number of bug reports per expected CVE across the eight high-level vulnerability types in Magma. While we expect to find 112 CVEs, the five analyzers generate 44 *thousand* bug reports. Clang SA generates fewer reports out-of-the-box, while other tools produce an



(a) Magma Dataset (b) CGC Dataset

Fig. 4. The number of discovered CVEs / generated SAST bug reports across vulnerability types. The color represents the ratio between reports and bugs. A reddish color indicates no true positives, while a blueish color indicates that true positives are found, where a darker blue indicates better ratios. Regarding CGC, Clang SA cannot be used due to the custom standard library, and also, there are no *Logic* bugs, hence we omit the corresponding column and row.

impractical number of reports for developers to investigate (at least in our configuration). Most importantly, *only 0.23%* of bug reports from CodeQL, the best performing tool regarding true positives, correspond to expected Common Vulnerabilities and Exposuress (CVEs). However, quantifying the exact number of false positives is difficult since we do not know how many bugs exist in the base Magma subjects beyond the known forward-ported bugs. Some of the 44k reports may indicate true positives, but analyzing all reports would be infeasible. Managing large numbers of potentially invalid bug reports remains an active research area in software engineering and programming languages [10, 41, 36, 50, 69]. Looking at Figure 4, we can see that tools generate varying numbers of bug reports across vulnerability types. For example, CodeQL reports many *Math* bugs across datasets. Similarly, Flawfinder produces many reports for *Data* and *Bounds* bugs—these include all 21 bugs it finds in Magma, but even with a corresponding high number for CGC no bugs are found (Figure 4b). This is due to the syntactic nature of the tool: Flawfinder reports usage of a set of dangerous functions from the standard library. Thereby, it will miss unsafety in non-standard functions and report safe usages of dangerous functions.

Fuzzers. While fuzzers may miss bugs in individual runs due to their non-deterministic nature, they generally do not produce false positives (depending on the sanitizer). However, when a fuzzer finds a bug, developers have to manually identify the root cause of the crash at hand, a potentially time-intensive task. Furthermore, fuzzers often generate multiple crashing inputs that are caused by the same underlying issue. While these inputs are not false positives, deduplication remains challenging, though recent research has shown promising results in this area [40, 42]. Our evaluation relies on Magma's built-in triggers that simulate a perfect sanitizer, but, in contrast to real sanitizers, do not induce a crash. Therefore, we cannot realistically quantify the number of duplicated crashes in our dataset. Nevertheless, from the developer's perspective, this is no concern: Since all crashes represent true positives, they can arbitrarily analyze and fix one of them, (programmatically) check which inputs still crash, and proceed to investigate this remaining subset.

Static analyzers require significant manual effort to filter false positives. For each CVE in Magma, developers would have to review about a thousand analyzer-generated reports. Fuzzers depend on the quality of sanitizers and bug deduplication tools to minimize effort for the user.

4.6 Threats to Validity

In the following, we outline several potential threats to the validity of our quantitative analysis and describe the measures we take to mitigate them.

External validity refers to the degree to which our results generalize to other programs, analyzers, and fuzzers outside our study but within the scope of this work. Like any empirical study, our results may not generalize beyond our evaluated programs, vulnerabilities, and bug-finding tools. However, our evaluation uses an established benchmark of six real-world programs containing 112 security bugs—the largest benchmark of known vulnerabilities suitable for our study. We include CGC as a control group for additional validation, though it is less realistic and requires manual effort. On the static analysis side, evaluate five widely-used analyzers that represent diverse approaches, including one commercial tool (CodeQL). We study 13 fuzzers, ranging from industry standards to experimental academic implementations.

Internal validity refers to the degree to which our study minimizes potential methodological mistakes. To mitigate internal threats, we defined clear selection criteria (Section 3) for choosing bugs and programs for our evaluation. To mitigate threats of selection and confirmation bias, we required that bugs were collected irrespective of the techniques used to find them. Magma satisfied these criteria. We triangulated these results using a secondary benchmark (CGC).

Construct validity refers to the degree to which a study measures what it claims to measure. To mitigate experimenter bias, we evaluate static analysis in Magma programs by removing all canaries (i.e., Magma-specific preprocessor directives for each bug), while using the canaries as (sanitizer-independent) CVE detectors for fuzzers. To mitigate another threat to construct validity, *we always err in favor of the static analyzer* when automatically identifying detected CVEs. We verify this aspect in Section 5.1. As with any empirical study, there may be errors in our experimental infrastructure or analysis scripts. Thus, to enable independent verification, we make all experimental infrastructure, scripts, and data publicly available.

5 QUALITATIVE EVALUATION

Beyond the quantitative evaluation, we aim to gain a deeper understanding of the utility of static analyzers and fuzzers. We start by validating the results from our automated evaluation on a sample. Afterwards, we focus on two aspects: First, we analyze the types of bugs found by both approaches, the complexity of reasoning required, and how bug detection depends on the execution environment. Second, we evaluate the tools in terms of setup effort, configuration options, and resource requirements. Finally, we look at integration points into development workflows and examine the real-world adoption of security tooling.

5.1 Manual validation

We randomly sample eight CVEs from Magma for detailed manual inspection as follows. For a fair comparison, we require that the root cause and observed crash must be in the same function. This selection criterion simplifies manual verification and benefits static analyzers (which often only work intra-procedurally). There is *no* preference for bugs better found by fuzzing or static analysis. From all bugs that satisfy this criterion, we randomly select eight bugs. For

Table 3. Number of Analyzer Reports per Function Sampled for Manual Analysis

						Manual	
Program	Function	CVE ID	CWE	Flags	Auto[1]	TP	Bug[2]
SQLite3	`tableColumnList`	2017-15286	476	1	1	1	1
Poppler	`Parser::getObj`	2018-13988	125	1	0	0	0
OpenSSL	`X509_NAME_oneline`	2016-2176	119	5	4	0	0
Poppler	`FoFiTrueType::cvtSfnts`	2019-12360	125	22	14	5	0
Libxml2	`xmlStrncat` / `xmlStrncatNew`	2016-1834	119	8	8	7	7
LibTIFF	`TIFFWriteDirectoryTagTransferfunction`	2019-7663	476	1	0	0	0
OpenSSL	`ssl3_read_bytes`	2016-6305	20	2	2	0	0
Libxml2	`xmlStringLenDecodeEntities`	2016-4449	20	9	7	2	0

[1] Using the conservative *Similar* approach in our empirical analysis.
[2] We consider the CVE discovered after careful manual analysis.

each tool, we are interested whether it found the expected bugs (true/false negatives) and whether the reported bugs are valid reports (true/false positives), even if not the expected ones. The functions containing these bugs are listed in Table 3, where each function is associated with a single CVE.

For the selected bugs, we mark source code lines where tools reported bugs based on results from the five static analyzers in our empirical analysis (*report*). We also mark lines that caused the actual bug according to Magma (*fault*) and how the bug was patched (*patch*).

Manual Labeling. Three researchers with expertise in automatic bug finding and system building classified (i) bug reports as true or false positives, reports could be correct even if not identifying the expected bug, and (ii) whether tools found the expected bug. To mitigate threats to internal validity, we follow the standard coding protocol from grounded theory [24]. Each researcher independently assigned labels, spending approximately four hours. The researchers then compared their results and resolved all disagreements during multiple sessions taking another four hours in total. We publish the meeting protocol to ensure verifiability.

Results. Table 3 shows the results of our manual labeling. *Out of 49 static analysis reports, only 8 were manually confirmed to match the known CVE.* For another seven seemingly correct reports, we are uncertain whether they indicate actual vulnerabilities due to the limited scope of this analysis.[5] 34 reports were false positives, a rate of at least 69% in our sample. We also note that our automatic approach (which filters false positives based on similarity to the expected vulnerability type; cf. Section 4.1) *overestimates* the number of true positives in favor of static analysis. We refer to Lipp *et al.* [50] for a detailed analysis of this challenge in static analysis.

The CWE bucketing approach used in Section 4 to evaluate static analysis tools slightly overestimates their true positive rate, erring towards the benefit of the tools as intended.

5.2 Limits to Bug Discovery

In this section, we take a qualitative approach to understand the types of bugs that SAST tools and fuzzers can detect.

[5] CodeQL reports "cpp/missing-negativity-test" for multiple usages of poppler's `seekTable` function as array indices. We are not certain whether these tables are guaranteed to exist; if so, the return value cannot be -1 and the access is safe.

Types of bugs. Static analysis and fuzzing detect different types of bugs. Bugs found by *fuzzing* depend on code coverage and bug oracles (i.e., sanitizers). When a fuzzer generates input that triggers problematic execution, the sanitizer alerts the fuzzer. While sanitizers can detect memory corruption and other issues in C/C++ programs [73], the fuzzing community has called for new sanitizers to detect additional bug types [55]. Without appropriate bug oracles, certain bugs are difficult to detect – particularly logic bugs, which require inferring expected program behavior.

Static analysis uses bug oracles encoded as patterns that are either hard-coded or configurable (as in Flawfinder, Clang SA, and Infer) or fully customizable via rule databases (as in CodeQL and SemGrep). Users can customize bug detection by adding, removing, or modifying these patterns. When new bugs appear in production, corresponding patterns can prevent similar issues in the future. Based on these patterns, static analyzers can detect non-crashing semantic bugs affecting program correctness, such as SQL injection vulnerabilities, information leaks, and logic bugs – issues that fuzzing cannot detect without sanitizer support.

Reasoning. While static analysis predicts problematic states without execution, fuzzing can only detect issues in observed executions. Consequently, fuzzers cannot find bugs in uncovered code. Thus, carefully designed harnesses, good code coverage, and a meaningful seed corpus are crucial for fuzzing [46]. If, for example, integrity checks like checksums are hard to satisfy with random mutations, they may need removal in fuzzing builds to reach deeper code regions.

Static analyzers do not need to execute the program but instead are limited by their reasoning approach. *Syntactic analyzers* check syntactic patterns by comparing code fragments. For instance, Flawfinder matches hard-coded regular expressions to flag problematic function calls (*e.g.*, `sprintf`). This approach is brittle since different syntactic representations can mask issues. *Semantic analyzers* can theoretically reason about program behavior without limitations. However, they must handle large, multi-language codebases with complex invariants. For example, our benchmark OpenSSL (626K LoC) is mainly C code, but 30% consists of Perl preprocessing scripts, making analysis challenging for semantic analyzers.

Environment. Some bugs only manifest in a *specific* program environment. The program environment is determined by factors like the machine architecture, system calls, compiler flags, preprocessor directives, and program configuration. *Fuzzing* only finds bugs in the active environment. For instance, in our setup, all programs are compiled for x86_64, preventing fuzzers from detecting bugs that only occur on x86_32. For example, Magma bug MAE004 / PHP002 (Listing 1 shows the relevant code) is caused by overflowing a 32-bit user-controlled integer on a 32-bit system. On 64-bit systems, however, the value is upcast to 64 bits, preventing the overflow and hiding the bug. Consequently, none of the fuzzers triggered this bug. *Static analysis* can, in theory, reason about programs under all possible environments and configurations. Even for functions with unknown implementations, static analysis can model arbitrary return values. However, this generality leads to more false positives. To increase precision, some static analyzers integrate into the build process to capture more information. In turn, this makes them susceptible to the same environmental limitations as fuzzers.

Static analyzers and fuzzers detect different types of bugs due to their distinct approaches: fuzzers execute *programs, while static analyzers* reason *about them.*

```c
static bool exif_process_IFD_in_TIFF_impl(image_info_type *ImageInfo, size_t dir_offset, int section_index)
{   /* ... */
    // integer overflow if dir_offset=0xFFFFFFFF
    if (ImageInfo->FileSize >= dir_offset+2) {
        int sn = exif_file_sections_add(ImageInfo, M_PSEUDO, 2, NULL);

        // read fails (unchecked) because dir_offset is larger than file size
        php_stream_seek(ImageInfo->infile, dir_offset, SEEK_SET);
        php_stream_read(ImageInfo->infile, (char*)ImageInfo->file.list[sn].data, 2);

        // ImageInfo->file.list[sn].data is uninitialized (CWE-908)
        int num_entries = php_ifd_get16u(ImageInfo->file.list[sn].data, ImageInfo->motorola_intel);
        /* ... */
    } else {
        exif_error_docref(/* ... */);
    }
}
```

Listing 1. Part of PHP's EXIF handling code that contains CVE-2019-9641 [16], which only surfaces if `size_t` is 32 bits.

5.3 Usability and User Considerations

The adoption of bug-finding tools depends not only on their effectiveness, but also on their usability. We discussed the challenges of bug deduplication for fuzzing and false positive reports for static analysis in Section 4.5. Additionally, our manual analysis (Section 5.1) shows that most of the 44 thousand bug reports from the SAST tools are false positives. In this section, we examine three additional qualitative usability factors: (i) setup complexity, (ii) configurability, and (iii) resource requirements.

(i) Setup. Before using an automatic bug-finding tool, it must be set up for the target program. *Static analysis* requires minimal setup effort. Syntactic analysis tools like Flawfinder or SemGrep have no up-front cost and can directly analyze source code. However, semantic analysis tools like Clang SA, Infer, and CodeQL need integration into the project's build process. In our experience, this integration is straightforward, requiring only the build command to be passed to the static analyzers.

Fuzzing has higher upfront costs. The fuzzer's compiler must be integrated into the build process to add coverage instrumentation and possibly sanitizer instrumentation. Additionally, fuzzing requires a fuzz harness, which is often time-consuming and requires project-specific knowledge. The OSS-Fuzz project [75] scales fuzzing across over 1,000 projects, made possible only by maintainers providing fuzz harnesses for their projects. Reducing this effort through automatic harness generation (e.g., using recent advances in code generation models) is a topic of ongoing research [86, 84].

(ii) Tuning. Developers may want to fine-tune bug-finding tools for their specific projects. While tools usually aim to be general-purpose, projects often have differing requirements. *Static analysis* tools are usually highly configurable to support tuning for specific codebases. This configurability helps manage their high false positive rate by allowing developers to disable certain analyses or ignore parts of the codebase. Additionally, tools that decouple their rule set from the analysis engine allow developers to encode custom bug patterns, enabling analysis specific to the codebase. To assess the effort required and flexibility we performed a case study, which we present in the appendix due to its length Appendix A. We extend a CodeQL query to detect a previously undetectable security vulnerability, demonstrating the advantages of this decoupled design.

Similarly, for *fuzzing*, which requires more initial effort to produce first results, due to build integration and harnesses. Fuzzers can be tuned further through dictionaries [20], grammars [64], protocol awareness [65], improved seed corpora, and better fuzz harnesses [74].

(iii) Hardware Resources. Hardware resources are an important consideration when adopting automatic bug-finding tools. Fuzzers are known to require lots of hardware resources, indeed, a typical fuzzer could be run forever since there is no inherent stopping rule. In contrast, *static analysis* terminates once analysis results are generated. To compare these differences concretely, we calculate the approximate cost per bug found for each tool on the Magma bugs. Note that this is a coarse approximation given that many factors need to be taken into account, as our results show, the cost can heavily vary depending on the analyzed subject.

Assuming energy costs of 40 ¢/kWh, we calculate the cost per core hour. The fuzzing CPUs (26 cores/socket, TDP 150 W [37]) cost 0.23 ¢/h/core. The static analysis CPUs (24 cores/socket, TDP 205 W [38]) cost 0.35 ¢/h/core. We estimate the cost per bug by multiplying these rates with the tool runtime and number of harnesses for fuzzing, then dividing by bugs found. We recorded runtimes for each static analyzer and subject (see Table 4). For this calculation, we count analyzer flags using the *similar* matching described in Section 4.1 for static analysis and the *median* run for fuzzers. This results in the following costs: Flawfinder is cheapest at 0.0003 ¢/bug, followed by SemGrep at 0.005 ¢/bug. Semantic analysis tools require more power: Clang SA costs 0.09 ¢/bug, while CodeQL (0.4 ¢/bug) and Infer (1.9 ¢/bug) are the most expensive SAST tools. Fuzzing is even more expensive, with the best performer (AFL++) costing 58.5 ¢/bug. Note that fuzzing costs are expected to grow exponentially for a linear increase in found bugs [4]. Which we can also confirm, for example, the first bug found after 15 s costs much less than the last one found after 32.5 h.

Note that these calculations depend on our evaluation parameters and may not reflect typical usage. Practitioners might terminate SAST tools early when they appear stuck rather than waiting 24 hours. They may also run fewer parallel fuzzing campaigns or extend campaigns beyond 48 hours. Moreover, the results can vary wildly between subjects, though, this variation also depends on the used tool. Nevertheless, these numbers illustrate the relative resource usage differences between tools.

Static analysis tools require less initial effort to set up than fuzzing. However, both tool groups can be extensively tuned to improve their performance. The hardware cost per bug found is approximately two orders of magnitude lower for static analysis tools than for fuzzing.

5.4 Automatic Bug Finding in the Development Process

Bugs are cheaper to fix when caught early in development process, so OWASP recommends applying security tooling early (“shift left” paradigm) [63]. In the following, we examine where fuzzers and SAST tools fit in the development pipeline and investigate how they are used in practice.

Due to its high runtime cost (discussed in the previous section), fuzz testing is typically deployed late in development, typically on already *released* software. For example, OSS-Fuzz [75] mostly tests the public main branches of popular open-source software [27]. Fortunately, our study suggests significant overlap between different fuzzers, making it less important to use multiple fuzzers.

In contrast, static analysis tools can be used earlier in the development pipeline and can easily be integrated into *continuous integration* on popular code sharing platforms [35]. As discussed in Section 5.3, tools can be fine-tuned or

Table 4. CPU Core Time [hh:mm:ss] for SAST Tools

Subject	LoC	semgrep	clang-scan	codeql	infer	flawfinder
libtiff	80K	09	04:07	09:46	09:53	02
libxml2	201K	22	14:37	01:20:54	42:28:18[1]	05
libpng	63K	06	01:42	06:33	09:31	02
php	865K	01:02	09:00	30:22:22[2]	04:56:37[1]	36
sqlite3	445K	01:03	16:43	24	28	12
poppler	290K	02	01:27[3]	31:29	20:21	06
openssl	626K	35	26:49	05:27:30	58:16	07
Total		03:19	01:14:25	37:58:58	49:03:24	01:10

[1] killed out-of-memory
[2] timeout after 24 hours real time
[3] failed to capture build system

limited to recently changed code to reduce reports while still improving code quality incrementally. However, our results presented in Section 4.5 suggests that SAST tools are not utilized in Magma projects, given the large number of reports generated.

Taking a broader perspective, the results from fuzzers and SAST tools hint at a fundamental limitation of memory-unsafe languages like C and C++. These tools are essentially trying to detect bugs that the language itself would be best equipped to prevent. In the case of memory errors, for instance, C and C++ rely heavily on the programmer's diligence, as ownership cannot be checked at compile time for these languages [39]. In contrast, modern systems programming languages address this issue directly through their design: languages like Go use automatic memory management, while Rust enforces explicit ownership checking at compile time. Although these approaches involve trade-offs, they effectively eliminate memory safety violations through *language design* rather than relying on external tools to catch such errors after the fact.

However, even in C/C++, some bugs can still be caught *at compile time*. Using a current language version and enabling compiler warnings[6] is an important first step to detect issues early—for example, the Linux kernel team upgraded from C89 to C11 in 2022 to support block-level variable declarations after a vulnerability [13]. Addressing compiler warnings also reduces noise from subsequent SAST analysis. Our analysis found SAST flags for unused variables, indicating these basic steps are sometimes skipped even in popular open-source projects.

CI Pipeline Analysis. Motivated by the insight that (at time of inclusion into Magma), some of the projects had apparently not used bug-finding tools as part of the development process, we want to quantify the in-practice usage of such tools. Notably, Beller *et al.* [3] found in 2016 that almost half of the open source projects they surveyed used static analysis, but only sporadically. To get a sense of how commonly fuzzers and SAST tools are used in the most critical 100 OSS projects today (as per OpenSSF), we manually analyzed all 57 OSS projects that are hosted on GitHub (to focus on one CI pipeline). A project is counted as using fuzzing if the word "fuzz" is mentioned in the GitHub repository's CI configuration or a corresponding entry in OSS-Fuzz exists. Similarly, usage of SAST tools is counted if the name of a SAST tool is mentioned in the CI configuration. Both of these occurrences are manually double-checked. As shown in Figure 5, we empirically found that 21 of the 57 analyzed projects use both fuzzing and SAST tools, 8 use only fuzzing,

[6] In GCC and Clang, developers can set the language version to C11 (`-std=c11`), enable warnings with `-Wall -Wextra` and optionally, request strict ISO C checking with `-Wpedantic` [22].

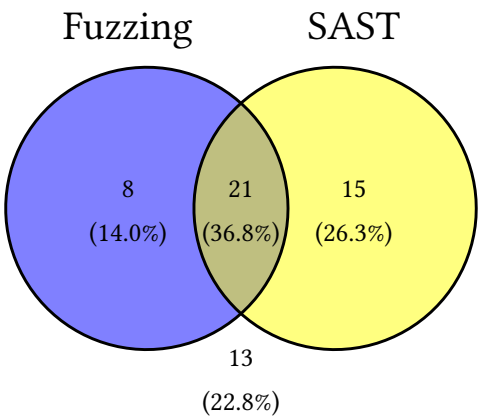


Fig. 5. Projects using Fuzzing or SAST tools during CI.

15 use only SAST tools, and 13 projects use neither of the two methods. Our complete analysis results are available in the paper's code repository.

Static analysis tools can be used as part of the continuous integration pipeline, while fuzzers are typically used later in the development process. Of the most security-critical OSS projects, nearly 80% use a fuzzer or SAST tool as part of their continuous integration pipeline.

6 DISCUSSION

Based on our qualitative and quantitative analyses, we provide advice for software project maintainers looking to improve security, and discuss potential improvements for fuzzing and static analysis research.

6.1 Recommendations for Project Maintainers

To minimize security flaws in deployed code, OWASP [63] recommends"shifting left", i.e., applying security tooling early in the development cycle. The tools we study can be integrated either directly after running tests (syntactic SAST tools) or as longer-running asynchronous checks (fuzzing tools and semantic SAST tools). We can confirm that running SAST tools early in the pipeline before fuzzing is more economical given the estimated cost per bug. Even though fuzzers are generally more computationally expensive, tools in both groups can be used as part of continuous integration [12]. However, for C/C++ projects specifically, SAST tools can produce too many false positives to be practical. One mitigation is to analyze only newly added or modified code to avoid being overwhelmed. Additionally, most tools allow for customization which can be used to disable or adapt bug reports. This customization is especially important for classes of reports that have too much noise to provide value for the user. For example, the largest number of reports for CodeQL stem from checks for *Math* bugs, disabling these reports would more than halve the number of total reports while only minimally affecting true positives. Even though a single tool (CodeQL) provides nearly all true positives among the tested tools, without any mitigation, the number of bug reports would make its use impractical.

In contrast, fuzzers only produce true positive results but need more computational resources and require a more elaborate initial setup such as writing harnesses and build integration. Again, a single tool (AFL++) will find nearly all bugs that fuzzers would find.

Since SAST tools and fuzzers find different types of bugs with minimal overlap, they complement each other. While fuzzers are generally recommended if computational resources are available, combining them with SAST tools can be beneficial despite the additional manual effort required.

6.2 Future Directions for Research Communities

Based on the quantitative results (Section 4) and our case studies (Section 5), we see several opportunities for future improvements for both research communities.

Comparing SAST tools and fuzzers, the found bugs are quite disjunct. While not entirely surprising, this is an interesting finding, that should motivate future work to combine the two approaches (e.g., guiding the fuzzing process with information obtained during static analysis). Thus, we challenge both research communities to find ways to combine the two approaches. While fuzzing has already been combined with symbolic execution [66], we believe there is still potential for other innovative combinations. For example, there already is work on reducing false positive reports of SAST tools by using fuzzing to double-check the results [59]. It might also be feasible to use the extensive number of rules available in SAST tools to guide fuzzing (explored in [51]) or to create new sanitizers.

Regarding SAST tools, in line with previous research, we see a vast amount of false positives, which will pose a serious hurdle to adoption of SAST tools in practice. Regarding fuzzers, we see the required computational resources and the manual effort to write harnesses as the main hurdles to adoption in practice. However, since compute power is relatively cheap compared to developer hours and the setup is a one-time effort, compared to SAST tools, fuzzing seems to be more reasonable to get actionable results on vulnerable C/C++ code.

Benchmarks for Static Analysis. We find that a fundamental problem for static analysis research is the lack of real-world benchmarks. The fuzzing community has Magma [32] and Fuzzbench [56], which provide benchmarks based on real-world programs. While not perfect, as manual bug selection can introduce bias, these benchmarks contain realistic targets. However, to our knowledge, no equivalent exists for SAST tools, which are typically evaluated on small artificial code snippets like Juliet [61]. Such a benchmark does not represent realistic programs, which can have deeply nested code, as well as complex and stateful bugs.

To create an effective benchmark, we suggest developing a programmatically accessible bug reporting format first. This format should include ways to classify bugs, possibly with different accuracy levels for more accurate tool assessment, and to specify bug locality, possibly across multiple locations. Without such a format, a benchmark for SAST tools would be impractical, as only programmatic evaluation enables automated and reproducible assessment.

For bug classification, the tools in our study support Static Analysis Results Interchange Format (SARIF) [19] reports. However, this format allows custom strings with bug descriptions that require manual mapping to CWEs. An adapted SARIF format facilitating programmatic evaluation could be a viable approach. However, CWEs have limitations: it is often unclear which CWE to assign to a bug, since CWEs overlap and can be very generic. This requires clustering to group *similar* CWEs. While CWEs are hierarchically organized, the resulting clusters are too coarse, which required us to develop a stricter relatedness clustering for our study (see Appendix B, Table 6).

For bug location, tools report line numbers, which we mapped to functions for coarse-grained analysis. Information on a level of affected variables would provide better precision. Complex bugs may appear in multiple code locations,

potentially far from their root cause, and a fair benchmark should handle these cases. The benchmark design must also prevent gaming the system—a tool that flags every line or variable provides no value.

7 CONCLUSION

In this work, we present the first systematic comparison of static analysis and fuzzing for addressing typical C/C++ security weaknesses in real-world scenarios. Quantitatively, we find that fuzzers detect slightly more vulnerabilities, with relatively little variation between tools. This consistency is expected, as most modern fuzzers are derivatives of AFL. Notably, fuzzers with sophisticated techniques, such as Angora or SymCC-AFL, often perform worse than conceptually simpler approaches. In contrast, static analyzers show greater performance variation. Notably, CodeQL detects more true positives than other tools, likely due to its extensive library of security analysis queries and the resources behind its commercial development. An unexpected finding is that fuzzing and SAST tools find mostly disjoint sets of bugs. This suggests that, to maximize true positive results, developers should use both types of tools in tandem for a more comprehensive bug detection strategy.

While CodeQL leads the SAST field in true positive reports, these come at the cost of many false positives. Indeed, both leading tools, CodeQL and Flawfinder, generate an overwhelming number of false positives. Worryingly, all static analysis tools suffer from a very high false positive rate, well above the 15–20% threshold considered acceptable by developers [10]. While customizing static analysis tools for specific projects can improve precision (*e.g.*, by disabling Math and Resource checks in CodeQL), our data suggests the resulting rates would still exceed this threshold. Note that we focus on C/C++ code in this work, which is particularly challenging for static analysis. In contrast, fuzzers demonstrate more practical utility despite requiring greater computational resources and initial effort for writing fuzzing harnesses. This can be attributed to their dynamic nature—unlike static analyzers, they do not reason about a program but only observe execution side effects, namely crashes, which prevents false positive reports. Given that developers prefer investing more analysis time for more precise results [10], fuzzing seems to be a more suitable approach in practice.

We also investigated the adoption of SAST tools and fuzzers in security-critical open-source projects and found that most projects use at least one of these approaches. However, our findings reveal a limitation: neither static analysis nor fuzzing can prove the absence of memory vulnerabilities, as shown by the high number of false negatives observed in our study. To address this fundamental issue, transitioning to safer programming languages and adopting robust design principles would provide a more reliable way to prevent these vulnerabilities by construction.

ACKNOWLEDGMENTS

This work was funded by the European Research Council (ERC) under the consolidator grant RS^3, no. 101045669 (https://doi.org/10.3030/101045669).

REFERENCES

[1] @LazyFishBarrel. 2019. #Memoryunsafety. Twitter. Retrieved Feb. 2, 2025 from https://twitter.com/LazyFishBarrel/status/1129000965741404160.

[2] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. 2008. Using Static Analysis to Find Bugs. *IEEE Software*, 25, 5, 22–29. DOI: 10.1109/MS.2008.130.

[3] 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 *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 470–481. DOI: 10.1109/SANER.2016.105.

[4] Marcel Böhme and Brandon Falk. 2020. Fuzzing: on the exponential cost of vulnerability discovery. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 713–724. DOI: 10.1145/3368089.3409729.

[5] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: an information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 678–689. DOI: 10.1145/3368089.3409748.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 1032–1043. DOI: 10.1145/2976749.2978428.

[7] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering*, 1621–1633. DOI: 10.1145/3510003.3510230.

[8] George Chatzieleftheriou and Panagiotis Katsaros. 2011. Test-Driving Static Analysis Tools in Search of C Code Vulnerabilities. In *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, 96–103. DOI: 10.1109/COMPSACW.2011.26.

[9] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*, 711–725. DOI: 10.1109/SP.2018.00046.

[10] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 332–343. DOI: 10.1145/2970276.2970347.

[11] Clang. 2023. Clang Analyzer: Checker Developer Manual. Retrieved Feb. 2, 2025 from https://clang-analyzer.llvm.org/checker_dev_manual.html.

[12] ClusterFuzzLite Contributors. 2024. Clusterfuzzlite - simple continuous fuzzing that runs in ci. Retrieved Feb. 2, 2025 from https://github.com/google/clusterfuzzlite.

[13] Jonathan Corbet. 2022. Moving the kernel to modern C. LWN.net. Retrieved Feb. 2, 2025 from https://lwn.net/Articles/885941/.

[14] Addison Crump, Andrea Fioraldi, Dominik Maier, and Dongjia Zhang. 2023. LIBAFL LIBFUZZER: LIBFUZZER on Top of LIBAFL. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, 70–72. DOI: 10.1109/SBFT59156.2023.00021.

[15] Jose D'Abruzzo Pereira and Marco Vieira. 2020. On the Use of Open-Source C/C++ Static Analysis Tools in Large Projects. In *2020 16th European Dependable Computing Conference (EDCC)*, 97–102. DOI: 10.1109/EDCC51268.2020.00025.

[16] Chamal Desilva. 2019. Sec Bug #77509: Uninitialized read in exif_process_IFD_in_TIFF. Retrieved Feb. 2, 2025 from https://bugs.php.net/bug.php?id=77509.

[17] Zhen Yu Ding and Claire Le Goues. 2021. An Empirical Study of OSS-Fuzz Bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 131–142. DOI: 10.1109/MSR52588.2021.00026.

[18] Facebook. 2022. Infer. Retrieved Feb. 2, 2025 from https://fbinfer.com.

[19] Michael Fanning and Laurence J. Golding. 2018. Static Analysis Results Interchange Format (SARIF) Version 2.0. Retrieved Feb. 2, 2025 from https://docs.oasis-open.org/sarif/sarif/v2.0/csprd01/sarif-v2.0-csprd01.html.

[20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

[21] Alex Gaynor. 2019. Introduction to Memory Unsafety for VPs of Engineering. alexgaynor.net. Retrieved Feb. 2, 2025 from https://alexgaynor.net/2019/aug/12/introduction-to-memory-unsafety-for-vps-of-engineering/.

[22] GCC team. 2024. Options to Request or Suppress Warnings. GCC online documentation. Retrieved Feb. 2, 2025 from https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html.

[23] GitHub. 2021. CodeQL. Retrieved Feb. 2, 2025 from https://codeql.github.com.

[24] Barney G. Glaser and Anselm L. Strauss. 2017. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. (1st ed.). DOI: 10.4324/9780203793206.

[25] Google. 2022. Go Fuzzing. GO. Retrieved Feb. 2, 2025 from https://go.dev/security/fuzz/.

[26] Google. 2010. Honggfuzz. Retrieved Feb. 2, 2025 from https://github.com/google/honggfuzz.

[27] Google. 2024. OSS-Fuzz. Retrieved Feb. 2, 2025 from https://github.com/google/oss-fuzz.

[28] Anjana Gosain and Ganga Sharma. 2015. Static Analysis: A Survey of Techniques and Tools. In *Intelligent Computing and Applications*. Vol. 343. Durbadal Mandal, Rajib Kar, Swagatam Das, and Bijaya Ketan Panigrahi, (Eds.), 581–591. DOI: 10.1007/978-81-322-2268-2_59.

[29] Katerina Goseva-Popstojanova and Andrei Perhinschi. 2015. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68, 18–33. DOI: 10.1016/j.infsof.2015.08.002.

[30] Andrew Habib and Michael Pradel. 2018. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 317–328. DOI: 10.1145/3238147.3238213.

[31] Rowan Hart. 2023. CGC Challenges. Retrieved Feb. 2, 2025 from https://github.com/novafacing/cgc-challenges.

[32] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4, 3, 1–29. DOI: 10.1145/3428334.

[33] HexHive. 2020. Frequently Asked Questions | magma. Retrieved Feb. 2, 2025 from https://hexhive.epfl.ch/magma/docs/faq.html.

[34] HexHive. 2023. MAGMA: Survival Report. Retrieved Feb. 2, 2025 from https://hexhive.epfl.ch/magma/reports/sample_2/.

[35] Justin Hutchings. 2020. Code scanning is now available! GitHub Blog. Retrieved Feb. 2, 2025 from https://github.blog/news-insights/product-news/code-scanning-is-now-available/.

[36] Nasif Imtiaz, Akond Rahman, Effat Farhana, and Laurie Williams. 2019. Challenges with Responding to Static Analysis Tool Alerts. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 245–249. DOI: 10.1109/MSR.2019.00049.

[37] Intel Corporation. 2020. Intel Xeon Gold 6230R Processor. Intel Product Specifications. Retrieved Feb. 2, 2025 from https://ark.intel.com/content/www/us/en/ark/products/199346/intel-xeon-gold-6230r-processor-35-75m-cache-2-10-ghz.html.

[38] Intel Corporation. 2020. Intel Xeon Gold 6248R Processor. Intel Product Specifications. Retrieved Feb. 2, 2025 from https://ark.intel.com/content/www/us/en/ark/products/199351/intel-xeon-gold-6248r-processor-35-75m-cache-3-00-ghz.html.

[39] Jansens, Dana and Anforowicz, Lukasz and Palmer, Chris. 2021. Borrowing Trouble: The Difficulties Of A C++ Borrow-Checker. Retrieved Feb. 2, 2025 from https://docs.google.com/document/d/e/2PACX-1vSt2VB1zQAJ6JDMaIA9PlmEgBxz2K5Tx6w2JqJNeYCy0gU4aoubdTxlENSKNSrQ2TXqPWcuwtXe6PlO/pub.

[40] Zhiyuan Jiang, Xiyue Jiang, Ahmad Hazimeh, Chaojing Tang, Chao Zhang, and Mathias Payer. 2021. Igor: Crash Deduplication Through Root-Cause Clustering. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 3318–3336. DOI: 10.1145/3460120.3485364.

[41] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, 672–681. DOI: 10.1109/ICSE.2013.6606613.

[42] Ashwin Kallingal Joshy and Wei Le. 2022. FuzzerAid: Grouping Fuzzed Crashes Based On Fault Signatures. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 1–12. DOI: 10.1145/3551349.3556959.

[43] Arvinder Kaur and Ruchikaa Nayyar. 2020. A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code. *Procedia Computer Science*, 171, 2023–2029. DOI: 10.1016/j.procs.2020.04.217.

[44] Paul Kehrer. 2019. Memory Unsafety in Apple's Operating Systems. Langui.sh. Retrieved Feb. 2, 2025 from https://langui.sh/2019/07/23/apple-memory-safety/.

[45] Sunghun Kim and Michael D. Ernst. 2007. Which warnings should I fix first? In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 45–54. DOI: 10.1145/1287624.1287633.

[46] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *CCS '18: 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2123–2138. DOI: 10.1145/3243734.3243804.

[47] Ted Kremenek and Dawson Engler. 2003. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *Static Analysis*. Vol. 2694. Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, and Radhia Cousot, (Eds.), 295–315. DOI: 10.1007/3-540-44898-5_16.

[48] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 475–485. DOI: 10.1145/3238147.3238176.

[49] Alexander Lex, Nils Gehlenborg, Hendrik Strobelt, Romain Vuillemot, and Hanspeter Pfister. 2014. UpSet: Visualization of Intersecting Sets. *IEEE Transactions on Visualization and Computer Graphics*, 20, 12, 1983–1992. DOI: 10.1109/TVCG.2014.2346248.

[50] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 544–555. DOI: 10.1145/3533767.3534380.

[51] Stephan Lipp, Severin Kacianka, Alexander Pretschner, and Marcel Böhme. SAST-Guided Grey-Box Fuzzing. (2024).

[52] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security Symposium*. Vol. 14, 271–286.

[53] LLVM. 2020. Clang Static Analyzer. Retrieved Feb. 2, 2025 from https://clang-analyzer.llvm.org.

[54] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, 1949–1966.

[55] Jonathan Metzman, Dongge Liu, and Oliver Chang. 2022. Fuzzing beyond memory corruption: Finding broader classes of vulnerabilities automatically. Google Security Blog. Retrieved Feb. 2, 2025 from https://security.googleblog.com/2022/09/fuzzing-beyond-memory-corruption.html.

[56] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: an open fuzzer benchmarking platform and service. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1393–1403. DOI: 10.1145/3468264.3473932.

[57] Microsoft. 2022. /fsanitize (Enable sanitizers). Microsoft Learn. Retrieved Feb. 2, 2025 from https://learn.microsoft.com/en-us/cpp/build/reference/fsanitize?view=msvc-170.

[58] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33, 12, 32–44. DOI: 10.1145/96267.96279.

[59] Aniruddhan Murali, Noble Mathews, Mahmoud Alfadel, Meiyappan Nagappan, and Meng Xu. 2024. FuzzSlice: Pruning False Positives in Static Analysis Warnings through Function-Level Fuzzing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13. DOI: 10.1145/3597503.3623321.

[60] Tukaram Muske, Rohith Talluri, and Alexander Serebrenik. 2018. Repositioning of static analysis alarms. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 187–197. DOI: 10.1145/3213846.3213850.

[61] NIST. 2017. Juliet C/C++ 1.3. Retrieved Feb. 2, 2025 from https://samate.nist.gov/SARD/test-suites/112.

[62] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, 2289–2306.

[63] OWASP. 2021. Application Security Verification Standard. Retrieved Feb. 2, 2025 from https://github.com/OWASP/ASVS.

[64] Van-Thuan Pham, Marcel Boehme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2020. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering*, 1–1. DOI: 10.1109/TSE.2019.2941681.

[65] Van-Thuan Pham, Marcel Bohme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 460–465. DOI: 10.1109/ICST46399.2020.00062.

[66] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, 181–198.

[67] Clyde Rodriguez and Collin Greene. 2022. How Meta and the security industry collaborate to secure the internet. Engineering at Meta. Retrieved Feb. 2, 2025 from https://engineering.fb.com/2022/07/20/security/how-meta-and-the-security-industry-collaborate-to-secure-the-internet/.

[68] 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 13th International Conference on Software Engineering - ICSE '08*, 341. DOI: 10.1145/1368088.1368135.

[69] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at Google. *Communications of the ACM*, 61, 4, 58–66. DOI: 10.1145/3188720.

[70] Moritz Schloegel et al. 2024. SoK: Prudent Evaluation Practices for Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*, 1974–1993. DOI: 10.1109/SP54263.2024.00137.

[71] Semgrep. 2020. Semgrep OSS. Retrieved Feb. 2, 2025 from https://github.com/semgrep/semgrep.

[72] Semgrep. 2020. Semgrep rules. Retrieved Feb. 2, 2025 from https://github.com/semgrep/semgrep-rules.

[73] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 309–318.

[74] Kosta Serebryany. 2016. Continuous Fuzzing with libFuzzer and AddressSanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, 157–157. DOI: 10.1109/SecDev.2016.043.

[75] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. Retrieved Feb. 2, 2025 from https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany.

[76] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *2019 IEEE Symposium on Security and Privacy (SP)*, 1275–1295. DOI: 10.1109/SP.2019.00010.

[77] Adrian Taylor, Bartek Nowierski, and Kentaro Hara. 2022. Use-after-freedom: MiraclePtr. Google Security Blog. Retrieved Feb. 2, 2025 from https://security.googleblog.com/2022/09/use-after-freedom-miracleptr.html.

[78] Adrian Taylor, Andrew Whalley, Dana Jansens, and Nasko Oskov. 2021. An update on memory safety in Chrome. Google Security Blog. Retrieved Feb. 2, 2025 from https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html.

[79] 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 27th IEEE/ACM International Conference on Automated Software Engineering*, 50–59. DOI: 10.1145/2351676.2351685.

[80] Jeff Van der Stoep and Chong Zhang. 2019. Queue the Hardening Enhancements. Google Security Blog. Retrieved Feb. 2, 2025 from https://security.googleblog.com/2019/05/queue-hardening-enhancements.html.

[81] Various. 2023. Github: analysis-tools-dev/static-analysis. Retrieved Feb. 2, 2025 from https://github.com/analysis-tools-dev/static-analysis#c.

[82] 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 *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 38–49. DOI: 10.1109/SANER.2018.8330195.

[83] David A. Wheeler. 2001. Flawfinder. Retrieved Feb. 2, 2025 from https://dwheeler.com/flawfinder/.

[84] Jiahao Yu, Yangguang Shao, Hanwen Miao, Junzheng Shi, and Xinyu Xing. 2024. PROMPTFUZZ: Harnessing Fuzzing Techniques for Robust Testing of Prompt Injection in LLMs. arXiv: 2409.14729 [cs].

[85] Michał Zalewski. 2017. American Fuzzy Lop. Technical Whitepaper. Retrieved Feb. 2, 2025 from https://lcamtuf.coredump.cx/afl/technical_details.txt.

[86] Cen Zhang, Yaowen Zheng, Mingqiang Bai, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. 2024. How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1223–1235. DOI: 10.1145/3650212.3680355.

[87] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J.P. Hudepohl, and M.A. Vouk. 2006. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32, 4, 240–253. DOI: 10.1109/TSE.2006.38.

[88] Misha Zitser, Richard Lippmann, and Tim Leek. 2004. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, 97–106. DOI: 10.1145/1029894.1029911.

```
1 idx = (Elf_Internal_Group *) shdr->contents;
2 n_elt = shdr->sh_size / 4;
3
4 while (--n_elt != 0) {
5     if ((++idx)->shdr == hdr) {...}
6 }
```

Listing 2. NULL pointer dereference vulnerability (CVE-2017-13710) in Binutils.

```
1 from VariableAccess va
2 where
3     maybeNull(va) and
4     dereferenced(va)
5 select va, "Value may be null; it should be checked before dereferencing."
```

Listing 3. Default CodeQL rule to detect NULL pointer dereference vulnerabilities.

A CASE STUDY: FINE-TUNING CODEQL

CodeQL is a powerful tool that allows developers to add custom queries to find new kinds of bugs. In this section, we explore what it takes to add a missed CVEs into search space of CodeQL by designing a custom query, and what exactly is the cost (in terms of new false positives). But first we find out why this CVE was missed in the first place.

Vulnerability. Listing 2 shows the vulnerability we seek to investigate: a NULL pointer dereference in the Binutils (version 2.29) function `setup_group` that is publicly reported as `CVE-2017-13710`. Specifically, the root cause of this vulnerability is the assignment in Line 2 where — depending on the program execution — the field `shdr->sh_size` and thus the value of `n_elt` can be 0. As a result, due to the prefix decrement operation performed on `n_elt` within the loop condition in Line 4, the condition always holds true. This results in infinitely many iterations of the loop body. Eventually, we exceed the bounds of the allocated memory for the header contents (*i.e.*, `idx` in Line 1) when iterating over the contents in line 5. This results in a NULL pointer dereference (and likely a program crash) when trying to access the member variable `shdr`.

While this vulnerability is fairly easy to understand for a human, it is very difficult to find by a static analyzer. This is because the manifestation of the vulnerability depends on the interplay of two different variables, `n_elt` and `idx`, which is difficult to find out unless the concrete program semantics are properly interpreted (without executing the program).

Existing bug pattern. To detect NULL pointer dereferences, CodeQL provides the query shown in Listing 3. In general, the static analysis engine in CodeQL is decoupled from the actual rules. For this, CodeQL first generates an abstract code model of the target program while attaching to the build process. Once this code abstraction is fully built, queries can be run against this model, searching for different code properties, including security vulnerabilities, depending on the rule. These rules are written in CodeQL's specific query language, which resembles Structured Query Language (SQL) (except the reversed query order).

Line 1 in Listing 3 specifies the source data of this query, namely all variable accesses in the analyzed program. Similar to SQL, the `where` clause allows filtering the sourced data points based on the predicates specified. In our example, `maybeNull` (provided by CodeQL's standard library) in line 3 filters for all variables access where the value of the (pointer) variable may be NULL, using a simple data-flow analysis. The second predicate, `dereferenced` in line 4, further filters the results of the previous step to only return variable accesses that perform dereferencing operations.

```
1  from
2      PointerFieldAccess fa, /* v->f       */
3      PrefixCrementOperation co,
4      PointerArithmeticOperation ao
5  where
6      fa.getAChild() = co or /* v = --/++p */
7      fa.getAChild() = ao    /* v = p ⊗ n  */
8  select fa.getAChild(), "Field access involving pointer arithmetic; potential NULL pointer dereference"
```

Listing 4. Our new query to detect `CVE-2017-13710`.

The `select` statement in line 5 selects certain properties of the result set; here, it outputs the name and location of the respective variables for which a NULL check is missing along with an error message.

Same bug. Different structure. When running this query on the vulnerable Binutils version, CodeQL reports 414 findings. Yet, the NULL pointer dereference does *not* appear among those findings. We found that neither `maybeNull` nor `dereferenced` when used alone, flagged this vulnerable code location. For the latter, the default data-flow analysis provides only limited support, or rather it becomes inaccurate for more complex data structures like nested C structs. Even more interesting, the dereference `(++idx)->shdr` was also not detected as such. Digging deeper, we found that `dereferenced` works with regular dereference operations like `*v` and `v->f`, but not if the dereference is directly performed after a pointer arithmetic expression as in Listing 3. Accordingly, if the prefix pointer increment plus field access had been split over two separate statements, `dereferenced` would have marked the corresponding line. Note that this difference is only syntactical. The semantic meaning, and thus the vulnerability, is preserved. Also, `maybeNull` still misses the line. This example illustrates that the same vulnerability can have multiple different code representations, where only a subset of those may be supported by the default queries provided by CodeQL.

Successful bug pattern. In order to detect different code variants of the NULL pointer dereference in Binutils, we need to create another query that identifies field accesses directly succeeding pointer arithmetic. Listing 4 shows an example of such a query. It queries all pointer field accesses (Line 2) where the pointer variable contains a prefix in- or decrement operation, or is involved in any other arithmetic computation (Lines 6 & 7).

Extra cost. Extra benefit. Running this query on the Binutils codebase yields 24 additional bug reports, including `CVE-2017-13710` and other possibly dangerous but probably benign pointer dereferences.

We were able to extend the default query to match the target bug pattern, which demonstrates a use case for the "shift-left" approach: Found bugs can be used to improve the bug-finding pipeline.

B DATA ON BUG REPORTS, CWES AND BUCKETING

In this section, we present the total number of SAST reports (Table 5) and our custom CWE bucketing used for automatic evaluation of these reports (Table 6). Furthermore, we provide an exhaustive list of CWEs in our study and the *bug type* we assign to it in Table 7.

Table 7. The CWEs Belonging to Different Bug Types, Closely Modeled After the CWE Hierarchy

Name	CWE	Description
Except	703	Improper Check or Handling of Exceptional Conditions

Continued on next page

Table 7. The CWEs Belonging to Different Bug Types, Closely Modeled After the CWE Hierarchy (Continued)

Name	CWE	Description
Except	754	Improper Check for Unusual or Exceptional Conditions
Except	755	Improper Handling of Exceptional Conditions
Except	252	Unchecked Return Value
Except	476	NULL Pointer Dereference
Data	707	Improper Neutralization
Data	20	Improper Input Validation
Data	129	Improper Validation of Array Index
Data	134	Use of Externally-Controlled Format String
Data	73	External Control of File Name or Path
Data	807	Reliance on Untrusted Inputs in a Security Decision
Data	643	Improper Neutralization of Data within XPath Expressions ('XPath Injection')
Data	170	Improper Null Termination
Data	233	Improper Handling of Parameters
Data	466	Return of Pointer Value Outside of Expected Range
Data	99	Improper Control of Resource Identifiers ('Resource Injection')
Data	117	Improper Output Neutralization for Logs
Bounds	119	Improper Restriction of Operations within the Bounds of a Memory Buffer
Bounds	125	Out-of-bounds Read
Bounds	126	Buffer Over-read
Bounds	787	Out-of-bounds Write
Bounds	120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
Bounds	823	Use of Out-of-range Pointer Offset
Bounds	805	Buffer Access with Incorrect Length Value
Bounds	788	Access of Memory Location After End of Buffer
Bounds	786	Access of Memory Location Before Start of Buffer
Bounds	121	Stack-based Buffer Overflow
Math	682	Incorrect Calculation
Math	131	Incorrect Calculation of Buffer Size
Math	369	Divide By Zero
Math	190	Integer Overflow or Wraparound
Math	191	Integer Underflow (Wrap or Wraparound)
Math	193	Off-by-one Error
Math	1077	Floating Point Comparison with Incorrect Operator
Math	468	Incorrect Pointer Scaling
Math	467	Use of sizeof() on a Pointer Type
Math	189	Numeric Errors

Continued on next page

Table 7. The CWEs Belonging to Different Bug Types, Closely Modeled After the CWE Hierarchy (Continued)

Name	CWE	Description
Math	195	Signed to Unsigned Conversion Error
Math	196	Unsigned to Signed Conversion Error
Resource	664	Improper Control of a Resource Through its Lifetime
Resource	401	Missing Release of Memory after Effective Lifetime
Resource	415	Double Free
Resource	416	Use After Free
Resource	681	Incorrect Conversion between Numeric Types
Resource	770	Allocation of Resources Without Limits or Throttling
Resource	772	Missing Release of Resource after Effective Lifetime
Resource	399	Resource Management Errors
Resource	457	Use of Uninitialized Variable
Resource	244	Improper Clearing of Heap Memory Before Release ('Heap Inspection')
Resource	665	Improper Initialization
Resource	22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
Resource	789	Memory Allocation with Excessive Size Value
Resource	775	Missing Release of File Descriptor or Handle after Effective Lifetime
Resource	732	Incorrect Permission Assignment for Critical Resource
Resource	843	Access of Resource Using Incompatible Type ('Type Confusion')
Resource	771	Missing Reference to Active Allocated Resource
Resource	377	Insecure Temporary File
Resource	590	Free of Memory not on the Heap
Resource	908	Use of Uninitialized Resource
Resource	826	Premature Release of Resource During Expected Lifetime
Resource	704	Incorrect Type Conversion or Cast
Resource	456	Missing Initialization of a Variable
Resource	672	Operation on a Resource after Expiration or Release
Resource	570	Expression is Always False
Resource	824	Access of Uninitialized Pointer
Resource	822	Untrusted Pointer Dereference
Smell	710	Improper Adherence to Coding Standards
Smell	561	Dead Code
Smell	563	Assignment to Variable without Use
Smell	685	Function Call With Incorrect Number of Arguments
Smell	676	Use of Potentially Dangerous Function
Smell	686	Function Call With Incorrect Argument Type
Smell	690	Unchecked Return Value to NULL Pointer Dereference

Continued on next page

Table 7. The CWEs Belonging to Different Bug Types, Closely Modeled After the CWE Hierarchy (Continued)

Name	CWE	Description
Smell	628	Function Call with Incorrectly Specified Arguments
Smell	398	Code Quality
Smell	758	Reliance on Undefined, Unspecified, or Implementation-Defined Behavior
Smell	571	Expression is Always True
Smell	562	Return of Stack Variable Address
Smell	587	Assignment of a Fixed Address to a Pointer
Smell	242	Use of Inherently Dangerous Function
Logic	691	Insufficient Control Flow Management
Logic	670	Always-Incorrect Control Flow Implementation
Logic	835	Loop with Unreachable Exit Condition ('Infinite Loop')
Logic	362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Logic	367	Time-of-check Time-of-use (TOCTOU) Race Condition
Logic	768	Incorrect Short Circuit Evaluation
Logic	364	Signal Handler Race Condition
Logic	611	Improper Restriction of XML External Entity Reference
Other	290	Authentication Bypass by Spoofing
Other	494	Download of Code Without Integrity Check
Other	327	Use of a Broken or Risky Cryptographic Algorithm
Other	497	Exposure of Sensitive System Information to an Unauthorized Control Sphere
Other	402	Transmission of Private Resources into a New Sphere ('Resource Leak')
Other	78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
Other	780	Use of RSA Algorithm without OAEP
Other	328	Use of Weak Hash
Other	829	Inclusion of Functionality from Untrusted Control Sphere
Other	560	Use of umask() with chmod-style Argument
Other	760	Use of a One-Way Hash with a Predictable Salt
Other	297	Improper Validation of Certificate with Host Mismatch
Other	15	External Control of System or Configuration Setting
Other	250	Execution with Unnecessary Privileges
Other	785	Use of Path Manipulation Function without Maximum-sized Buffer
Other	114	Process Control
Other	272	Least Privilege Violation
Other	89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
Other	14	Compiler Removal of Code to Clear Buffers
Other	201	Insertion of Sensitive Information Into Sent Data
Other	862	Missing Authorization

Table 5. The number of reports created by each SAST tool and in total.

Analyzer	Magma	CGC	Total
clang	363	0	363
codeql	13092	2208	15300
flawfinder	25106	1019	26125
infer	2451	331	2782
semgrep	3320	9	3329
Total	44332	3567	47899

Table 6. Custom CWE Bucketing Used in Our Analysis

Expected	Similar Vulnerability Types (CWEs)
CWE-20	20, 119, 120, 125, 126, 129, 134, 170, 190, 196, 466, 786, 787, 788, 805, 823
CWE-129	20, 119, 120, 125, 126, 129, 786, 787, 788, 805, 823
CWE-119	119, 120, 125, 126, 190, 196, 676, 788, 805
CWE-120	119, 120, 125, 786, 787, 788, 805, 823
CWE-121	119, 120, 121, 125, 190, 196, 468, 682, 786, 788, 823
CWE-125	119, 120, 125, 126, 190, 196, 468, 786, 787, 788, 805, 823
CWE-126	119, 120, 125, 126, 131, 190, 196, 786, 787, 788, 805, 823
CWE-787	119, 120, 125, 190, 196, 676, 786, 787, 788, 805, 823
CWE-131	119, 120, 125, 126, 131, 467, 468, 682, 786, 787, 788, 805, 823
CWE-369	369, 457, 682
CWE-190	20, 190, 196, 682
CWE-401	401, 404, 772, 775
CWE-415	415, 416, 672
CWE-416	415, 416, 672
CWE-457	457
CWE-681	195, 196, 197, 681, 704, 1077
CWE-770	20, 119, 120, 125, 129, 134, 170, 190, 466, 770, 786, 787, 788, 789, 805, 823
CWE-772	401, 404, 772, 775
CWE-476	457, 476, 690
CWE-835	196, 682, 834, 835
CWE-670	617, 670
CWE-755	703, 755
CWE-754	252, 703, 754
CWE-399	399, 400, 402, 770, 834, 835
CWE-664	399, 400, 402, 664, 770, 834, 835
CWE-189	128, 190, 191, 193, 196, 369, 839, 1335, 1339, 1389
CWE-201	201
CWE-824	457, 824
CWE-822	119, 822
CWE-611	611

C ADDITIONAL FIGURES

In this section, we present alternative versions for the figures from the quantitative evaluation (Section 4). In Figure 6, we show the distribution of bugs found per bug type for CGC, analogously to Figure 3, where we look at Magma only.

Figure 7 shows the bug overlap between tools for the Magma and CGC dataset combined, analogously to Figure 2, which looks at Magma only. Additionally, we provide a CGC-only version in Figure 8.

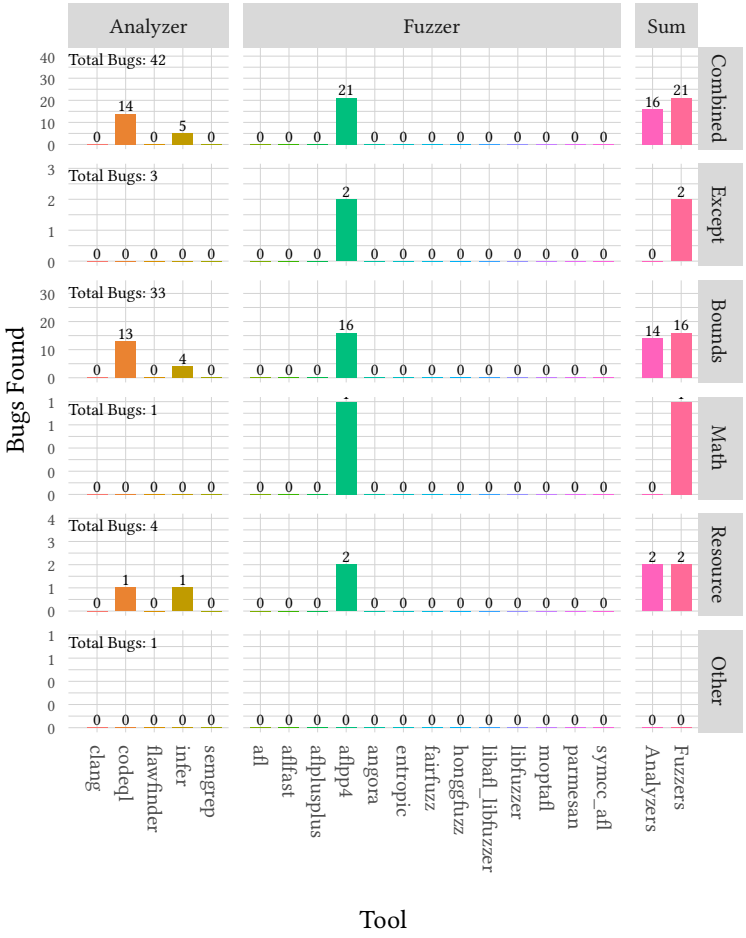


Fig. 6. Total bugs found only for CGC.

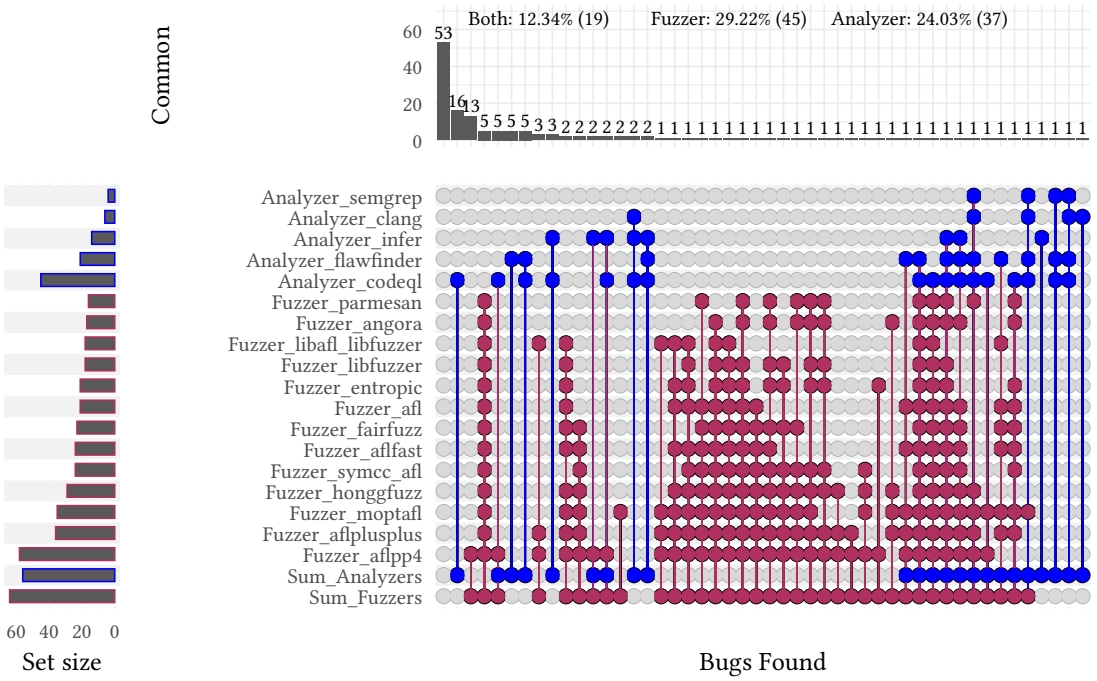


Fig. 7. Upset plot to show overlap of bugs for both Magma and CGC.

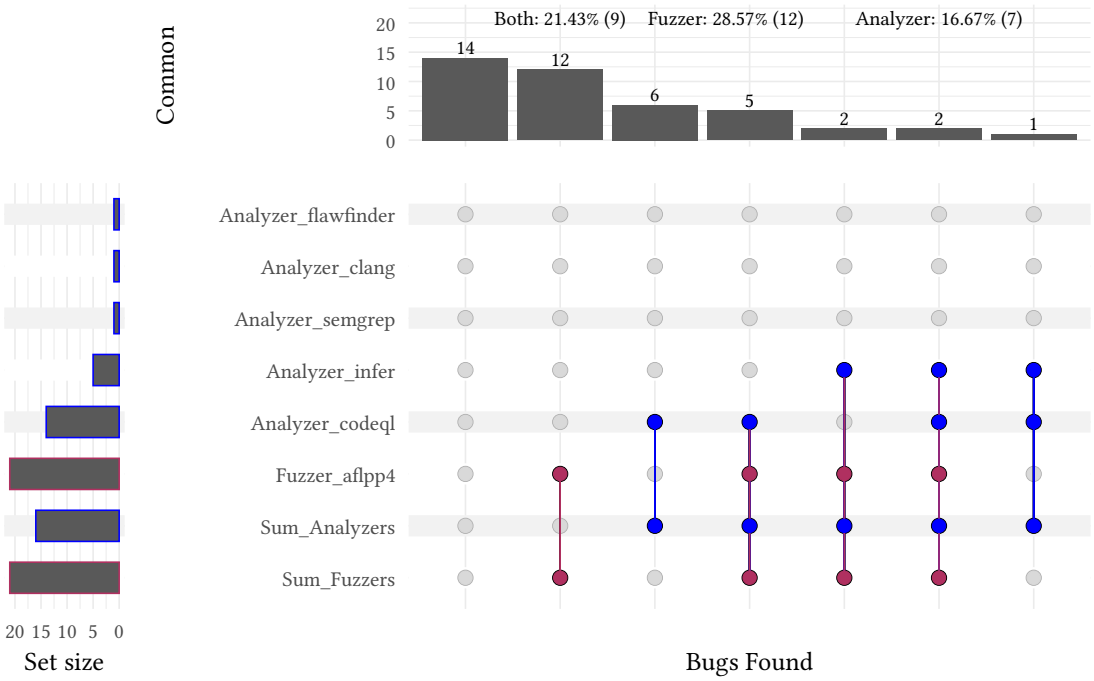


Fig. 8. Upset plot to show overlap of bugs for CGC.